****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc137818107)

[Document Revision History 3](#_Toc137818108)

[Client 3](#_Toc137818109)

[Instructions 3](#_Toc137818110)

[Developer 4](#_Toc137818111)

[1. Algorithm Cipher 4](#_Toc137818112)

[2. Certificate Generation 6](#_Toc137818113)

[3. Deploy Cipher 8](#_Toc137818114)

[4. Secure Communications 8](#_Toc137818115)

[5. Secondary Testing 10](#_Toc137818116)

[6. Functional Testing 12](#_Toc137818117)

[7. Summary 16](#_Toc137818118)

[8. Industry Standard Best Practices 17](#_Toc137818119)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **5/18/23** | **Chris Trimmer** | **Document creation and initial submission** |
| **2.0** | **06/16/23** | **Chris Trimmer** | **Revision with updated data, additional sections, and final recommendations** |

## 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

Christopher Trimmer

## Algorithm Cipher

To begin the analysis of ciphers we should briefly describe the “CIA triad”, as well as the overall purpose of Artemis Financials’ business requirement. The CIA triad is a security model used as a basis for defining three primary concepts of security in information technology (Andress, 2019). It consists of confidentiality, integrity, and availability. Confidentiality is the process of ensuring data is hidden. Integrity is the process of ensuring data has not been altered or replaced. Availability is ensuring that data is accessible when it is needed. Artemis Financials’ business requirement to protect data at rest - their long-term file storage. This will require using an encryption algorithm for encrypting and decrypting the data, as well as a hashing algorithm to ensure data has not been modified. The following details my recommended cipher for each of these purposes:

Encryption/Decryption

A common approach to encrypting data and then being able to decrypt it, is to use symmetric keys, asymmetric keys, or a combination of both. A symmetric solution means that the key used to encrypt the data is the same key that is used to decrypt the data. An asymmetric solution uses a public and private key system which are two different keys. The public key is used to encrypt the data, and the private key is used to decrypt the data. Since Artemis Financial is primarily concerned with their long-term file storage, we should consider a symmetrical algorithm. Overall, I am recommending AES for Artemis Financial. AES was the finalist in NIST’s standardization process during 1997 – 2000 (which is why it is now named AES) and is therefore supported by industry standards and federal regulations:

• NIST Federal Information Processing Standards Publication 197 (FIPS PUB 197)

• Federal Financial Institutions Examination Council (FFIEC)

• Cryptographic Module Validation Program (CMVP)

• Cryptographic Algorithm Validation Program (CAVP)

• ISO/IEC 18033-3:2010 (Block Ciphers)

Furthermore, AES is suitable for encryption of bulk data such as Artemis Financials’ long-term archive files due to its high-speed encryption, low computation cost, and is considered one of the most secure symmetric algorithms available. AES is a block cipher meaning that it operates on fixed sized blocks of data (128-bits). It has three key sizes: 128, 192, and 256-bits. Typically, the larger the key the more difficult it is to break the algorithm. Plaintext is encrypted by a series of transformations (rounds) that are based on the key size. According to Computer Security Division, I. T. L. (2016), all the keys mentioned above are approved for encrypting Government SECRET level information. At minimum, 192-, and 256-bit keys are required for processing TOP SECRET government information (Computer Security Division, I. T. L., 2016). A combination of high security, fast encryption, and low computational requirements helped AES (Rijndael) become the Government standard for encryption.

Random numbers play an important role in cryptography. We first need to distinguish between a truly random number and pseudorandom number. Truly random events are unpredictable and typically have no way to be reproduced. Pseudorandom numbers are used by computing systems to replicate the process of generating a number that cannot be predicted. The importance of random numbers is essential to how encryption algorithms work because we want the algorithms to be unbreakable. For example, “RSA encryption generates a public key by multiplying two large, random prime numbers together, and using these same prime numbers, generates a private key” (Daniel, 2021). If an algorithm is seeded with a set value, or a predictable value, then it could be easily broken. Therefore, is important for an algorithm to use random numbers instead of seeded numbers that do not change.

The history of encryption algorithms goes back as far as we have records of their existence. A notable example is the Caesar cipher, which is a type of substitution cipher that was thought to be used by Julius Caesar (Andress, 2019). As mentioned earlier, in the late 1990’s the U.S. Government needed a replacement for the Data Encryption Standard (DES) algorithm, and therefore created the AES contest. Rijndael won the competition and therefore became what is now known as AES, which is the Government approved standard for encryption. The concern for the future is with quantum computing. Many algorithms are considered unbreakable simply because of the time-complexity associated with successfully performing a brute-force attack on them.

However, quantum computing could change that. Grover’s algorithm is an example of a quantum algorithm that could possibly brute-force symmetric keys in less iterations than what is currently possible (Bernstein et al., 2009). It is also suggested that Grover’s algorithm could speed up collision attacks and pre-image attacks (Bernstein et al., 2009). In 2016 NIST initiated another competition to evaluate and standardize quantum-resistant public-key cryptographic algorithms (Computer Security Division, 2017). It is possible we will see standardization documents by 2024 for quantum-resistant algorithms. Until then, AES with a 256-bit key is considered quantum-resistant and is likely to be one of the most secure encryption algorithms for the foreseeable future.

Hashing

The previous discussion focused on encrypting and decrypting data to satisfy the “confidentiality” aspect of the CIA-Triad model. Hashing is a one-way “encryption” process that involves transforming data into a set of random, jumbled set of characters. The hashed value cannot be “decrypted”. We are motivated to use a hashing algorithm to satisfy the “integrity” aspect of the CIA-Triad model – ensuring data has not been altered. Two main factors to consider when selecting a hashing algorithm are that it produces a strong hash, and that it is resistant to collisions. A collision is when two different messages hash to the same checksum. This would likely indicate that the original message has been modified or replaced with a different message by a nefarious actor and gets through the system undetected.

The Secure Hashing Algorithm (SHA) is a family of hash functions that were developed for the purpose of generating checksums that are highly resistant to collisions. SHA has gone through numerous iterations since its inception in 1993 beginning with SHA-0. Today, the most widely used hash functions come from the SHA-2 family and include SHA-256, SHA-384, and SHA-512. SHA-3 is the next generation of the SHA family and is used as well.

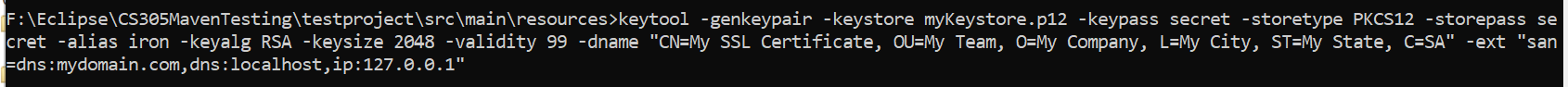
The main difference between SHA-256, SHA-384, and SHA-512 is the size of the message digest that is produced by the hashing algorithm. SHA-256 produces a 256-bit hash, SHA-384 produces a 384-bit hash, etc. The size of the hash can be an indicator of the strength of the algorithm, although the size of the hash could impact performance. However, studies have shown that the larger hash algorithms like SHA-384 and SHA-512 outperform SHA-256 (PyCryptodome, n.d.). A consideration for the future is that we should choose a hashing algorithm that is quantum resistant. Nakov (2023) identifies that algorithms in the SHA-2 (as well as SHA-3) are quantum resistant.

I am recommending SHA-384 for Artemis Financial for several reasons. First, it is a stronger algorithm compared to SHA-256. Even though SHA-256 has yet to be hacked (Thomas, 2023), it is likely to be more vulnerable to quantum attacks in the future compared to SHA-384 and SHA-512. To be “future proof”, both SHA-384 and SHA-512 are therefore considered better than SHA-256. SHA-512 is more current and may be considered the strongest, however it can be vulnerable to a length extension attack unless a truncated version is used (such as SHA-512/256). The intention is for Artemis Financial to use the checksum to verify that files have not been altered, and to verify public keys. With all other things being equal, SHA-384 might be a better choice in this case, as we will not need to worry about truncation issues associated with SHA-512. Furthermore, from the perspective of output size, SHA-384 would produce the larger value compared to SHA-512/256.

## Certificate Generation

The following is a screenshot of my certification generation process. Note that I used PKCS12 as my keystore format instead of JKS. Furthermore, I ended up having to add a Subject Alternate Name (SAN) to the certificate to identify a domain and an IP address of the localhost (172.0.0.1). My version of Chrome browser refused to accept my self-signed certificate without the SAN. I also experimented with using x.509 format instead of PKCS12. When using x.509 I did not have consistent behavior (i.e., I could not reliably connect to localhost using https) even when creating a new project from scratch. So, I ended up staying with PKCS12. I also attribute my solution for creating the certification based on Almalki’s (2023) GitHub solution and our textbook.

*Create the PKCS12 Keystore:*



*Listing of keystore:*

A picture containing text, screenshot, font

Description automatically generated

*Export of certificate:*

A screen shot of a computer

Description automatically generated with low confidence

## Deploy Cipher

The following is a screenshot showing the resulting checksum of data in my refactored code:

A screenshot of a computer

Description automatically generated

## Secure Communications

The following is a screenshot showing the web browser using HTTPS:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the localhost connected to the Tomcat server:

The following screenshots show output from the Dependency-Check tool:

A picture containing text, screenshot, font

Description automatically generated

A picture containing text, screenshot, font

Description automatically generated

The following is a screenshot of the Dependency-Check report before my code modifications:

A screenshot of a computer error

Description automatically generated with low confidence

The following is a screenshot of Dependency-Check results after my code modifications:

A screenshot of a computer error

Description automatically generated with low confidence

Note that no new vulnerabilities were discovered after my code modifications.

## Functional Testing

The following screenshots show the code I used for generating the checksum, setting up the end point, the application properties, and the pom file.

A screen shot of a computer program

Description automatically generated with low confidence

(*SHA-384 Hash in Java*, 2018)

As part of the code review, the main security consideration is the use of the try-catch block. We are motivated to use a try-catch to ensure that string value we inject into the MessageDigest class is valid. If we don’t use a try-catch block, and an invalid string value is used, then the program will fail without catching the NoSuchAlgorithmException. Applications can be vulnerable to exploitation when they crash in scenarios like this. Using a try-catch block is one of the best ways to prevent this type of exploitation.

The following is a screenshot of the /hash endpoint used to display the results of the checksum to the webpage:

A picture containing text, screenshot, software, multimedia software

Description automatically generated

The following are screenshots of the pom.xml file:

A screen shot of a computer program

Description automatically generated with medium confidence

A picture containing text, screenshot

Description automatically generated

The following is the application.properties file:

A screen shot of a computer

Description automatically generated with medium confidence

I also added the keystore and cert to the project resources:

A screenshot of a computer

Description automatically generated

The following screenshot shows the application running with my localhost connected and browsing the web page:

A screenshot of a computer

Description automatically generated with medium confidence

## Summary

Information systems are exposed to countless threats, risks, and vulnerabilities. It is not possible to completely secure an application using only one product, or only one defensive measure. Andress (2019) discusses implementing defense in depth at various layers of an information system. This might include layers such as: external network, network perimeter, internal network, host, application, and data (Andress, 2019). Furthermore, at each layer we will have multiple defense measures in place to protect against various vulnerabilities that the system might be exposed to. At this point in Artemis Financials Secure Software Development Life Cycle (SSDLC), we are focused on the application and data layers.

In the application layer, most applications will need to protect against attacks like cross-site scripting (XSS), spoofing sessions, automated account registration, privilege escalation, and many others. In the current state of the Artemis Financials SSDLC, we have implemented SSL/TLS to ensure authenticity and establish secure communication sessions. This is accomplished by creating a self-signed digital certificate that proves we are who we say we are.

In the data layer, we need to ensure Artemis Financials’ data is only readable by people who we trust. If financial information is readable by nefarious actors, made publicly available, or is altered, then this would be a major breach of security. Artemis Financial would face federal, state, and possible international penalties, as well as severe damage to their reputation. At this stage of the SSDLC, we are recommending the implementation of two measures to protect Artemis Financials’ data. First, we recommend AES-256 to encrypt their long-term archive files. Second, we recommend SHA-384 to accomplish integrity – proving that data has not been altered or replaced.

We followed a methodical process for implementing security – the Vulnerability Assessment Process Flow Diagram (VAPFD). This included identifying seven primary categories of the application at one layer, and seven categories at the second layer. The first layer is composed of: input validation, API’s, cryptography, client/server, code error, code quality, and encapsulation. The second layer includes performing code review of the following areas: views, models, controllers, data access, services, plug-ins, and API’s. We covered most of these categories in the first iteration of our report for Artemis Financial. The following review will cover areas of the VAPFD that pertain to this iteration of the report.

During the code refactoring process, we implemented cryptography, dealt with API’s, performed code error checking, and ensured our code changes were consistent with code quality. Cryptography was implemented using Java’s security framework to create a MessageDigest object and employing the SHA-384 for hashing. We also used SSL/TLS so that the web application could be run using secure HTTP (HTTPS) instead of HTTP which is unsecure. This process included creating a properties file and creating a self-signed digital certificate. We made a private key that only we know, and a public certificate that is used by external actors. The main API we used in the application is Spring Boot, which means we had to make sure vulnerabilities associated with Spring Boot were identified and remedied.

As mentioned earlier in this report, we used a try-catch block when implementing the hashing algorithm. This applies directly to the code error category of the VAPFD. The try-catch block ensures that if an unrecognizable algorithm attempted to be used, then the program will handle the error gracefully and without compromising our application. Lastly, code quality was implemented throughout the program by ensuring we adhered to recommended naming conventions and programming standards.

## Industry Standard Best Practices

Throughout the process of analyzing Artemis Financials’ business requirements, and determining a sound security posture for this application, we have followed a variety of industry standards and best practices. This included adhering to industry accepted security guidance from OWASP, NIST, and Oracles’ secure coding guidelines. We performed static analysis of the code using the OWASP Dependency-Checker to verify if the dependencies we are using have vulnerabilities. We also performed a manual review of the code which included reviewing all the application files to look for code that might produce vulnerabilities, as well as ensuring the application adhered to industry accepted coding convention such as Oracles’ “Code Conventions for the Java Programming Language”.

HTTPS is considered the standard for secure communication between devices over a network. We implemented HTTPS by creating a self-signed digital certificate, and ensured the client and server could communicate securely. We also implemented a hashing using one of the best algorithms available (SHA-384). Furthermore, we implemented file encryption using AES-256, which is the government standard for encryption. Finally, by using VAPFD as a guideline for producing a securing application and integrating this process into our development process we have created a secure software development lifecycle (SSDLC). This prepares Artemis Financial for the next stage of the software development process which includes DevSecOps – adding continuous security (CS) into a CI/CD pipeline.

The value of applying industry standards and best practices for secure coding helps ensure this application will meet Artemis Financials’ goals. Artemis Financial has recognized the need for secure applications, and since their business deals with highly sensitive information such as financial data, they must adhere to a multitude of federal, state, and intentional laws and regulations. Violations of these laws and regulations could result in fines and loss of licenses. Furthermore, Artemis Financials’ reputation must remain impeccable if they are to be considered a leader in financial management.

References

Almalki, F. (2023). *Generate self-signed PKCS#12 SSL certificate and export its keys using Java keytool and openssl.* GitHub. <https://gist.github.com/Eng-Fouad/6cdc8263068700ade87e4e3bf459a988>

Andress, J. (2019). *Foundations of Information Security: A Straightforward Introduction*. No Starch Press.

Bernstein, D. J., Buchmann, J., & Dahmen, E. (2009). *Post-Quantum Cryptography*. Springer Science & Business Media.

*Code Conventions for the Java Programming Language: 9. Naming Conventions*. (n.d.). Www.oracle.com. <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>

Computer Security Division, I. T. L. (2016). *AES Development - Cryptographic Standards and Guidelines | CSRC*. Nist.gov. <https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development>

Computer Security Division, I. T. L. (2017, January 3). *Post-Quantum Cryptography Standardization - Post-Quantum Cryptography | CSRC | CSRC*. CSRC | NIST. <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization>

*CS305 Vulnerability Assessment Process Flow Diagram*. (n.d.). Southern New Hampshire University.

*CVE - Common Vulnerabilities and Exposures (CVE)*. (2016). Mitre.org. <https://cve.mitre.org/index.html>

Daniel, B. (2021, May 4). *Symmetric vs. Asymmetric Encryption: What’s the Difference?* Www.trentonsystems.com. <https://www.trentonsystems.com/blog/symmetric-vs-asymmetric-encryption#:~:text=By%20far%20the%20biggest%20disadvantage>

Horstmann, C. S. (2022). *Core Java, Volume II - Advanced features* (Vol. 2). Pearson.

Jeganathan, S. (2019). DevSecOps: A Systemic Approach for Secure Software Development. *ISSA Journal*, *17*(11), 20–27. International Security & Counter Terrorism Reference Center.

Manico, J., & Detlefsen, A. (2015). *Iron-Clad Java: Building Secure Web Applications /*. New York, N.Y. McGraw-Hill Education.

Nakov, S. (2023). *Practical Cryptography for Developers*. Cryptobook.nakov.com. <https://cryptobook.nakov.com/>

NIST. (2019). *NVD - Home*. Nist.gov. <https://nvd.nist.gov/>

Oracle. (n.d.-a). *Java Security Standard Algorithm Names*. Docs.oracle.com. <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

Oracle. (n.d.-b). *MessageDigest (Java Platform SE 8 )*. Docs.oracle.com. Retrieved May 20, 2023, from <https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html>

Oracle. (2019). *Secure Coding Guidelines for Java SE*. Oracle.com. <https://www.oracle.com/java/technologies/javase/seccodeguide.html>

OWASP. (2019). *Input Validation OWASP Cheat Sheet Series*. Owasp.org. <https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html>

OWASP. (2021a). *A06 Vulnerable and Outdated Components - OWASP Top 10:2021*. Owasp.org. <https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/>

OWASP. (2021b). *OWASP Top Ten*. Owasp.org; OWASP. <https://owasp.org/www-project-top-ten/>

PyCryptodome. (n.d.). *SHA-384 — PyCryptodome 3.15.0 documentation*. Pycryptodome.readthedocs.io. Retrieved May 20, 2023, from <https://pycryptodome.readthedocs.io/en/latest/src/hash/sha384.html>

*SHA-384 Hash In Java*. (2018, September 27). GeeksforGeeks. <https://www.geeksforgeeks.org/sha-384-hash-in-java/>

Thomas, D. (2023, January 10). *Quantum Computers Break Encryption in China But Far From Cracking Bitcoin*. BeInCrypto. <https://beincrypto.com/quantum-computers-cannot-crack-bitcoin/>