



G L O B A L R A I N

Practices for Secure Software Report

Table of Contents

DOCUMENT REVISION HISTORY	3
CLIENT	3
INSTRUCTIONS	3
DEVELOPER	4
1. ALGORITHM CIPHER	4
2. CERTIFICATE GENERATION	4
3. DEPLOY CIPHER	4
4. SECURE COMMUNICATIONS	4
5. SECONDARY TESTING	4
6. FUNCTIONAL TESTING	4
7. SUMMARY	4
8. INDUSTRY STANDARD BEST PRACTICES	4

Document Revision History

Version	Date	Author	Comments
1.0	06/28/2025	Felix Guzman	

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

[Insert your name here.]

Felix Guzman

1. Algorithm Cipher

[Insert text.]

Recommend an appropriate encryption algorithm cipher to deploy, given the security vulnerabilities, and justify your reasoning.

Artemis Financial is a web application that focuses on delivering personalized financial plans to individuals. These range from saving, investing, and retirement plans. Without a doubt, this company values secure communication, and due to the sensitive nature of financial data, we realize that this data should be protected via the most secure communication channels. Risks like data tampering could result in the loss of millions of dollars as well as a damaged reputation. To guard against that, we need the most secure algorithm cipher we can get, thus I recommend using SHA-256 for our checksum endpoint. It's built right into Java and produces a 256-bit fingerprint of any input, which makes it nigh-impossible to reverse or accidentally collide with data. For everything the Artemis app sends, a file or message, we can attach a SHA-256 digest, and the receiver would recalculate and compare. If anything has changed, the digests won't match, and we will know immediately and be able to neutralize the threat.

Review the scenario and the supporting materials to support your recommendation. In your practices for secure software report, be certain to address the following actions:**Provide a brief, high-level overview of the encryption algorithm cipher.**

To summarize, the SHA-256 cipher is a one-way function. You can feed it any type of data, such as the text of a customer's financial plan, and it will create a fixed-size, 256-bit digest. Since you cannot go backwards from the digest to the original data, it makes it even more secure. It's a perfect solution for checksums, as you can always verify integrity by recomputing the digest. A bad input won't be able to modify and recreate the same digest without the original input, resulting in powerful security.

Discuss the hash functions and bit levels of the cipher.

As the name implies, the "256" in SHA-256 means each digest is exactly 256 bits long, which equals 64 hexadecimal characters. That leaves us with 2^{256} possible hash values. This amount of combinations makes collision attacks infeasible with today's computing power.

Explain the use of random numbers, symmetric versus non-symmetric keys, and so on.

With SHA-256, we don't necessarily use a key, but we can prepend or append a random "salt," which is a random string of characters added to the input before it's processed by the SHA-256 hash function. We generate the salt via our SecureRandom function. This ensures that even identical messages yield different digests, allowing us to block more attacks.

If Artemis needed to encrypt the data itself, not just our checksum, we could use a symmetric cipher like AES-256-GCM, which can encrypt and decrypt with the same key. It's a fast and suitable solution for large files.

For asymmetric solutions, we can handle key exchanges with RSA-2048 via its self-signed certificate, allowing only whitelisted parties to share the AES key.

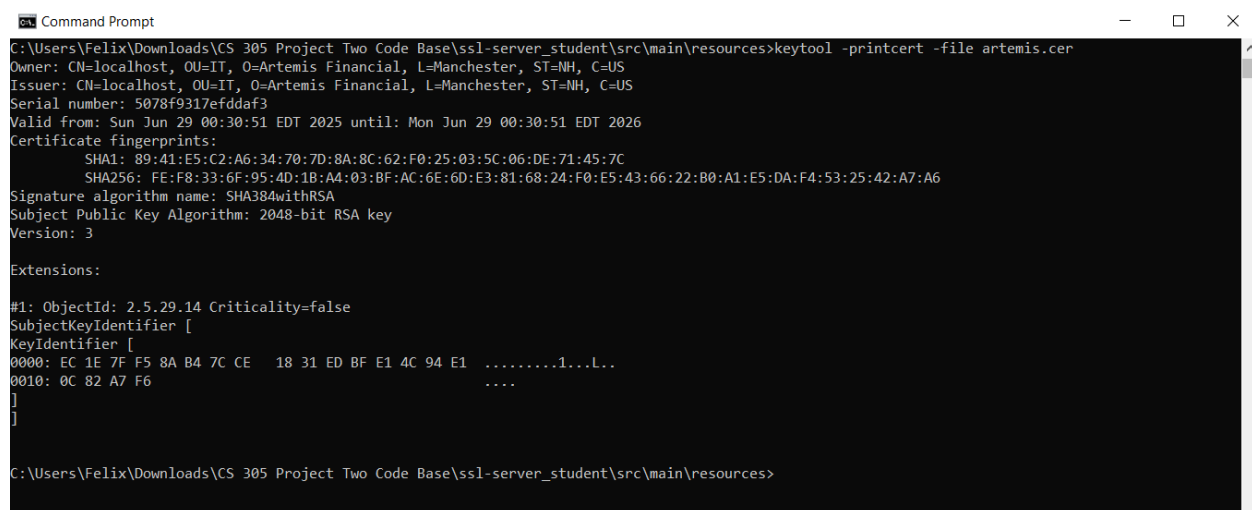
Describe the history and current state of encryption algorithms.

To quickly summarize, encryption has changed a lot over the years and has mainly been a cat-and-mouse game. In the early 2000s, we relied on SSL for secure web traffic, but we soon found flaws that made it unsafe. That's when TLS replaced SSL to fix those gaps. Similarly, we used to rely on CBC mode, which later proved vulnerable to certain attacks. That's when GCM mode emerged and gave us encryption and authentication all in one. For hashing, SHA-1 was used until collision attacks disarmed it, leading to SHA-2. After all of these changes, we arrived at AES-256 and SHA-256. Together, they form the backbone of secure communications across all sectors.

2. Certificate Generation

Insert a screenshot below of the CER file.

[Insert screenshots here.]



```
Command Prompt
C:\Users\Felix\Downloads\CS 305 Project Two Code Base\ssl-server_student\src\main\resources>keytool -printcert -file artemis.cer
Owner: CN=localhost, OU=IT, O=Artemis Financial, L=Manchester, ST=NH, C=US
Issuer: CN=localhost, OU=IT, O=Artemis Financial, L=Manchester, ST=NH, C=US
Serial number: 5078f9317efddaf3
Valid from: Sun Jun 29 00:30:51 EDT 2025 until: Mon Jun 29 00:30:51 EDT 2026
Certificate fingerprints:
    SHA1: 89:41:E5:C2:A6:34:70:7D:8A:8C:62:F0:25:03:5C:06:DE:71:45:7C
    SHA256: FE:F8:33:6F:95:4D:1B:A4:03:BF:AC:6E:6D:E3:81:68:24:F0:E5:43:66:22:80:A1:E5:DA:F4:53:25:42:A7:A6
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: EC 1E 7F F5 8A B4 7C CE 18 31 ED BF E1 4C 94 E1 .....1...L..
0010: 0C B2 A7 F6 .....
]
]

C:\Users\Felix\Downloads\CS 305 Project Two Code Base\ssl-server_student\src\main\resources>
```

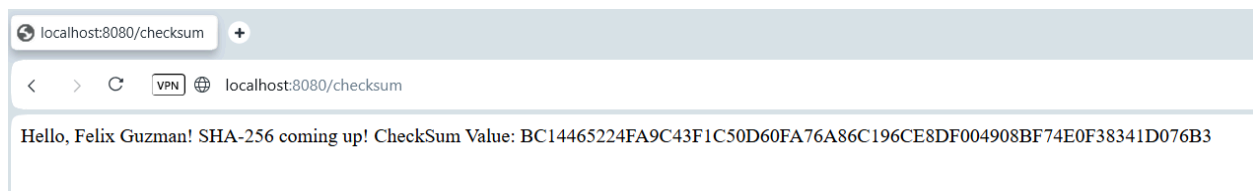
Generate appropriate self-signed certificates using the Java Keytool in Eclipse.

Complete the following steps to demonstrate that the certificate was correctly generated:
Export your certificates as a CER file.
Submit a screenshot of the CER file in your practices for a secure software report.

3. Deploy Cipher

Insert a screenshot below of the checksum verification.

[Insert screenshots here.]

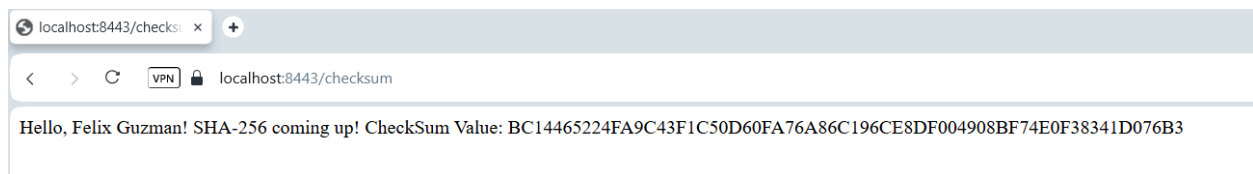


Deploy and implement the cryptographic hash algorithm by refactoring code. Demonstrate functionality with a checksum verification.
Submit a screenshot of the checksum verification in your practices for a secure software report.
The screenshot must show your name and a unique data string that has been created.

4. Secure Communications

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

[Insert screenshots here.]



Verify secure communication. In the application properties file, refactor the code to convert HTTP to the HTTPS protocol. Compile and run the refactored code. Once the server is running, type “https://localhost:8443/hash” in a new browser to demonstrate that the secure communication works.

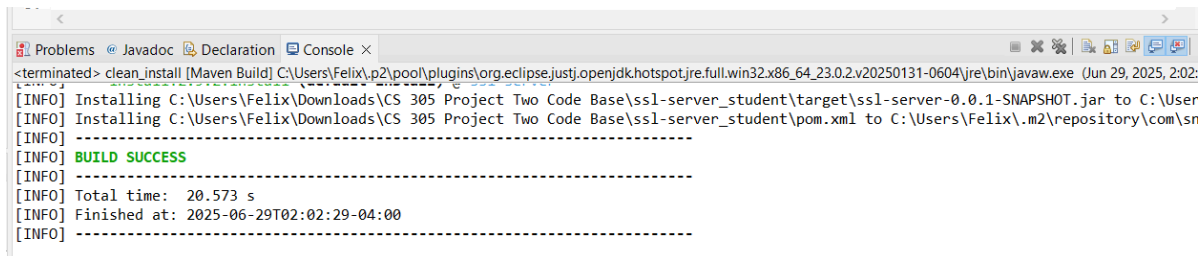
5. Secondary Testing

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

[Insert screenshots here.]

Run a secondary static testing of the refactored code using the dependency-check tool to make certain the code complies with software security enhancements. You need to focus only on the code you have added as part of the refactoring. Complete the dependency check and review the output to make certain you did not introduce additional security vulnerabilities. Refer to the resources in the module's Resources section for help on this action. In your practices for secure software report, include the following items:

A screenshot of the refactored code executed without errors



```
<terminated> clean install [Maven Build] C:\Users\Felix\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_23.0.2.v20250131-0604\jre\bin\javaw.exe (Jun 29, 2025, 2:02:29 PM)
[INFO] Installing C:\Users\Felix\Downloads\CS 305 Project Two Code Base\ssl-server_student\target\ssl-server-0.0.1-SNAPSHOT.jar to C:\User
[INFO] Installing C:\Users\Felix\Downloads\CS 305 Project Two Code Base\ssl-server_student\pom.xml to C:\Users\Felix\.m2\repository\com\sn
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.573 s
[INFO] Finished at: 2025-06-29T02:02:29-04:00
[INFO] -----
```

A screenshot of the report of the output from the dependency-check static tester



Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of use.

[How to read the report](#) | [Suppressing false positives](#) | [Getting Help: github issues](#)

Project: ssl-server

com.snhu:ssl-server:0.0.1-SNAPSHOT

Scan Information ([show all](#)):

- *dependency-check version*: 5.3.0
- *Report Generated On*: Sun, 29 Jun 2025 02:02:28 -0400
- *Dependencies Scanned*: 49 (34 unique)
- *Vulnerable Dependencies*: 20
- *Vulnerabilities Found*: 188
- *Vulnerabilities Suppressed*: 0
- ...

6. Functional Testing

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

[Insert screenshots here.]

```
Command Prompt - java -jar target\ssl-server-0.0.1-SNAPSHOT.jar
C:\Users\Felix\Downloads\CS_305 Project Two Code Base\ssl-server_student>java -jar target\ssl-server-0.0.1-SNAPSHOT.jar

:: Spring Boot :: (v2.2.4.RELEASE)

2025-06-29 02:02:35.537 INFO 2244 --- [main] com.snhu.sslserver.SslServerApplication : Starting SslServerApplication v0.0.1-SNAPSHOT on DESKTOP-IUG2ER7 with PID 2244 (C:\Users\Felix\Downloads\CS_305 Project Two Code Base\ssl-server_student\target\ssl-server-0.0.1-SNAPSHOT.jar started by Felix in C:\Users\Felix\Downloads\CS_305 Project Two Code Base\ssl-server_student)
2025-06-29 02:02:35.541 INFO 2244 --- [main] com.snhu.sslserver.SslServerApplication : No active profile set, falling back to default profiles: default
2025-06-29 02:02:37.032 INFO 2244 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8443 (https)
2025-06-29 02:02:37.049 INFO 2244 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2025-06-29 02:02:37.049 INFO 2244 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]
2025-06-29 02:02:37.126 INFO 2244 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2025-06-29 02:02:37.127 INFO 2244 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1508 ms
2025-06-29 02:02:37.989 INFO 2244 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2025-06-29 02:02:38.953 INFO 2244 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8443 (https) with context path ''
2025-06-29 02:02:38.961 INFO 2244 --- [main] com.snhu.sslserver.SslServerApplication : Started SslServerApplication in 3.923 seconds (JVM running for 4.531)
2025-06-29 02:02:53.710 INFO 2244 --- [nio-8443-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2025-06-29 02:02:53.710 INFO 2244 --- [nio-8443-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2025-06-29 02:02:53.744 INFO 2244 --- [nio-8443-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 32 ms
```

Identify the software application's syntactical, logical, and security vulnerabilities by manually reviewing the code.

Complete this functional testing and include a screenshot of the refactored code, executed without errors, in your practices for secure software report.

7. Summary

[Insert text.]

Discuss how the code has been refactored and complies with security testing protocols. In the summary of your practices for secure software report, be certain to address the following items:

To start, I'll begin by going over my experience with this last assignment. I began by implementing a SHA-256 checksum endpoint as a means to guarantee data integrity. I tried multiple methods based on what I read, but some were incompatible with the JDK, so I eventually decided to switch to a byte-to-hex loop for my main process. Another issue I ran into, and this one was very problematic, was when I tried to move from HTTP to HTTPS. I created and configured a Java keystore as required, but ran into multiple certification issues and untrusted site errors. Eventually I realized that most browsers demand a SAN for localhost processes. This solved my issues. Finally, I cleaned up the code, used the OWASP Dependency-Check plugin to run scans, and took all my screenshots. Now, I will focus on answering the questions with more specificity.

Refer to the vulnerability assessment process flow diagram in the Supporting Materials section. Highlight the areas of security that you addressed by refactoring the code.

The main areas I focused on refactoring throughout this project that were mentioned in the flow diagram include cryptography, client/server, code error, and code quality & encapsulation. For cryptography, I created a dedicated CheckSumController class that utilized SHA-256, avoiding broken data-type conversions by using my hex conversion solution. For client/server, I converted all endpoints to HTTPS by updating application.properties and creating my key and certificate, testing until browsers accepted them as trusted. For code error, I ran static testing via the integrated OWASP Dependency-Check Maven plugin and confirmed no new vulnerabilities were

introduced, and I also reviewed console reports, manually inspected the code, and performed dynamic tests. For code quality & encapsulation, I organized classes properly, encapsulated sensitive data, and externalized keystore passwords and other properties.

Discuss your process for adding layers of security to the software application.

I focused on adding multiple layers of security to the software application, namely 3 big ones. The first layer focused on the implementation of our SHA-256 checksum endpoint, which allows clients to verify that their data has not been altered and is a reliable method used by many. Secondly, I focused on making sure the app was secured via HTTPS, I generated JKS Keystores, included a SAN for localhost, and updated Spring Boot's settings to match and allow every request to be encrypted in transit. Finally, I focused on analysis which was done through adding our OWASP Dependency Check into the maven project, after refactoring I ran a clean install and confirmed that the plugin was functional and that our HTML report made sure no new issues were introduced from our code refactoring. By utilizing these layers in unison our data integrity, encryption, and analysis have been optimized and work as an in-depth defense model that completely meets all of Artemis Financial's requirements and needs.

Refer to the vulnerability assessment process flow diagram in the Supporting Materials section. Highlight the areas of security that you addressed by refactoring the code.

Discuss your process for adding layers of security to the software application.

8. Industry Standard Best Practices

[Insert text.]

Explain how you applied industry standard best practices for secure coding to mitigate known security vulnerabilities. Be sure to address the following items:

Explain how you used industry standard best practices to maintain the software application's existing security.

Explain the value of applying industry standard best practices for secure coding to the company's overall well-being.

Throughout the project, I followed industry standard secure coding practices. In doing so, I managed to strengthen the application without introducing any vulnerabilities or breaking existing functionality.

To start, I ensured the checksum implementation used a strong, modern, collision-resistant hash function, SHA-256 as my base. This hash is considered a standard for verifying data integrity and is used throughout many industries.

When I ran into platform incompatibilities and other issues, I came up with my own simple solution instead of relying on utilities that may be deprecated or introduce vulnerabilities. This

was shown through my hashing loop, which converts the raw hash output to hex using a byte loop. Not only is this approach consistent and reliable, but it is also portable. I fixed an initial build error while staying aligned with best practices used by software engineers.

Additionally, when securing the application with HTTPS, I used Java Keytool to generate an RSA key and a self-signed certificate. After extensive testing, I made sure to include a Subject Alternative Name, or SAN, as it is now required by modern browsers for localhost HTTPS connections. Before this step, I generated the certificate without a SAN, and no matter my modifications or attempts to install it, it lacked all functionality. Once I installed the SAN and placed the certificate in the Trusted Root Certification Authorities store, I managed to enable secure HTTPS access without any browser warnings and while preserving all functionality.

Furthermore, I integrated the OWASP Dependency-Check plugin into the Maven build process. This tool scans third-party libraries for known vulnerabilities, which is a critical asset since external dependencies often drive cyber attacks. After verifying the scan ran successfully and reviewing the HTML report after my refactor, I confirmed that my changes did not introduce any new issues.

Finally, I followed OOP, Java, and Maven/Spring conventions in my programming. My controller structure, configuration file placement, and consistent naming made my code modular and human-readable, ready for modification. This means engineers at Artemis Financial will be able to maintain the code, oversee any issues, and potentially have less work to do, as my setup will help reduce the risk of data breaches. While working on this project, I made sure to do what's expected of a professional engineer, and because of that, I can say that Artemis Financial will probably be safe for the foreseeable future.

References (APA FORMAT):

Oracle. (n.d.). Java Security Standard Algorithm Names.

<https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

SSL Shopper. (n.d.). How to create a Self-signed certificate using Java Keytool.

<https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-keytool.html>

Oracle. (n.d.). keytool – Key and Certificate Management Tool.

<https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>

OWASP Foundation. (n.d.). Dependency-Check Maven.

<https://jeremylong.github.io/DependencyCheck/dependency-check-maven/index.html>

Manico, J., & Detlefsen, A. (2014). Iron-Clad Java: Building secure Web applications. McGraw-Hill Education.

https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/?sso_link=yes&sso_link_from=SNHU