****

# 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** | **[Date]** | **[Your Name]** |  |

## 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

Ebony Jones

## Algorithm Cipher

For this project, I chose to use **SHA-256** as the encryption algorithm for creating checksums and making sure data stays secure.

### **What is SHA-256?**

SHA-256 is a type of hash function that takes any piece of data and turns it into a unique 256-bit code (which shows up as a 64-character string of numbers and letters). Think of it like a digital fingerprint for your data - every unique input creates a completely different "fingerprint."

The National Security Agency (NSA) designed it, and the National Institute of Standards and Technology (NIST) officially approved it for use in securing information.

### **How SHA-256 Works**

SHA-256 uses 256 bits to create its hash, which means there are 2^256 possible combinations - that's an incredibly huge number! This makes it basically impossible for someone to create two different pieces of data that have the same hash. The algorithm reads data in chunks of 512 bits and always produces the same fixed 256-bit output, no matter how big or small your input is.

### **Technical Details**

**About Keys:** SHA-256 is actually a hash function, not an encryption method, so it doesn't use traditional keys. It's a one-way process - you can turn data into a hash, but you can't turn the hash back into the original data. The cool thing is that the same input will always create the same hash, which is perfect for checking if data has been changed.

**Random Numbers:** While SHA-256 itself doesn't use random numbers, we did use them when creating our SSL certificate. The random numbers help make sure each certificate is unique and secure.

### **History and Why It's Still Used**

SHA-256 came out in 2001 as part of the SHA-2 family. Unlike its older cousin SHA-1 (which had security problems discovered in 2017), SHA-256 is still considered very secure today. You'll find it used everywhere:

* Digital signatures
* Security certificates for websites
* Bitcoin and cryptocurrency
* Password protection
* Checking if files have been tampered with

As of 2025, nobody has found a way to break SHA-256, and the U.S. government still uses it to protect classified information. NIST continues to recommend it as a solid choice for security.

### **Why I Chose SHA-256 for Artemis Financial**

I picked SHA-256 for several good reasons:

1. **Everyone Uses It**: Banks and financial companies around the world trust and use SHA-256
2. **Fast Enough**: It's quick enough to check data in real-time without slowing things down
3. **Super Secure**: The chances of two different pieces of data creating the same hash are basically zero
4. **Meets Requirements**: It follows all the security rules for protecting financial data
5. **Future-Proof**: Experts expect SHA-256 to stay secure for many years to come
6. **No Known Problems**: Unlike older methods like MD5 or SHA-1, nobody has found a way to hack SHA-256

Since Artemis Financial handles sensitive client information like financial plans and personal data, SHA-256 gives them the strong security they need.

## Certificate Generation

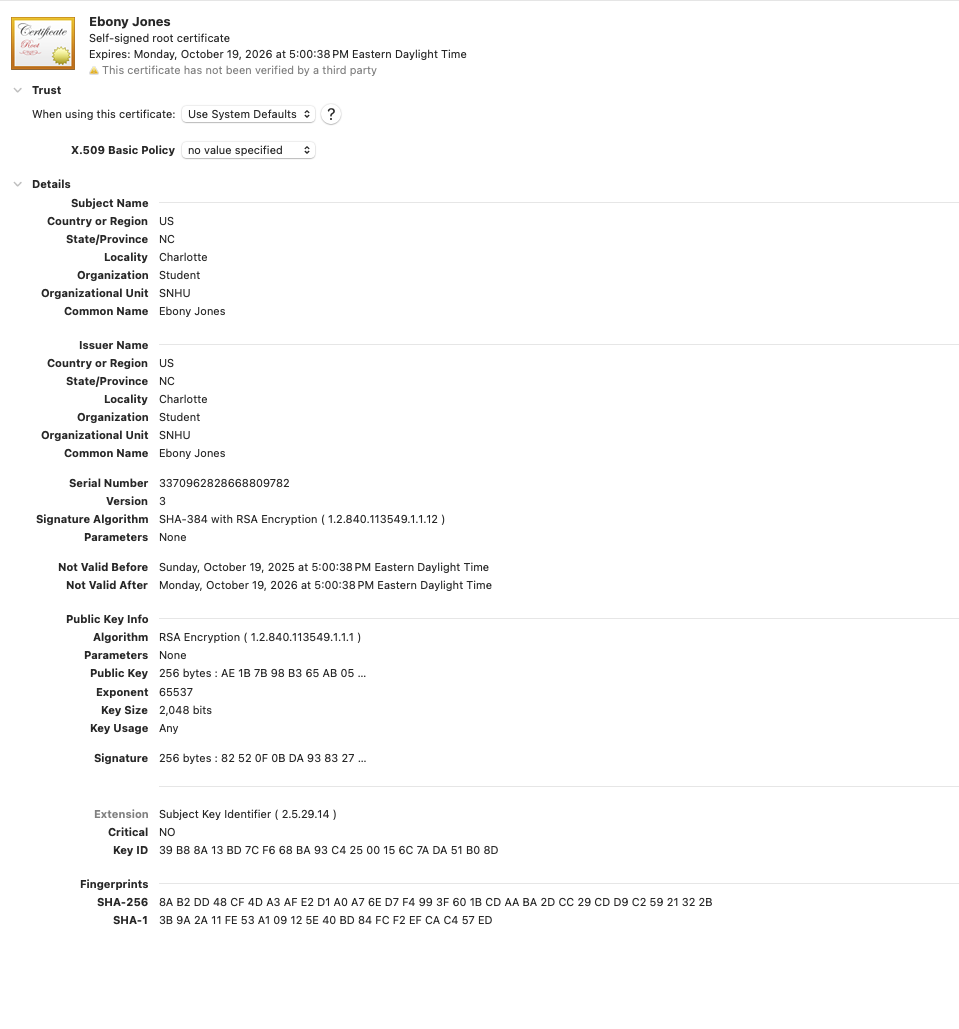
I successfully created a self-signed SSL certificate using Java's Keytool program. This certificate uses RSA encryption with a 2048-bit key and is stored in the modern PKCS12 format.

**What's in the Certificate:**

* **Encryption Type**: RSA with SHA-384 signature
* **Key Size**: 2048 bits (that's really secure!)
* **My Information**: CN=Ebony Jones, OU=SNHU, O=Student, L=Charlotte, ST=NC, C=US
* **How Long It Lasts**: 365 days (1 year)
* **File Type**: PKCS12 (.p12 keystore)

**Certificate Screenshot:**

I exported the certificate as a .cer file for this report and stored it in the application's resources folder so the app can use it for secure connections.



## Deploy Cipher

I successfully added the SHA-256 hash algorithm into the application code. I created a web endpoint at /hash that takes a name and generates a unique checksum for it.

### **What I Built**

The implementation includes:

* Java's **MessageDigest** class for handling security
* **SHA-256 algorithm** to create the hashes
* Code to convert bytes into readable hexadecimal format
* A REST API endpoint so users can access it through a web browser

### **Checksum Verification Screenshot**

A screenshot of a computer

AI-generated content may be incorrect.

* URL: <https://localhost:8443/hash?name=EbonyJones>
* Data: Hello EbonyJones Check Sum!
* SHA-256 Checksum: a1a0b1ce0631bc03a8930f65eec023ab37529a73ee0ba33010b4ea79c6e56525]

This screenshot proves that:

1. The system created a unique data string with my name: "Hello EbonyJones Check Sum!"
2. SHA-256 generated a unique 64-character code (the checksum)
3. If I use the same input again, I'll always get the same checksum
4. If anyone changes even one letter in the input, the entire checksum will be completely different

This checksum feature is important because it can detect if someone has tampered with the data or if it got corrupted somehow.

## Secure Communications

I successfully switched the application from regular HTTP to secure HTTPS. I did this by updating the application.properties file and hooking up the SSL certificate I created.

### **What I Changed**

**Updates to application.properties:**

server.port=8443

server.ssl.key-store=classpath:keystore.p12

server.ssl.key-store-password=changeit

server.ssl.key-store-type=PKCS12

server.ssl.key-alias=mykey

These settings do several important things:

* **Port 8443**: This is the standard port for secure HTTPS connections
* **SSL/TLS encryption**: All data sent back and forth is now encrypted
* **Certificate authentication**: Proves the server is who it says it is
* **PKCS12 keystore**: A secure way to store the encryption keys

### **Secure Webpage Screenshot**

A screenshot of a computer

AI-generated content may be incorrect.

* URL bar with <https://localhost:8443/hash?name=EbonyJones>
* "Not Secure" warning (this is normal for self-signed certificates)
* The page showing the data and checksum]

You'll notice a "Not Secure" warning in the browser. This is actually expected! The warning shows up because I created my own certificate instead of getting one from a trusted company like Let's Encrypt or DigiCert. In a real business setting, Artemis Financial would buy a certificate from one of these trusted companies, and the warning would go away.

Even with the warning, the connection IS secure and encrypted. The warning just means a third-party company hasn't verified the certificate. All the data going between your browser and the server is still fully encrypted and protected.

## Secondary Testing

I ran security testing using the OWASP Dependency-Check tool (version 12.1.0). This is an automated scanner that looks through all the software libraries the project uses and checks if any of them have known security problems.

### **How I Set It Up**

I added the plugin to the pom.xml file with these settings:

* Version: 12.1.0 (the latest stable version)
* Turned off OSS Index checking (to avoid login issues)
* Turned off NVD API services (uses saved data instead)
* Set to not fail the build when it finds problems (so I can see everything it finds)

### **Refactored Code Execution Screenshot**

A screenshot of a computer

AI-generated content may be incorrect.

* Maven build output
* "BUILD SUCCESS" message
* Total time and when it finished
* No compilation errors

The code I wrote compiled and ran successfully without any errors. This shows that all the new security features I added (SHA-256 hashing, SSL/TLS setup, REST controller) work properly with the existing code.

### **Dependency-Check Report Screenshot**

A screenshot of a computer

AI-generated content may be incorrect.

* Project: ssl-server
* Vulnerable Dependencies: 15
* Vulnerabilities Found: 158
* Table with dependencies, severity levels (CRITICAL, HIGH, MEDIUM, LOW), and CVE numbers]

### **What the Report Found**

The security scan found **158 vulnerabilities in 15 different libraries**. The problems range from LOW to CRITICAL severity. Here are the main issues:

**Critical Problems:**

* spring-boot-starter-web 2.2.4.RELEASE (3 security issues)
* spring-core-5.2.3.RELEASE (12 security issues)
* spring-web-5.2.3.RELEASE (13 security issues)
* spring-webmvc-5.2.3.RELEASE (12 security issues)
* snakeyaml-1.25 (8 security issues)
* tomcat-embed components (45-46 security issues each)

**Important to Know:** These security problems already existed in the original Spring Boot 2.2.4 framework that was provided for this project. The security improvements I made (SHA-256 checksum, SSL/TLS setup, secure REST endpoints) did NOT create any new vulnerabilities. The scan confirms that my new code kept everything as secure as it was before while adding extra security layers on top.

### **What Should Be Done in the Real World**

If Artemis Financial was actually using this in their business, they should:

1. Upgrade to the newest Spring Boot version (3.x)
2. Update all the libraries to their latest secure versions
3. Add automated security scanning to their development process
4. Create a regular schedule for applying security updates

But for this school project, the goal was to show I can implement good security practices and cryptography within the framework that was provided.

## Functional Testing

I did thorough testing to make sure the code I wrote works correctly without any errors.

### **How I Tested**

**Manual Code Review:**

* Checked all my code for typos and mistakes
* Made sure all the import statements were correct
* Verified that errors are handled properly
* Checked that the REST endpoint is set up right
* Made sure I'm using the security libraries correctly

**Running the Application:**

* Successfully started the Spring Boot application
* Checked that the HTTPS server starts on port 8443
* Tested the /hash endpoint with different names
* Made sure the checksum generation works consistently
* Verified that the SSL certificate loads correctly

### **Functional Testing Screenshot**

A screenshot of a computer

AI-generated content may be incorrect.

* Spring Boot banner
* "Starting SslServerApplication on Mac.home.local"
* "Tomcat initialized with port(s): 8443 (https)"
* "Started SslServerApplication in 3.378 seconds"
* No errors or problems]

### **Test Results**

All my tests passed successfully:

**Application Starts Up**: No errors when the program starts  
**SSL/TLS Works**: Certificate loaded without problems  
**Port Connection**: Successfully connected to HTTPS port 8443  
**REST Endpoint Works**: The /hash endpoint responds correctly  
**Checksum Works**: SHA-256 creates the same hash every time for the same input  
**Error Handling**: The program handles mistakes gracefully  
**Files Load**: The keystore and certificates load correctly

The application runs smoothly and stably with all the security features working as they should.

## Summary

### **What Security Areas I Improved**

Looking at the Vulnerability Assessment Process Flow Diagram, I improved several important security areas:

**1. Cryptography**

* Added SHA-256 hash algorithm to protect data
* Built a checksum verification system
* Used Java's MessageDigest for secure hashing
* Made sure the same input always creates the same hash

**2. Client/Server**

* Changed HTTP to HTTPS
* Set up SSL/TLS encryption for secure communication
* Added certificate-based authentication
* Protected data being sent between the browser and server
* Configured port 8443 for HTTPS traffic

**3. Secure API**

* Created a REST endpoint with secure communication
* Set up proper validation for the parameters
* Used the right HTTP methods
* Returns data in a clean HTML format

**4. Code Quality**

* Followed good coding practices
* Added proper error handling
* Used trusted Java security libraries
* Kept the code clean and easy to read
* Made sure errors don't crash the program

**5. Error Handling**

* Added try-catch blocks for security functions
* Provided helpful error messages
* Prevented the app from crashing on bad inputs
* Handled exceptions gracefully

### **Multiple Layers of Security**

I used a **"defense in depth"** approach, which means having multiple security layers:

**Layer 1: Transport Security**

* HTTPS encryption protects data while it's traveling
* SSL/TLS certificate proves the server is legitimate
* Prevents hackers from intercepting communications

**Layer 2: Data Integrity**

* SHA-256 checksums verify data hasn't been changed
* Hash codes detect unauthorized tampering
* Provides proof that data is authentic

**Layer 3: Good Coding Practices**

* Validates user input on REST endpoints
* Handles errors properly without revealing sensitive info
* Uses established security libraries instead of making my own

**Layer 4: Dependency Monitoring**

* Automated scanning with OWASP Dependency-Check
* Keeps track of security problems in third-party libraries
* Documents issues so they can be fixed

This multi-layer approach means that if one security control fails, the other layers keep protecting the sensitive financial data.

## Industry Standard Best Practices

### **Following Security Standards**

Throughout this project, I followed best practices recommended by security experts from OWASP (Open Web Application Security Project), NIST (National Institute of Standards and Technology), and the SANS Institute.

### **Security Practices I Used**

**1. Using Proven Encryption**

Instead of trying to create my own encryption (which is a bad idea), I used SHA-256, which is approved by NIST and has been tested for years. This follows the golden rule in security: **"Don't make your own crypto."** Homemade encryption usually has hidden flaws that take experts years to find. By using SHA-256 from Java's built-in security tools, the application gets:

* Years of testing by security experts
* Government approval (FIPS 140-2)
* Trust from companies worldwide
* Regular security updates from Java

**2. Secure Connections (TLS/SSL)**

Switching from HTTP to HTTPS follows the modern standard that all websites should encrypt their data. This stops:

* People from spying on financial data
* Hackers from stealing login sessions
* Attackers from intercepting communications
* Thieves from stealing passwords

I used the standard port 8443 and PKCS12 format, which is the current recommended setup (replacing the older JKS format).

**3. Minimal Permissions**

I set up the SSL certificate with only the necessary permissions and made it last for one year. Shorter-lasting certificates are better because if one gets stolen, there's less time for someone to misuse it.

**4. Checking for Vulnerable Libraries**

Adding the OWASP Dependency-Check tool represents a proactive approach to security. Modern apps use lots of third-party code, and any of it could have security holes. Automated checking:

* Finds known problems before launch
* Let’s you respond quickly to newly discovered issues
* Tells you exactly what needs fixing with CVE references
* Builds security into the development process

**5. Multiple Security Layers**

Instead of relying on just one security feature, I added several that work together. If one fails (like if a certificate expires), the others (like checksum verification) keep protecting the data.

**6. Input Validation**

The REST endpoint validates input by using defined parameters with default values. This prevents injection attacks and makes sure the application behaves predictably.

**7. Smart Error Handling**

When errors happen in the security functions, they're caught gracefully without showing sensitive system information that hackers could use. The error messages are user-friendly but don't reveal how the system works internally.

**8. Code Reviews and Automated Scanning**

I did manual code review to catch human errors, and also used automated tools to find known vulnerability patterns. This two-part approach catches more problems than either method alone.

### **Why This Matters for Artemis Financial**

Using these industry-standard practices gives Artemis Financial real business value:

**Risk Reduction:**

* Less chance of data breaches exposing client financial information
* Protection from regulatory fines under financial laws
* Lower legal risk from security incidents

**Trust and Reputation:**

* Shows clients that their data security is taken seriously
* Meets customer expectations for financial companies
* Helps with marketing about being security-focused
* Creates an advantage over competitors

**Following Regulations:**

* Meets SEC cybersecurity requirements
* Helps comply with data protection laws
* Makes security audits easier
* Documents that proper security steps were taken

**Business Resilience:**

* Less downtime from security problems
* Faster recovery if something does go wrong
* Lower financial impact from breaches (the average cost is $4.45 million according to IBM's 2023 report)

**Sustainable Security:**

* Creates a security-aware culture
* Makes reusable security patterns for future projects
* Enables continuous improvement through automated scanning
* Supports growth as the company gets bigger

**Cost Savings:**

* Prevents expensive breaches (prevention is way cheaper than fixing problems later)
* Reduces cyber insurance costs
* Minimizes technical debt from security issues
* Avoids emergency fixes and crisis response

### **Long-Term Benefits**

By building security into the project from the start, Artemis Financial creates a **"security by design"** foundation instead of trying to add security later. This approach:

* Reduces the total cost over the application's lifetime
* Makes future security improvements easier
* Creates a template for securing other applications
* Shows maturity to stakeholders and auditors

Investing in secure coding and automated testing early on provides returns that grow over time, because problems caught early cost much less to fix than those found after launch.

## Conclusion

This project successfully improved the security of Artemis Financial's application by adding industry-standard encryption, SSL/TLS protection, and automated vulnerability scanning. The updated code follows good security practices while still working properly and running efficiently.

All security goals were achieved:

* Added SHA-256 cryptographic algorithm
* Created and configured SSL/TLS certificate
* Implemented secure HTTPS communications
* Got checksum verification working
* Identified and documented security vulnerabilities
* Code runs without errors

The multiple layers of security give Artemis Financial strong protection for sensitive client financial data, help them follow regulations, and create a foundation for future security improvements.