****

# 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 6](#_Toc102040759)

[3. Deploy Cipher 6](#_Toc102040760)

[4. Secure Communications 6](#_Toc102040761)

[5. Secondary Testing 7](#_Toc102040762)

[6. Functional Testing 8](#_Toc102040763)

[7. Summary 9](#_Toc102040764)

[8. Industry Standard Best Practices 12](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **October 11, 2023** | **Matthew Pool** |  |

## Client



## Developer

Matthew Pool

## Algorithm Cipher

For secure transferring of encryption/decryption keys, RSA (using 2048-bits if possible) meets security standards for industry best practices. RSA can be integrated with SHA-256 (of the SHA-2 family) to implement a cryptographic hashing algorithm that creates a checksum (hash value) that can be used to verify the integrity of the key. It should be noted that if other data (including a simple text string) is intended to be made available for download to users, then AES-256 (or similar) should be used for such data; however, for distributing keys, RSA is ideal.

RSA is an asymmetric encryption cipher algorithm commonly used for public/private key exchange and digital signatures and is typically used with CAs and initial TLS/SSL (HTTPS) handshakes. When implemented with (SHA-256) SHA-2 or SHA-3 encryption, a checksum can be calculated for the key and downloaded by the receiving client. Client software can then calculate the expected checksum using the same SHA hash algorithm and verify it matches the downloaded checksum to verify its integrity. At this point, AES can be used to encrypt a data file (or simple text string) for a client to download, alongside a checksum created to verify the file’s integrity. Along with implementing a trusted Certificate Authority (CA), this strategy offers a robust cryptographic solution for the provided scenario!

One major rule in hash function design is including the property of collision-resistance, in which it is computationally infeasible to find two distinct bit strings that produce the same hash value output:

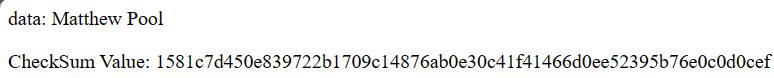
Older versions of hash functions like MD5 and SHA-1 are quite susceptible to multi-collision attacks, which exposes data to unauthorized access, allows forgery of digital signatures, and allows man-in-the-middle attacks to tamper with legitimate data without detection. Modern hash functions like SHA-256 are much more mathematically secure, as it makes it computationally infeasible to find collisions by using more rounds/layers for added complexity and resistance to attacks.

## Certificate Generation

A screenshot of a computer program

Description automatically generated

## Deploy Cipher



## Secure Communications

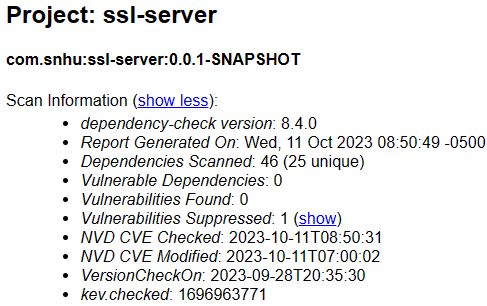
A blue rectangle with white text

Description automatically generated

## Secondary Testing

A screenshot of a computer

Description automatically generated



A close up of a text

Description automatically generated

## Functional Testing

A screen shot of error message

Description automatically generated

A screenshot of a computer

Description automatically generated

## Summary

The SslServerApplication.java file required creating a new @RestController class that I named SslServerController, as well as a new route using @RequestMapping(“/hash”) that called a getHash function. I implemented a try/catch block within the getHash method to catch “NoSuchAlgorithmException” and other errors. I used a secure pseud-random number generator (SecureRandom) along with KeyGenerator to create a 256-bit AES key to create the cipher algorithm for the data String. This creates a byte array, which is then converted to Base64 (test-based string), which is then passed into a SHA-256 MessageDigest instance to be stored in its own byte array. Then, this byte array (checksum) is converted into a Hexadecimal String (hexString) using a private helper function (bytesToHex) to be returned and presented in the client’s web browser, along with the original data String.

Artemis Financial is a financial planning/consulting company that operates on a global level and is responsible for quite an impressive and diverse portfolio, including savings and retirement accounts, investments, and insurance. At Global Rain, our mission statement says, “Security is everyone’s responsibility”, and we truly believe that! New security vulnerabilities are continually being discovered and new, innovative, and creative ways to counter them are constantly being created. Because Artemis Financial works at an international level involving financial transactions and government agencies (and other businesses), their “rest-service” software requires more than the typical RESTful web Application Programming Interface (API), such as the compliancy with the following domestic and international laws (including financial sector regulations):

Financial & International Government Standards & Regulatory Compliance:

* **(GLBA) Gramm-Leach-Bliley Act**: U.S. Law for financial info management & data protection
* **(SOX) Sarbanes-Oxley Act**: U.S. law for data integrity & confidentiality
* **(MiFID II) Markets in Financial Instruments Directive II**: European transparency regulations
* **(PSD2) Revised Payment Service Directive**: European directive for payment services
* **(GDPR**) **General Data Protection Regulation**: European Union law for data protection
* **(FISMA**) **Federal Information Security Management Act**: U.S. law for federal data systems
* Other industry- and regional-specific laws **(HIPAA (Health Insurance Portability & Accountability Act)**, etc.**)**

AES-256 encryption is a modern encryption standard commonly used for encrypting data at rest and in transit. SHA-256 is a modern, secure hash algorithm used for hashing and data integrity verification. These two cryptographic technologies provide a safe and proven methodology for securing RESTful web application data.

The following areas of security from the Vulnerability Assessment Process Flow Diagram (VAPFD), provided by Southern New Hampshire University (SNHU), were addressed in this application code:

* **Controllers**: handle HTTP requests and routing
* **APIs**: getHash() and bytesToHex()
* **Models**: data encryption and hashing/validation
* **View**: graphical output of text (even error messages) in client browser
* **Plugins**:
  + org.owasp dependency-check-maven
  + org.owasp maven-enforcer-plugin
  + org.springframework.boot spring-boot-maven-plugin
  + org.codehaus.mojo versions-maven-plugin
  + org.apache.maven.plugins maven-site-plugin
* *No Data Access*: no external databases utilized
* *No Services*: Spring Framework created so no refactorization needed

Securing a software application requires multiple layers of security, especially with web applications, as web apps typically have numerous components interacting with each other. Proper authentication and authorization checks should typically be integrated at various endpoints, as well as input validation/sanitization and rate limiting implementation. With the Artemis Financial web app, a 2048-bit RSA encryption and a strong password were used in creating the CA (Certificate Authority) and initiating the TLS (SSL) connection from the client browser to the web server. The application.properties file was edited to utilize port 8443 for TLS (Transport Layer Security) and HTTPS (Hypertext Transfer Protocol Secure) to provide a secure communications channel.

Additionally, updates were made to the Spring Framework, OWASP Dependency-Check, Maven, and other libraries, dependencies, and plugins by running appropriate CLI (Command Line Interface) commands and editing the pom.xml file. OWASP’s Dependency-Check was also used to identify any known dependency vulnerabilities that existed within the project. Finally, before presenting (or transmitting) any sensitive data to the client’s browser, AES-256 encryption was used to encrypt the data, which was then hashed using the SHA-256 hashing algorithm. These multiple layers of security are necessary to provide protection against various exploits that exist inherently in RESTful web software applications.

## Industry Standard Best Practices

Industry standard best practices for secure coding and vulnerability mitigation can be found from entities such as the following:

* OWASP (Open Web Application Security Project)
* NIST (National Institute of Standards and Technology)
* CERT (Computer Emergency Response Team)
* ISO (International Organization for Standardization) /

IEC (International Electrotechnical Commission)

Other databases and relevant sources of known vulnerabilities and associated mitigation strategies include:

* Oracle’s Secure Coding Guidelines for Java SE
* CWE (Common Weakness Enumeration)
* CVE (Common Vulnerabilities and Exposures)
* Maven Repository

Following industry standard best practices for secure coding is crucial in order to protect software application security, as well as the overall well-being of the company at risk. Web applications are inherently vulnerable to DoS (Denial-of-Service), injection attacks, cross-site scripting (XSS), cross-site request forgery (CSRF), URL open redirects, unauthorized access, and other attacks. To prevent these attacks, a number of measures can be taken, such as the following:

* input validation/sanitization
* session management
* authentication and authorization (access control)
* privacy logging and proper error handling
* proper encryption and hashing algorithms
* secure communications
* proper key management
* PoLP (Principle of Least Privilege) / PoLA (Principle of Least Authority)
* API rate limiting
* secure APIs (Application Programming Interface)
* WAF (Web Application Firewall)
* parameterized queries
* adequate password creation and log-in attempt rules
* data masking & tokenization
* security headers such as CSP (Content Security Policy) and HSTS (HTTP Strict Transport Security)
* patch management (update libraries, frameworks, software, dependencies, etc.)

Using these preventative strategies in conjunction with others in a layered architecture provides the most robust security approach. This protects not only the software application and servers/data/users, but also the organization behind the app. An organization’s reputation can be damaged and never salvaged, leading to great financial loss or even complete loss of the business itself. This is why it is critical to make software security one of the top priorities, as well as following industry standards best practices when designing, implementing, and maintaining software systems of any kind.

Thank you,

Matthew Pool

Computer Science Department

Southern New Hampshire University (SNHU)

**FILE CONTENTS**: The refactored Java code and the corresponding project files have been included. Additional project resource files, such as application.properties, keystore.jks, and suppression.xml, are located in *./src/main/resources/*. The dependency-check-report.html file is located in the .*/target/* directory.