****

# 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** | **10/19/2025** | **John Swindell** | **Project 2 First Revision** |

## Client



## Developer

John Swindell

## Algorithm Cipher

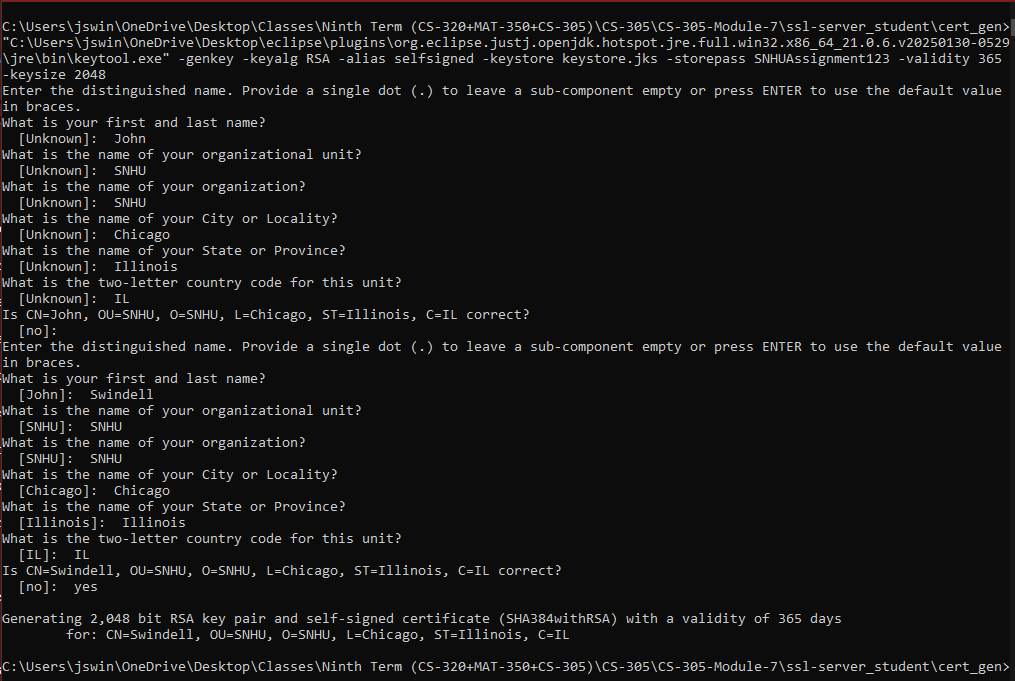
My recommendation for the algorithm cipher is the AES algorithm. The AES algorithm is what will encrypt data in fixed sized blocks, and a mode of operation is how that is applied to data larger than a single block. To be more specific, I would recommend its use with a 256 bit key, as it is the most secure and recommended for the best long term data protection. Additionally, the recommendation of a 256 bit key will provide the highest security and ensure the data will remain secure despite the advances in computing power on the horizon. Hash functions and modes of operations are another essential aspect to consider when coming up with a cipher suite, not just an algorithm to encrypt with. Put simply, the chosen mode of GCM is the blueprint for how this operation is handled, and is the superior choice to another mode like CBC for a number of reasons, but most importantly, the hash function is built in. This automatically creates an authentication tag then data blocks are stored, which will ensure data integrity, the other essential half of securing data at rest.

Secure random numbers are essential to ensuring the system is actually strong. For instance, if the encryption keys are predictable, it doesn’t matter how long they are, or how well you store them, someone can simply predict which seed of random number was generated for your encryption key. This essential practice uses a cryptographically secure pseudo random number generator to make genuinely unpredictable random numbers, as computers are generally deterministic, and simply using a randomly generated integer without ensuring it is cryptographically secure is a vulnerability in and of itself. To prevent this, using a library such as SecureRandom will ensure they cannot be predicted by an attacker. This solution intentionally uses a symmetric key, as it is highly efficient for securing data at rest. Symmetric essentially refers to the key used to encrypt and decrypt the data being the exact same. In this case, it is ideal because the person or entity that needs to encrypt the data, will be the same individual that must later decrypt it. Asymmetric keys largely serve a different purpose, catering to data in transit specifically. For instance, they are often used for establishing secure communications channels and digital signatures, not for the bulk encryption of static files (Manico & Detlefsen, 2014).

The history of encryption algorithms is extensive, essentially ever since we needed to storing things securely, we needed a way to make them “secure”. The first encryption standards like DES from the 70s are now considered “broken”, not necessarily because they won’t work, but because literally brute forcing the keys has become feasible with modern computational power. For instance, “It has already been proven that DES doesn't protect well against a brute force attack,” said David McNett, one of distributed.net’s primary coordinators, “but what this effort shows is that data encrypted with DES is safe for less than a day” (McNett, 1999). That was from 1999, and now the key is completely insecure for most use cases. The selected technology of AES, however, is the new standard for good reason. It is still, from back in 2001, the state of the art symmetric cipher that is trusted around the world for protecting sensitive data. It has now been tested extensively for decades, and while there may be a future in which the computational power required to crack this cipher is negligible, it is not going to be any time soon. As for best practices and the best cipher, the Advances Encryption Standard (AES) is listed as the government standard “by NIST in FIPS 197” (Oracle, 2017).

After reviewing the current cryptographic best practices and the Java Security Standard Algorithm Names given by Oracle (2017), I recommend the specific cipher suite: AES/GCM.NoPadding. Since it is FIPS certified, and has no known practical weaknesses, this is my recommendation. The Galois/Counter Mode (GCM) is the mode of operation I chose, as it provides Authenticated Encryption, something older and less modern modes do not (Oracle, 2017). The use of GCM here is best practice purely because it keeps confidentiality, data integrity, and authenticity bundled. This means we do not need a separate hashing algorithm to ensure integrity, as GCM will handle this for us, and keep the files from being tampered with secretly. In this case, it is difficult to make an argument for Artemis to not choose the strongest cipher suite, however, viable alternatives are AES 192 or 128. The main reason you’d opt for a lower bit level would be for performance. Encrypting and decrypting with a 256 bit key is substantially more intensive on your computer than with a key half the size. However, for these long term archives, it’s unlikely that data will be accessed often enough for this computational overhead to really make a difference. When the impact is negligible, and the security is superior, the choice to make is in obvious favor of the 256 bit key.

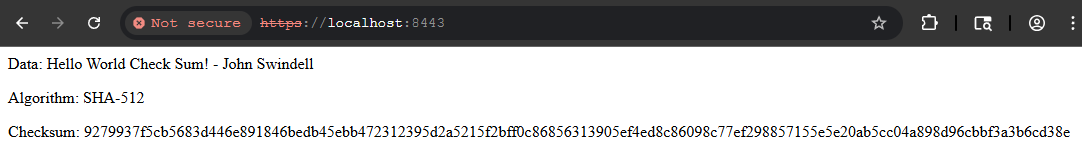
## Certificate Generation



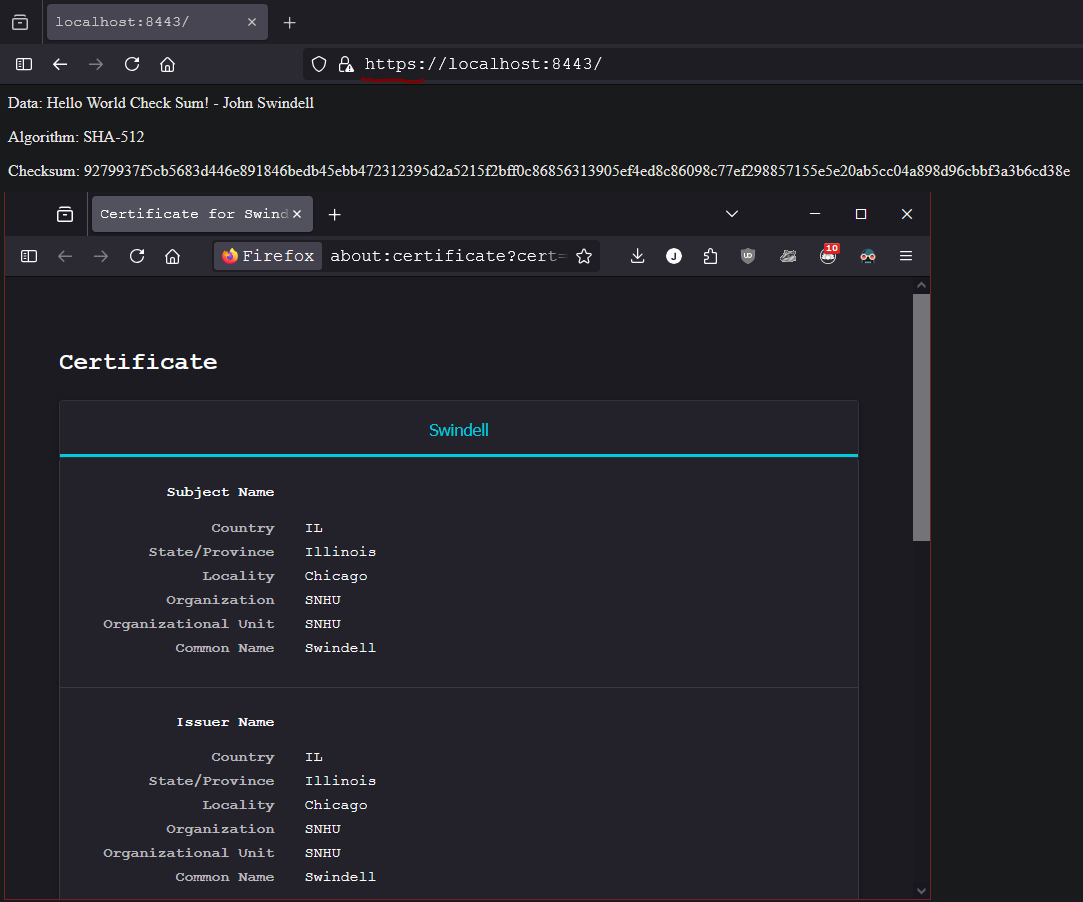
A screenshot of a computer program

AI-generated content may be incorrect.

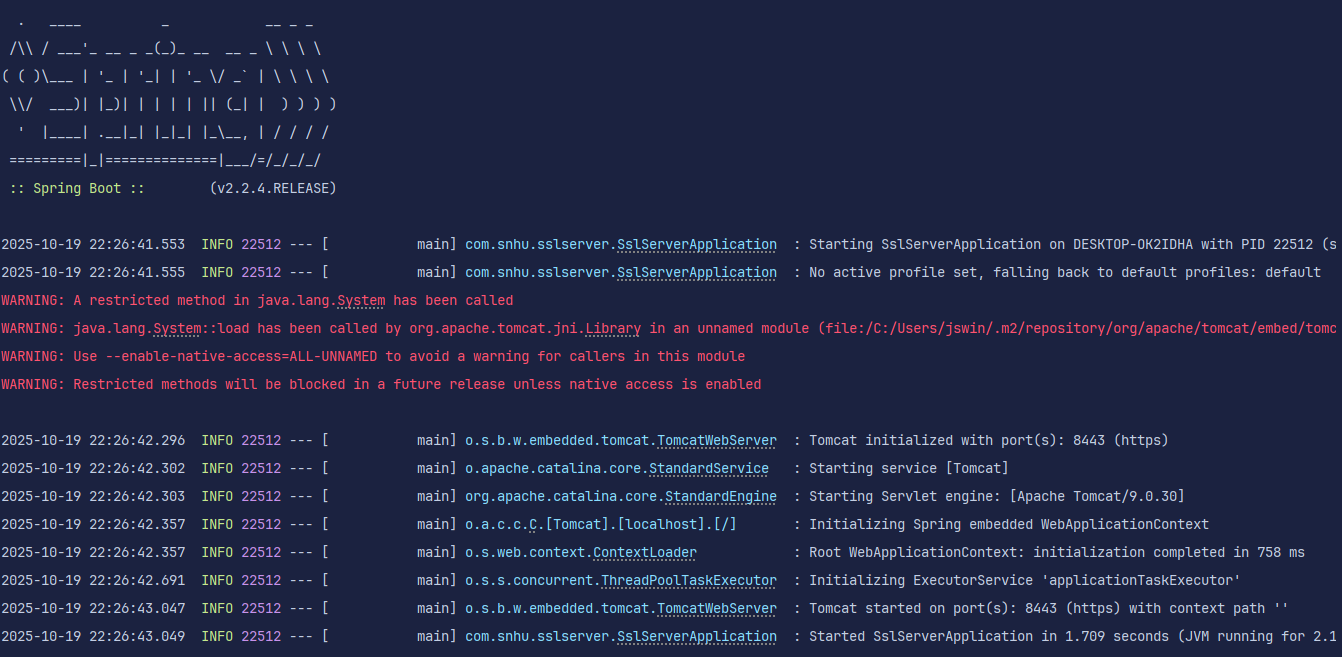
## Deploy Cipher

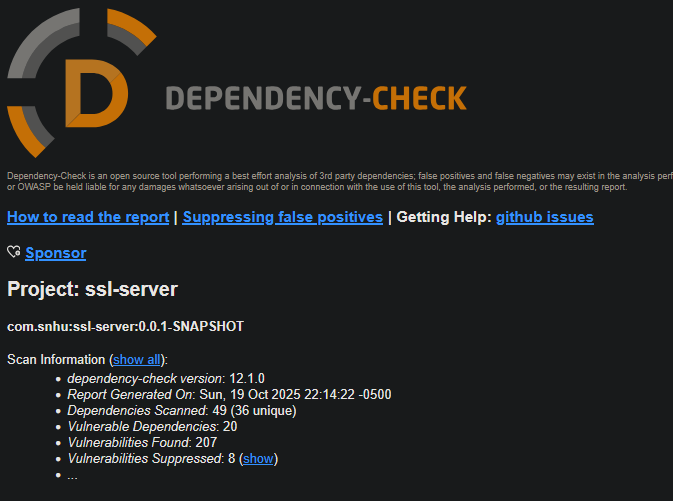


## Secure Communications

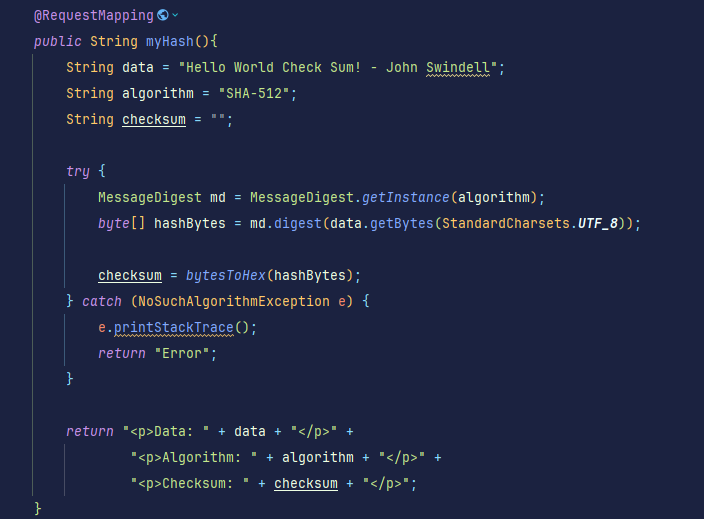


## Secondary Testing





## Functional Testing





## Summary

I ran the initial OWASP dependency check to see which vulnerabilities were in the code base prior to any refactoring. The goal of the dependency check was not necessarily to identify and cover all vulnerabilities, but rather to have a baseline for when the code has been altered to ensure no new vulnerabilities are created. After running the initial dependency check and taking count of any known vulnerabilities in the code base, I began refactoring the code. I implemented two new security controls, one of which being a hashing function for data integrity which served to /hash, and another being HTTPS for secure communication. After I ensured the code compiled correctly, I ran the dependency check once more to ensure I did not introduce any new vulnerabilities into the code base. Additionally, I added a suppression.xml, and used a Sonatype API key to ensure the entire code base was scanned effectively. After confirming no new issues were introduced, verifying the hashing was working correctly, and ensuring the HTTPS certificate was valid, the refactoring was complete. The security layers created are as such:

* Layer 1: I added a checksum hashing function which is for data integrity. Essentially, this allows the client to ensure the files or data received are not tampered with in transit.
* Layer 2: I generated a server file and keystore so that TLS/HTTPS could be enabled in my localhost port. Once the server is configured to use HTTPS, all communication between the client and server is protected from any eavesdropping, which is data confidentiality, and malicious acts like man in the middle attacks.
* Layer 3: I implemented a static analysis tool, OWASP dependency check, with the goal of finding any known vulnerabilities in the code base. I also created a suppression.xml to manage known false positives, aiming to effectively manage the risk and beginning the triage process.

## Industry Standard Best Practices

I believe SHA-3 is the strongest choice available from the listed Oracle hashing algorithms, thus, it was my choice to use it here. It is the newest hashing standard from NIST, and is specifically designed to be highly resistant to collision attacks. The 512 bit length was chosen due to it’s nature of being substantially more difficult to exploit. It refers to the length of the hash’s “fingerprint”, and the longer the hash is, the more difficult it is to crack or collide with. This is the most secure option available from the official Oracle documentation, and it is perfect for verifying the integrity of an important file like a public key.

Additionally, I implemented HTTPS here, which is utilizing the SSL/TLS protocols and is industry standard for something needing tight security like Artemis Financial. All web traffic will be protected once connected to the server. The CA generated is currently self signed and active for 365 days, but Artemis will need to have their CA authenticated by an authority when the actual server is live. For now, the proof of concept is working perfectly, as evidenced by the screenshots above.

I used the OWASP dependency check tool and Maven for dependency management, which is also an industry standard practice for managing supply chain risk. In addition to using this tool to perform static testing and identify vulnerabilities, I suppressed snake.yaml. This dependency is not actually used in the code base, and was throwing false positives in the final report. However, all other dependencies should be updated or addressed.

A company like Artemis financial must follow these practices in order to adhere to many different standards. The first of which is the standard with the consumer, as their data and safety is the top priority. For instance, if a customer tries to login to their financial accounts, and is hit with a warning telling them the site isn’t secure, customer trust is instantly destroyed. For a financial institution like Artemis, using HTTPS with a real CA is an absolute must. Additionally, you must comply with laws and regulations regarding PII and financial information. Things like GDRP for the UK and GLBA for the US legally require you to protect customer data and prevent it from being shared. For cost effectiveness, simply managing dependencies and implementing secure coding practices is paramount. It is much easier to perform static testing, like a dependency check, and ensure the code base has been thoroughly reviewed, than it is to recover after a massive data breach. The reputational damage alone could put many companies out of business entirely. Thus, testing early, often, and ensuring and vulnerability is handled during development is the most important cost management tool available.

**References**

Manico, J., & Detlefsen, A. (2014). Iron-clad Java. McGraw Hill. https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/

Oracle. (2017). Java Security Standard Algorithm Names. https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names

David C. McNett (1999). "US Government's Encryption Standard Broken in Less Than a Day" (PDF). https://www.distributed.net/images/d/d7/19990119\_-\_PR\_-\_release-des3.pdf