

# 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** | **8/18/2024** | **Eric Ross** |  |

## 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

Eric Ross

## Algorithm Cipher

I recommend the encryption algorithm cipher to deploy for Artemis Financial is AES-256. AES is one of the most widely used and secure encryption algorithms today. AES is known for its efficiency in both hardware and software implementations, making it ideal for securing data at rest and in transit. It is also highly resistant to all known cryptographic attacks, which is why it is widely adopted across industries, including finance. AES supports key sizes of 128, 192, and 256 bits. The larger the key size, the more secure the encryption, but it may also require more processing power. For Artemis Financial, a 256-bit key is recommended due to its high-security requirements. AES is not only secure but also performs well in various environments, including constrained devices like mobile phones or IoT devices, which is essential for a modern web application like Artemis Financial’s. AES also meets the encryption requirements of many industry standards, such as PCI-DSS for payment card data and GDPR for personal data protection.

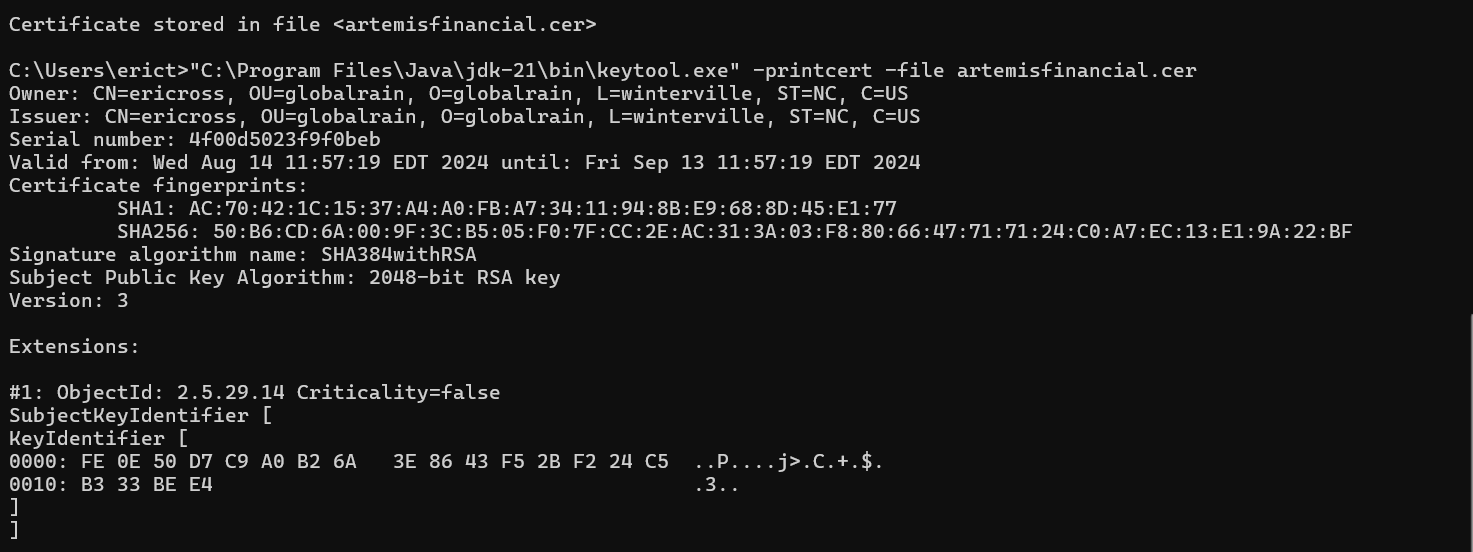
A commonly used hash function in conjunction with AES is SHA-256. It produces a fixed-size 256-bit hash value, regardless of the input data size. This hash can be used to verify the integrity of data transmitted over insecure channels. A 256-bit encryption provides a higher level of security at the cost of additional computational resources. Given Artemis Financial’s focus on security, a 256-bit key is recommended. AES uses a process called key expansion, where the original encryption key is expanded into multiple round keys. The number of rounds depends on the key length which would be 14 rounds for 256-bit.

Using random numbers in encryption for Initialization Vectors (IV) make sure that the same plaintext encrypted multiple times will produce different ciphertexts. IVs must be random and unique for each encryption session. The security of the encryption heavily depends on the randomness of the keys. A cryptographically secure pseudorandom number generator should be used for key generation to ensure unpredictability. AES uses symmetric keys so both the sender and receiver share the same key for encryption and decryption. It is efficient and fast but requires secure key distribution. On the other hand, RSA uses asymmetric keys which are a pair of keys that consist of a public key for encryption and a private key for decryption. It is more secure for key exchange but slower than symmetric encryption. Asymmetric encryption is often used to securely exchange symmetric keys. In secure web communications such as HTTPS, asymmetric encryption is used to securely exchange a symmetric AES key, which is then used for the actual data encryption.

The Data Encryption Standard (DES), which is the predecessor of the Advanced Encryption Standard (AES), was once the standard but became vulnerable due to its short key length of 56 bits. AES was introduced to replace DES with stronger security measures. AES was selected after a rigorous process involving cryptographic experts worldwide. It was designed to be secure against all known attacks and scalable for future needs. AES is now the default choice for most encryption needs, from securing sensitive data in databases to protecting communications over the internet. Cryptographic algorithms are continually evaluated by the cryptographic community. There is ongoing research into post-quantum cryptography, as future quantum computers could potentially break current encryption methods. However, AES with a 256-bit key is considered secure against quantum attacks for the foreseeable future.

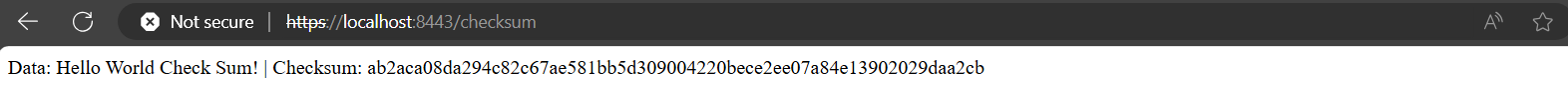
## Certificate Generation

Insert a screenshot below of the CER file.



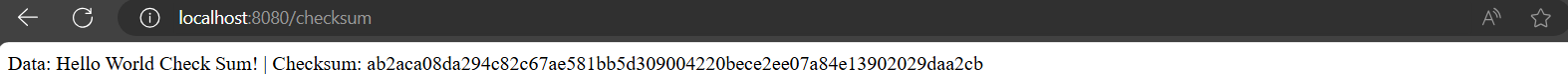
## Deploy Cipher

Insert a screenshot below of the checksum verification.



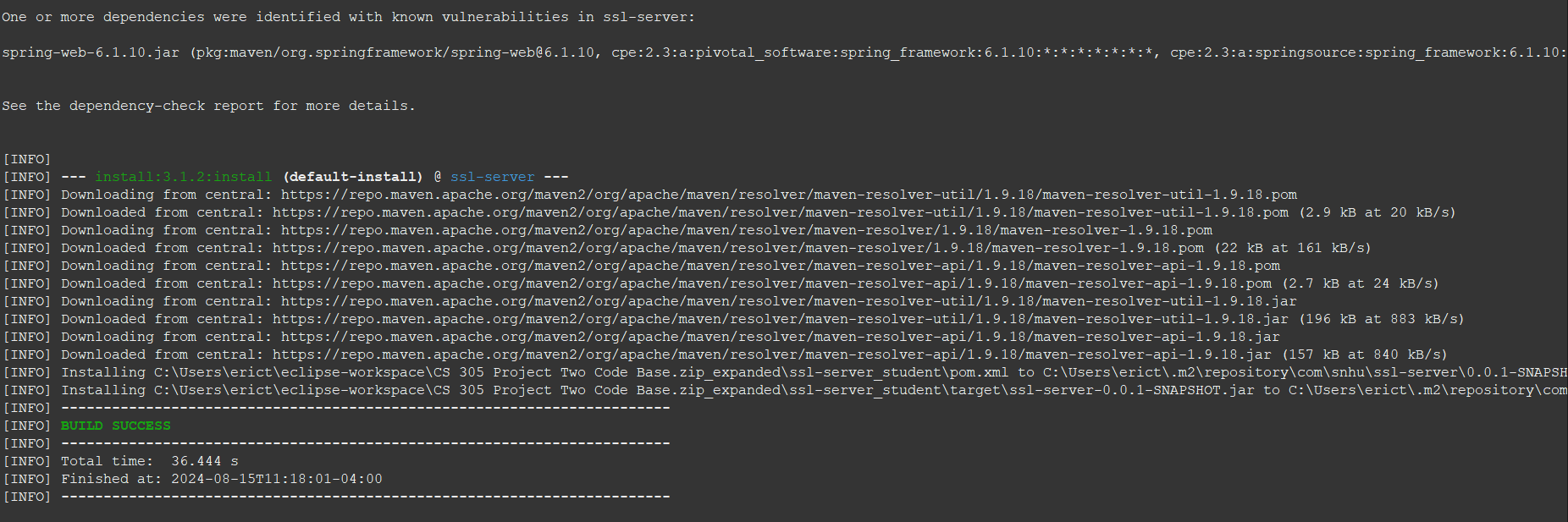
## Secure Communications

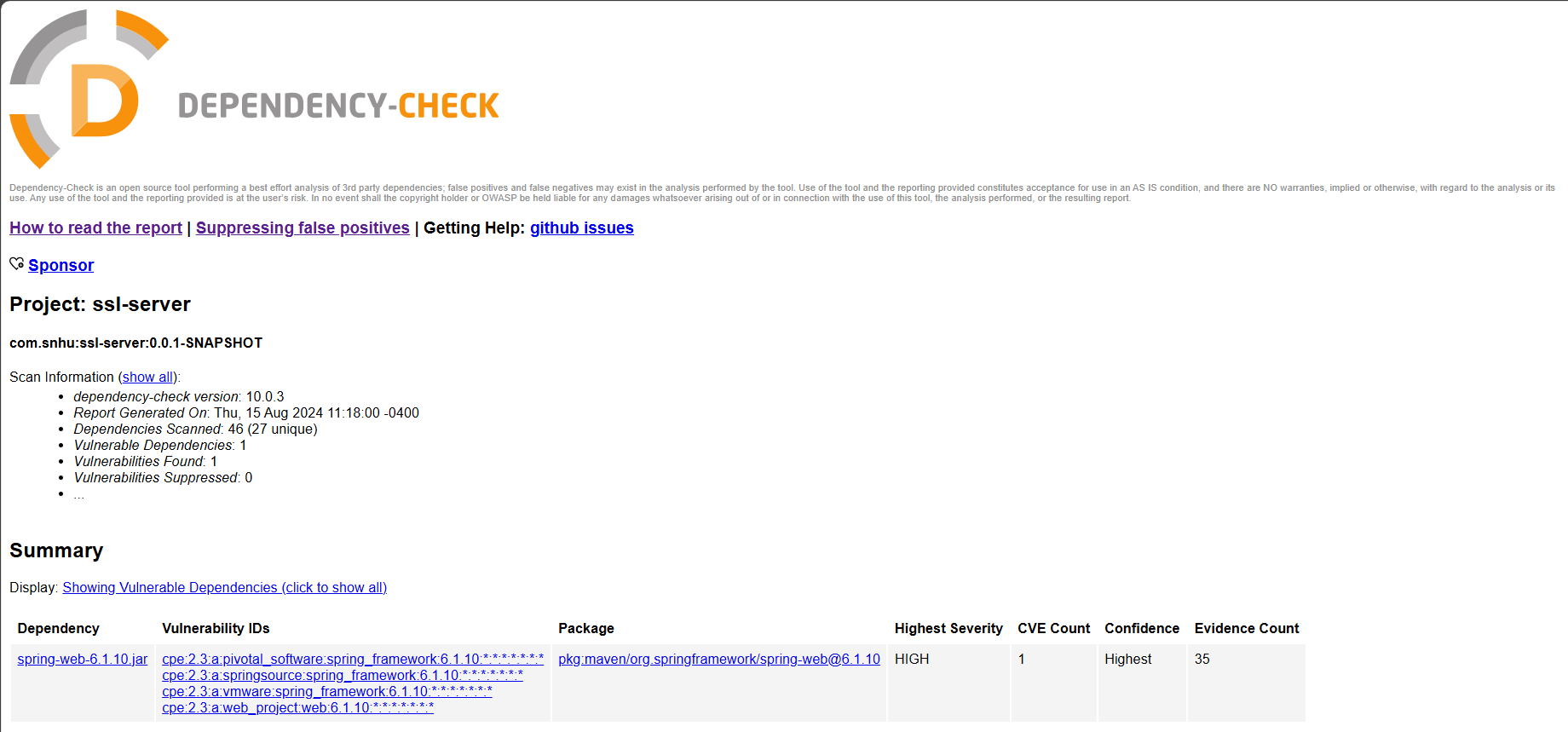
Insert a screenshot below of the web browser that shows a secure webpage.



## Secondary Testing

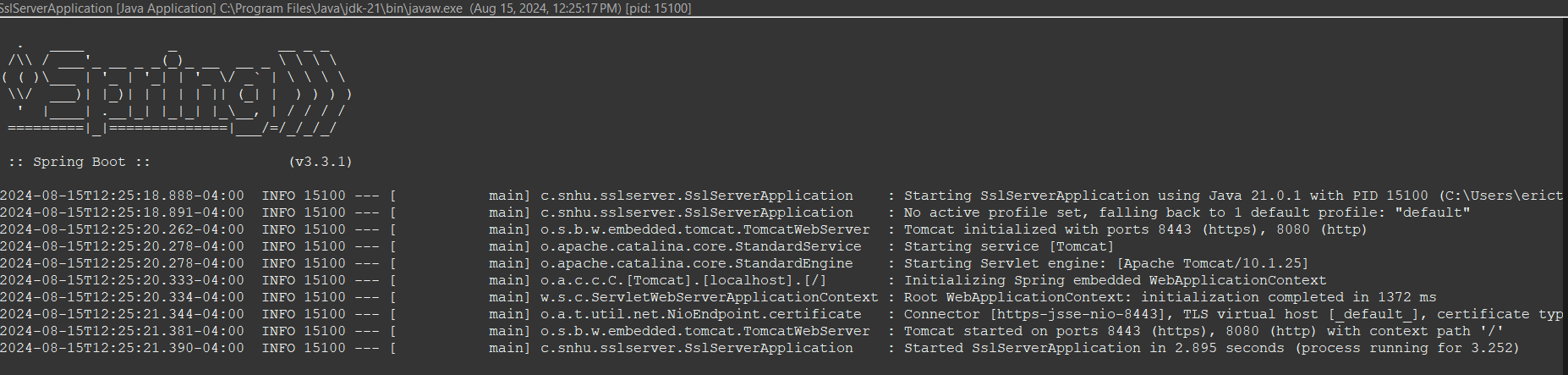
Insert screenshots below of the refactored code executed without errors and the dependency-check report.





## Functional Testing

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



## Summary

During the process of developing and refactoring the code, I made sure that the input data being handled, specifically in the checksum calculation process, was validated properly. While the data used in our example was the static string "Hello World Check Sum!", the design of the ‘ChecksumController’ allows for flexibility in future modifications where dynamic user input might be validated before processing. Proper input validation is critical to prevent vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and buffer overflows. In this project, making sure that any future input is validated before being processed into the checksum will protect the application from potential exploits. Throughout the refactoring process, I adhered to secure coding practices such as encapsulation and error handling. By encapsulating the checksum generation logic within a private method ‘getSHA256Checksum’, I ensured that the implementation details were hidden from other parts of the application. I used a try-catch block within the checksum generation method to handle any potential exceptions. This prevents the application from crashing and allows for proper error logging. By encapsulating logic and handling errors this way, we can reduce the likelihood of vulnerabilities being introduced into the codebase. I implemented the SHA-256 algorithm for generating a checksum of the data string. SHA-256 is a widely recognized and secure hashing algorithm that ensures data integrity. Cryptographic algorithms like SHA-256 are essential for protecting data integrity. By generating a checksum, I made sure that the data could be verified to detect any unauthorized modifications. This is critical for applications dealing with sensitive data. I configured the application to redirect all HTTP requests to HTTPS by adding the ‘HttpToHttpsRedirectConfig’ class. This will make sure that all data transmitted between the client and server is encrypted. Redirecting HTTP to HTTPS is vital for preventing man-in-the-middle attacks. By enforcing HTTPS, we can make sure that data in transit is encrypted, thus protecting it from interception and tampering. Throughout the project, I reviewed different parts of the code to ensure compliance with security best practices. Specifically, I focused on the ‘ChecksumController’ and the HTTPS redirection configuration. Regular code reviews are essential for catching security issues early in the development process. By reviewing the code, we can identify potential areas for improvement and make sure that refactored code adheres to security standards.

When adding layers of security to the software application, the initial code provided a basic structure for the Spring Boot application. I refactored the code by adding a checksum generation method using the SHA-256 algorithm and configuring HTTPS for secure communication. The checksum generation adds a layer of data integrity verification, while the HTTPS configuration ensures secure data transmission. These enhancements make the application more resilient against common security threats. After refactoring the code, I used the Dependency-Check tool to perform static analysis on the project's dependencies. This allowed me to identify vulnerabilities in third-party libraries, specifically a known vulnerability in the ‘spring-web’ library. By identifying vulnerabilities in dependencies, we can take steps to mitigate risks, such as updating the vulnerable libraries or applying security patches. Static analysis is a critical step in making sure that external components do not introduce security risks. I conducted functional testing by running the application and verifying that the refactored code executed without errors. This included testing the checksum generation and HTTPS redirection functionalities. I conducted functional testing by running the application and verifying that the refactored code executed without errors. This included testing the checksum generation and HTTPS redirection functionalities. Functional testing makes sure that the application behaves as expected and that the security features implemented worked correctly. This step validates the effectiveness of our security enhancements. I iteratively improved the application’s security by continuously testing and refining the code. Security is an ongoing process, and iterative improvements are necessary to maintain a secure application. By revisiting and refining our security measures, we can make sure that the application remains secure against evolving threats. I documented each step of the process, including the security enhancements, testing results, and any identified vulnerabilities. Proper documentation provides a clear record of the security measures implemented and allows for easier future audits and assessments. It also makes sure that any identified vulnerabilities are tracked and addressed appropriately.

## Industry Standard Best Practices

Throughout the project, I adhered to secure coding practices to maintain and enhance the application’s security. This included encapsulating sensitive operations, such as checksum generation, within private methods and made sure that error handling was implemented securely. Secure coding practices are a fundamental aspect of software development and are recommended by standards such as OWASP (Open Web Application Security Project). These practices help prevent common vulnerabilities like buffer overflows, improper error handling, and insecure cryptographic operations. By following these practices, we are able to maintain the integrity and confidentiality of the application, making sure that it remains resilient against potential attacks. The use of the SHA-256 algorithm, for example, provided a secure method for verifying data integrity. I configured the application to use HTTPS for secure communication, redirecting all HTTP traffic to HTTPS. This makes sure that data transmitted between the client and server is encrypted. Using HTTPS is a standard best practice recommended by security frameworks like SSL/TLS (Secure Sockets Layer/Transport Layer Security) and enforced by modern web browsers. It protects against data interception and man-in-the-middle attacks. The application’s security is significantly enhanced by making sure that all data in transit is encrypted, thus safeguarding sensitive information from potential eavesdroppers. I used the Dependency-Check tool to identify and address vulnerabilities in third-party libraries. This allowed me to pinpoint a known vulnerability in the ‘spring-web’ library. Regularly scanning dependencies for vulnerabilities is recommended by industry standards such as OWASP and NIST (National Institute of Standards and Technology). Keeping dependencies up to date and secure is critical in maintaining the overall security of the application. By identifying and addressing vulnerabilities in external libraries, we can make sure that the application is not compromised by insecure third-party components.

Applying industry-standard best practices makes sure that the company’s software is developed with security as a core consideration. This will reduce the likelihood of security breaches, which can result in significant financial and reputational damage. By following best practices, the company can maintain strong security while protecting its assets, client data, and intellectual property from malicious attacks. Many industries are subject to regulatory requirements that mandate the use of secure coding practices. By adhering to these practices, the company ensures compliance with legal standards, avoiding fines and legal complications. Compliance with regulations not only avoids potential legal penalties but also builds trust with clients and stakeholders, showing that the company takes security seriously. Secure coding practices contribute to overall software quality. Applications built with security in mind are generally more reliable, with fewer bugs and vulnerabilities. High-quality, secure software enhances user satisfaction and reduces maintenance costs, as fewer security incidents need to be managed or remediated post-deployment. By proactively addressing security risks through industry-standard best practices, the company can mitigate potential threats before they are exploited. This includes risks associated with both internal development and third-party components. Mitigating risks early in the development process reduces the cost and impact of potential security incidents, allowing the company to focus on innovation and growth rather than crisis management.