****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc153037157)

[Document Revision History 3](#_Toc153037158)

[Client 3](#_Toc153037159)

[Developer 4](#_Toc153037160)

[1. Algorithm Cipher 4](#_Toc153037161)

[2. Certificate Generation 6](#_Toc153037162)

[3. Deploy Cipher 9](#_Toc153037163)

[4. Secure Communications 10](#_Toc153037164)

[5. Secondary Testing 11](#_Toc153037165)

[6. Functional Testing 13](#_Toc153037166)

[7. Summary 17](#_Toc153037167)

[8. Industry Standard Best Practices 18](#_Toc153037168)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **12/10/2023** | **Allan O’Driscoll** | **Initial Recommendations** |

## Client



## Developer

Allan O’Driscoll

## Algorithm Cipher

Artemis Financial has requested updates to their application to ensure secure file transfers and to add verification steps using a checksum. These additions will enhance the security of the application by protecting communication channels over the network and also by adding validation steps to maintain the integrity of the data. This is accomplished by using TLS and a secure hashing mechanism.

The first step in this process is to create an SSL certificate and configure the application to make use of it. The details of this process are discussed later in this document. An SSL certificate allows the server to implement encryption to protect data in transit. It does this by implementing protocols and algorithms that are chained together in a cipher suite. An application and its associated framework will generally support a list of cipher suites and will negotiate with the client to select one that is appropriate. The server may also indicate a preference while negotiating the protocol. A suggested cipher suite to support this is TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256. The cipher suite specifies each component in the chain separated by an underscore. For example, this cipher suite includes the following components (NIST, 2023, Scanigma, n.d.-a):

* TLS: Transport Layer Security
* ECDHE: Ephemeral Elliptic Curve Diffie-Hellman (key exchange)
* ECDSA: Elliptic Curve Digital Signature Algorithm (authentication method)
* AES\_128: Advanced Encryption Standard (key type with a 128-bit key)
* GCM: Galois Counter Mode (encryption mode)
* SHA256: Secure Hash Algorithm 256-bit

This cipher suite is supported by TLS versions 1.2 and 1.3 which are the two most recent versions of TLS. Note that TLS 1.3 has a similarly named variant called TLS\_AES\_128\_GCM\_SHA256 (Amazon, n.d.). This is not exactly the same but uses a similar suite of protocols and algorithms. Both of these cipher suites have an A+ rating from companies like Scanigma (Scanigma, n.d.-a; Scanigma, n.d.-b). The selection of a strong cipher is an important step when developing a good security posture.

The next step is ensuring the privacy of data stored by the application. All private and otherwise sensitive data should be encrypted when written to the file system or database. To accomplish this, the application can utilize encryption keys and APIs such as Tink. Tink is an open-source encryption library with support for multiple languages, including Java, C++, and Python (Tink, 2023a; Tink, 2023b). The administrator (or the application) will create a keyset that is stored in a secure location that is accessible by the application. The keyset can include primary and secondary entries that allow keys to be rotated as necessary. This enhances security by supporting the re-encryption of data and creating a mechanism to revoke keys that have been compromised. The application uses the keyset and related algorithms to encrypt sensitive data when it is written to disk and decrypt the same data when it needs to be loaded again. A suggested algorithm to utilize when encrypting local files is AES128\_GCM, which is recommended by the Tink community (Tink, 2023b). The algorithm includes the following components (NIST, 2023, Scanigma, n.d.-a):

* AES128: Advanced Encryption Standard (key type with a 128-bit key)
* GCM: Galois Counter Mode (encryption mode)

Depending on the use case, a cipher suite may be configured to use symmetric or asymmetric keys. Asymmetric keys are used in TLS to create secure communication channels over the network. The associated SSL certificate includes both public and private key parts. Data encrypted with one of the keys can only be decrypted with the other. The public key can be shared with anyone that needs to send encrypted data to the server. This happens in the key exchange phase of the TLS protocol and simplifies the process of establishing the connection. The private key, on the other hand, is kept securely on the server and is not shared with anyone. Symmetric keys are used for scenarios including local file encryption. A single key is used for both encryption and decryption of data. For this reason, it is important to keep the key in a secure location. Anyone with access to the key and knowledge of the associated algorithm will be able to use it to gain access to the encrypted data. Both types of keys are common and secure if properly managed. As mentioned earlier, the choice really depends on the use case.

Another important factor of encryption is the ability to generate secure random numbers. This is necessary because if an attacker was able to guess the sequence of numbers that you might choose, the algorithm would be easy to break. Generating secure encryption keys relies on establishing a high level of entropy as input. According to Manico and Detlefsen (2015), it is difficult to produce truly random numbers. However, there are libraries that do a reasonable job at this. The SecureRandom class provided by Java is a good choice. Having a good source of randomness will ultimately result in a more secure system.

Finally, hashing algorithms are an important part of almost all secure systems. They are used to authenticate SSL certificates when establishing secure communication channels and are also used in the generation of checksums to validate the integrity of the data. A recommended hashing algorithm is SHA-512. This algorithm has been approved and recommended for use by the National Institute of Standards and Technology (NIST, 2017). The bit level used by the cipher (or hash function) specifies the strength of the resulting hash. For example, SHA-256 will produce hashes of 256 bits (32 bytes), and SHA-512 will produce hashes of 512 bits (64 bytes). The hash function uses a mathematical algorithm to reduce the data to a corresponding hash value. Each bit level has a very large but still finite number of possible values. For example, SHA-256 has combinations, while SHA-512 has combinations. One measure of strength is called collision resistance and is measured in bits (Walker et al., 2009; Dang, 2012). A collision happens when you have two or more input values that map to the same hash value. Mathematically speaking, if and are both input messages, and is a hash function, then a collision happens when , but yet, . This is possible since there are an infinite number of different input messages but a finite number of resulting hash values. The collision resistance is generally half of the bit length specified by the algorithm (Dang, 2012). In other words, a brute force attack might need to attempt combinations before finding a collision that would break SHA-512. The bit length is similarly important for encryption algorithms. In general, a hash or cipher using a higher number of bits will require more time and computer resources to break (possibly in thousands or millions of years).

Encryption has been used in one form or another throughout history. However, in recent years, it has become a critical part of our society. This is partly due to the massive amounts of data that we collect and the ease of its transmission. Without encryption, nothing would be private, and you wouldn’t be able to maintain the security of your assets or personal information (bank accounts, medical history, personal property, etc.). Academic institutions and private businesses have all been heavily involved in research related to encryption technology. As a result, there has been a constant stream of new algorithms and protocols.

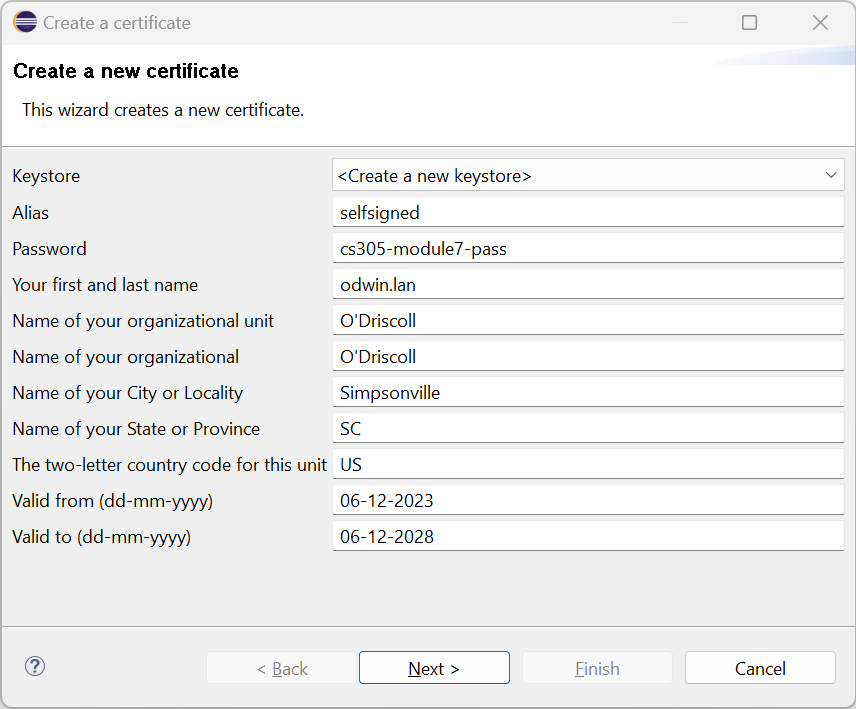
For example, DES was developed by IBM in the 1970s, and it was adopted by the U.S. National Bureau of Standards shortly after its introduction (Newton, 2015; Antonenko, 2023). The algorithm remained relevant for a while but was eventually replaced by AES due to security weaknesses. AES supports several different bit lengths, including 128, 192, and 256 bits, with the higher bit lengths being more resistant to attacks. AES is considered secure and is a standard approved by the federal government (NIST, 2021). However, as with other technology, it will eventually be broken.

Hashing algorithms have gone through a similar evolution. SHA-1 and MD5 were once considered secure, but they can now be broken with modern hardware. An example was provided by researchers from Google and CWI Amsterdam. The team was able to construct multiple PDF files with different content that resulted in the same SHA-1 checksum (Stevens et al., 2017). These algorithms have since been replaced by more secure mechanisms, including SHA-256 and SHA-512 (NIST, 2017).

Governments have also been heavily involved in the development of encryption technology. Not only in developing and approving encryption standards but also in creating laws to regulate its use, protect their interests, and protect the public (Garfinkel & Spafford, 2002). For example, the U.S. Department of Commerce has enacted regulations that impose export restrictions on certain types of encryption technology (BIS, n.d.). This prevents companies from selling technologies (like encryption) to embargoed states. Other countries have passed strict privacy laws and imposed fines for companies that do not take steps to protect private data. The GDPR, passed by the European Union, is a good example of this (GDPR.EU, n.d.). Encryption, like every other technology, will continue to evolve. As new technologies are developed, software will need to evolve along with it.

## Certificate Generation

An SSL certificate can be generated using the Java keytool, Eclipse with the keytool plugin, or other tools. Since Artemis Financials’ application is based on Java, it is reasonable to use the tools provided by the JRE or Eclipse for this purpose. I tried several methods for generating a certificate. The first attempt was to use Eclipse with the Keytool plugin to manage these certificates. A certificate was created, as shown in the following screenshot.



The required information was entered on the form, including the common name “odwin.lan,” which is the name of my laptop on my home network. This certificate was exported using the Eclipse keytool plugin and then viewed using the standard Java keytool utility. This is shown in the following screenshot.

A screenshot of a computer program

Description automatically generated

It was observed that the Eclipse plugin generates certificates that are insecure. The signature algorithm (SHA1withRSA) and public key algorithm (1024-bit RSA) are both weak. The tool does not enable these settings to be customized, and a weak certificate will not be acceptable to the customer. The second method was to utilize the Java keytool provided by the JRE. Another certificate was created, as shown in the following screenshot:

A screenshot of a computer program

Description automatically generated

The Java keytool provides more flexibility in the encryption algorithm and key length. This second certificate was exported using the Java keytool and viewed as shown in the following screenshot.

A screenshot of a computer program

Description automatically generated

Note that this new certificate uses a signature algorithm (SHA384withRSA) and key algorithm (2048-bit RSA) that are more secure. I was also able to specify a subject alternative name that satisfies another security requirement imposed by recent browsers (Medley, 2017). Without the addition of the SAN extension, the browser would still consider the connection insecure. For more details, see the discussion on functional testing later in this document.

## Deploy Cipher

The source code for the application has been modified to implement a checksum verification using a cryptographic hash function. The code for this update can be found in the attached zip file in the HashController.java file. The process for obtaining the checksum is as follows:

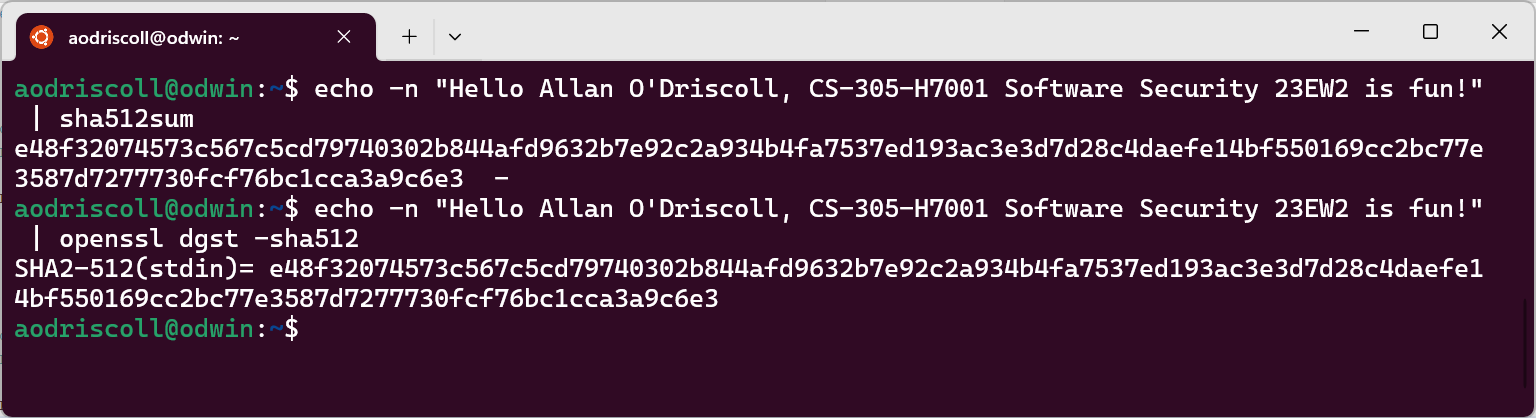
1. Construct a Java String containing the message to be hashed.
2. Convert the message to a byte array using UTF-8 as the encoding for the character set.
3. Get an instance of the message digester for the given algorithm. The selected algorithm for the current example is SHA-512.
4. Use the digester to process the byte array containing the input message. This produces another byte array containing the hash value.
5. Convert the byte array containing the hash value to a hexadecimal string.
6. Construct a response containing the original message, the algorithm name, and the computed hash value.
7. Return the response to the caller.

With this code in place, the user can hit the “/hash” endpoint of the REST application and see the generated checksum value. This looks like the following using the Mozilla Firefox browser:

A screenshot of a computer

Description automatically generated

The hash value can be verified using other programs such as OpenSSL and sha512sum. The SHA-512 checksums generated by these tools are consistent with the ones generated by the application.



## Secure Communications

The source code for the application has been modified to implement secure TLS-based communication between the browser-based client and the server. This involved generating a certificate as described in section 2 and then supplying the necessary configuration in the application.properties file. The code for this update can be found in the attached zip file.

Note that I experimented with the configuration that allowed the specification of specific protocols and cipher suites. In the end, I found that the default protocols and cipher suites provided by Java and Spring were sufficient. As Manico and Detlefsen (2015) mention in their book, the defaults are generally good, so many times, it’s best to just leave them alone. I can now browse to the “/hash” endpoint using a secure HTTPS connection in my browser. This looks like the following in Microsoft Edge:

A screenshot of a computer

Description automatically generated

The screenshot shows that the connection is secured using the installed SSL certificate. I was also interested in seeing the full set of cipher suites that were supported by the application. I ran the Linux sslscan tool to generate a list of preferred and accepted ciphers. These are shown below:

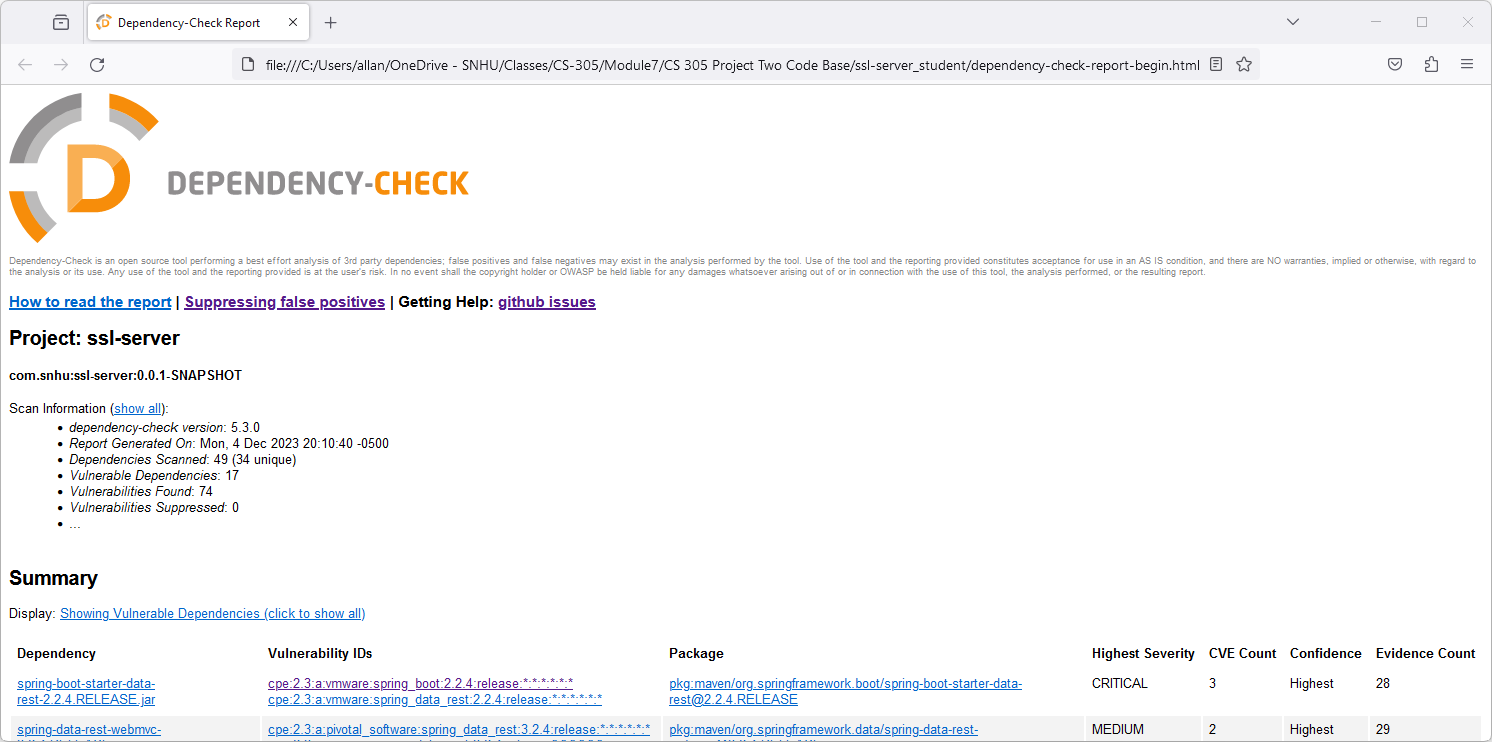
A screenshot of a computer program

Description automatically generated

Notice that TLS\_AES\_128\_GCM\_SHA256 is the first item listed and is the preferred cipher suite when using TLS version 1.3. As mentioned earlier, this cipher suite has a good rating.

## Secondary Testing

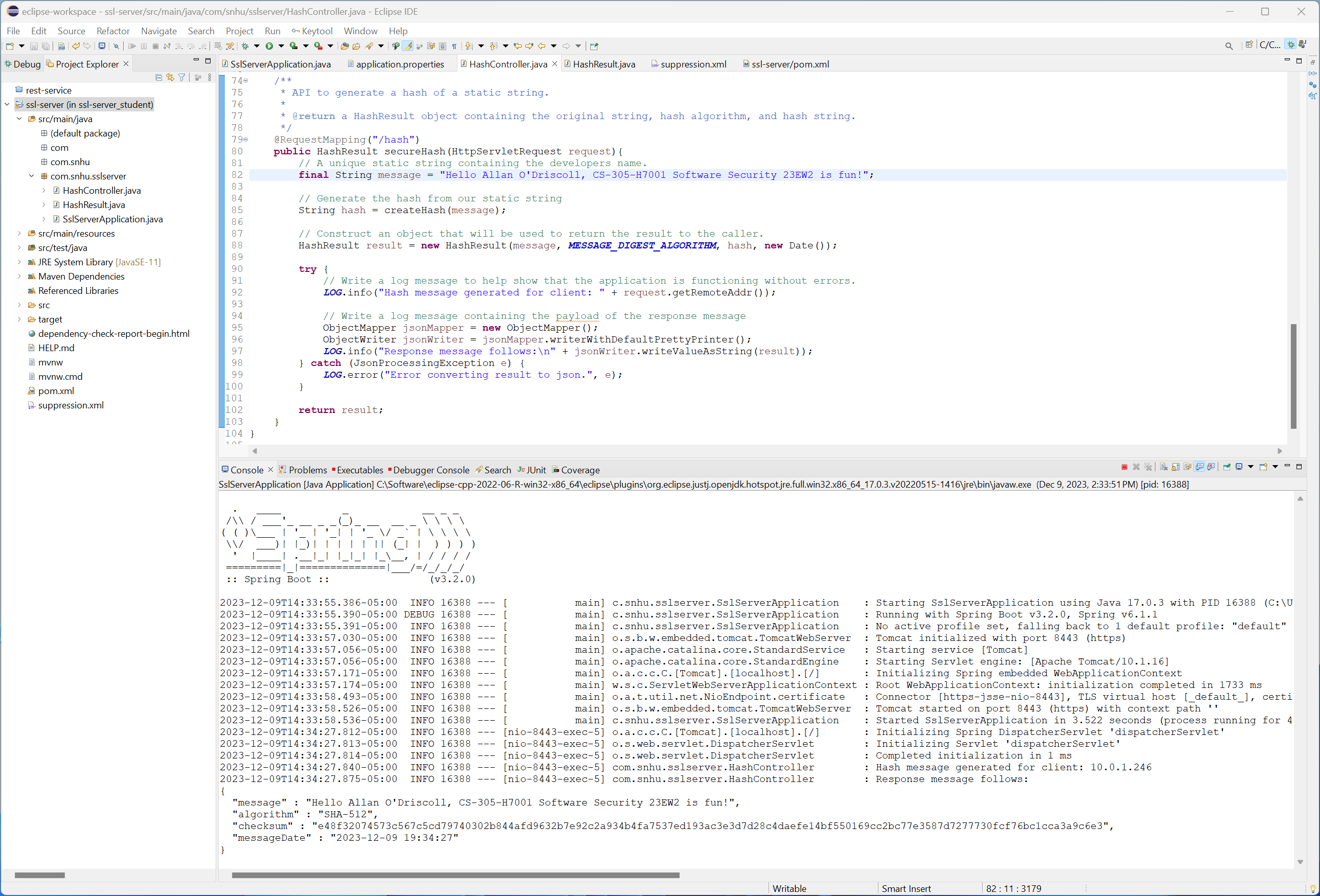
Before refactoring the code, I ran a dependency scan to check for vulnerabilities in the project. The report indicated that 49 dependencies were scanned, and 17 of these were found to have vulnerabilities. A total of 74 vulnerabilities were identified. The following is a screenshot from the header of this report.



A copy of this report is included in the zip file in a file called “CS 305 Project Two Code Base\ssl-server\_student\dependency-check-report-before.html.” To address security concerns and implement the requested functionality, I’ve refactored the code as follows:

* Generate an SSL certificate to ensure that all communications between the client browser and server are encrypted and secure. See the “CS 305 Project Two Code Base\ssl-server\_student\src\main\resources” for copies of the certificate.
* Configure Spring to use the SSL certificate by adding configuration to the application.properties file.
* Create a new Rest Handler class called HashController.java. Configure this class as the endpoint for the “/hash” path.
* Create a new POJO called HashResult.java to encapsulate the contents of the API result. This class is serialized to JSON before being returned to the client.
* Implement a hash function using the MessageDigest and other Java APIs. The implementation utilizes the SHA-512 algorithm to generate the hash. This also resulted in the inclusion of a new library called commons-codec-1.16.0.jar, which is used to convert byte arrays to hexadecimal strings.
* Correct as many vulnerabilities as possible to reduce the possibility of hidden security issues in the application. To accomplish this, I upgraded to Spring Boot 3.2.0 and Java 11 and then explicitly upgraded several other libraries, including logback-core-1.4.14.jar, logback-classic-1.4.14.jar, jackson-databind-2.16.0.jar.
* Reviewed the remaining six vulnerabilities and found evidence that they are false positives. In all cases, this was because the vulnerability referenced a different library than the one that was being used. Details for the suppressions were included in the suppression.xml file.

To demonstrate that the refactored code runs without errors, I’ve included a screenshot of the code execution inside the Eclipse IDE. This shows the primary method that processes the “/hash” endpoint. It also shows the output of log messages that were generated from a single request in the browser.



A screenshot of this same request inside the Mozilla Firefox Browser is included below.

A screenshot of a computer

Description automatically generated

Note that I added the messageDate field to the response in an effort to show that the output is unique and that the message generated by the application is the same as shown in the browser. Please also note that the output shows that the Spring version has been updated to version 3.2.0. This was a key part of the refactoring that helped eliminate a large number of potential security vulnerabilities.

After completing the necessary refactoring, I ran a secondary dependency check to verify that all issues were addressed and that no new issues were introduced. The following is a screenshot from the header of this report.

A screenshot of a computer

Description automatically generated

The report indicated that 47 dependencies were scanned, and zero of these were found to have vulnerabilities. Note that six vulnerabilities were analyzed and manually suppressed due to mismatches in the CPE (the CVEs were for libraries that are not included in this project). The details for the suppressions have been documented in the suppression.xml file. As discussed earlier, the drop in vulnerability count was primarily due to using a new version of the framework and leveraging the work done by the open-source community in monitoring and fixing these issues.

A copy of this second report is included in the zip file in a file called “CS 305 Project Two Code Base\ssl-server\_student\dependency-check-report-after.html.”

## Functional Testing

In addition to scans using the dependency-check-maven plugin, I conducted functional testing to uncover any additional security vulnerabilities that were present in the application. The first issue was identified while enabling SSL for the server. I found that the SSL certificate generated by the Keytool plugin for Eclipse generated insecure certificates. I verified that I was using the latest version of the tool and also checked for configuration settings that would allow me to customize the certificate. None were found. I concluded that the Eclipse keytool plugin is useful for development purposes but not for production environments. In addition, browsers such as Google Chrome no longer trust certificates if they don’t specify a subject alternative name (Medley, 2017). Even with the certificate installed, configured, and added to the system’s trusted CA certificate store, it still produced an error such as the following:

A screenshot of a computer

Description automatically generated

To address this problem, I recreated the certificate using the Java keytool utility. This allowed me to create a self-signed certificate that was more secure and included the required subject’s alternative name. This is discussed in more detail in section 2, which talks about certificate generation.

The next issue that I found is that Spring’s whitelabel error handler is enabled by default. The default error handler is configurable (via the application.properties file), but leaving it enabled has the potential to leak sensitive information to an attacker. The application should write detailed error messages to a local log file, but error messages that are presented to the end user should be sanitized. The following screenshot shows an example of an error that could be produced in this application by the default white label error handler included with Spring.

A screenshot of a computer

Description automatically generated

An attacker could use information like this to learn more about the system and formulate further attacks. To correct this, the white label error handler was turned off in the application.properties configuration file.

server.error.whitelabel.enabled=false

In addition, custom error pages that present a generic error message to the user were created. The process for creating custom error pages is simply a matter of creating a set of properly named HTML files and storing them in the “src/main/resources/public/error” folder (Spring.io, n.d.; Hossain, 2023). In this case, I created custom pages for HTTP 404 and 500 errors. More advanced error handlers can also be added to customize error messages. A screenshot of the resulting error page looks like the following:

A computer screen shot of a computer error

Description automatically generated

Note that an error message was purposefully introduced in the code to test error handling functionality. This was also necessary for demonstration purposes, as shown in the previous two screenshots. The error has been removed from the final version.

The last issue is related to application logging that was introduced during development. The initial version of the code logged a set of informational messages with every request. This is shown in some of the previous screenshots and was useful for testing and demonstration purposes. However, it would be unacceptable in a production environment. Excessive logging can be exploited to produce a denial-of-service attack (Oracle, 2023, sec. 1-1). I did not want to remove the logging from the application because it was useful for development. Instead, I changed the log level to DEBUG and wrapped the entire block in a condition to prevent logging if DEBUG is not enabled. In addition, I set the log level to INFO in the application.properties file.

logging.level.com.snhu.sslserver: INFO

This is a screenshot that shows the code executes without errors.

A screenshot of a computer

Description automatically generated

A screenshot of what the resulting request looks like in the browser is shown below.

A screenshot of a computer

Description automatically generated

## Summary

The refactoring process used by this project included a review of the application from a security perspective. Specific architectural reviews included an analysis of the application’s use of cryptography, client/server interactions, error handling, code quality, and encapsulation. This led to further reviews of the model, view, and controllers implemented by the code. The initial version of the application did not use secure communications and had no functionality to validate the integrity of data sent or received by the client. The application was also found to be vulnerable to a large number of known security issues. Improvements in these areas were specifically requested by the customer.

After a thorough review of the architecture, the source code for Artemis Financial has been refactored to include best practices for security, web development, and secure coding practices. Major areas of refactoring included the following:

* Generating a secure SSL certificate using industry-standard tools.
* Modifying the application to use SSL certificates and TLS for secure communication.
* Implementing a process that enables the verification of data using a checksum.
* Implementing a model that encapsulates the data returned to the client application.
* Resolving known security vulnerabilities by updating the version of the base framework and supporting Java libraries.
* Updating the application’s default error handling (white label errors) to address problems with potential information disclosure.
* Ensuring that logging is used in the application in a way that is not vulnerable to denial-of-service attacks.

As with many things in software development, code improvements happen in iterations. The first iteration of this project included configuration changes to implement secure communication as well as the implementation of a checksum mechanism for data validation. These changes were tested, and further improvements were made. For example, the initial SSL certificate created by the Eclipse keytool plugin was found to be insecure. Research was conducted to find a solution, and a new certificate was created. The next iteration included an in-depth dependency check to identify known vulnerabilities in the framework and other libraries. Most of these issues were resolved by upgrading to newer versions of the framework and its dependencies. The remaining issues were researched and found to be false positives. These latter items were added to a suppression file included with the project. The last iteration included functional testing of the application. Several new issues were identified in this phase, including error handling that could expose internal application details to an attacker and logging that was vulnerable to denial-of-service attacks. These last issues were addressed by implementing additional features and code changes in the application.

In addition to code refactoring, this document has also discussed encryption ciphers and hashing algorithms in some detail. The evolution of encryption, key types, and bit levels have been reviewed. Encryption ciphers and hashing algorithms have been recommended and used in the implementation of the refactored code. Finally, details of a thorough security review have been documented.

## Industry Standard Best Practices

As a developer, it’s important to have a background in the best practices of secure software development. It’s also necessary to have a set of industry-standard tools to rely on. Security is a complex topic and not generally something that you can tackle on your own. For this reason, I have followed best practices in analyzing the application for security vulnerabilities. This included a review of the application’s architecture using a vulnerability assessment process flow. It also included the use of industry-standard tools, such as the OWASP dependency checker, to identify areas of concern within the application. These tools have been integrated with the build process and will be executed each time the code is built. I have followed best practices in the generation and deployment of SSL certificates to establish a secure communications layer between the client and the server. I’ve also used a set of APIs that implement standard algorithms for encryption and hashing. Using standard algorithms helps ensure operability between different applications.

Establishing best practices and following standards like the ones discussed in this document helps to ensure that future development efforts will result in fewer security vulnerabilities. Ultimately, this protects all parties involved. The developers will be able to spend their time developing new features instead of investigating security issues. The customer will enjoy a reduced risk of attacks that result in data breaches. This lowers the risk of lawsuits and the related financial liability. Finally, the end user will be able to rest, knowing that their private data is secure. Spending the time to understand and implement best practices related to security benefits everyone.

**References**

Amazon. (n.d.). TLS listeners for your network load balancer. Elastic Load Balancing. https://docs.aws.amazon.com/elasticloadbalancing/latest/network/create-tls-listener.html

Antonenko, D. (2023, May 11). Understanding the data encryption standard (DES). Business Tech Weekly. https://www.businesstechweekly.com/cybersecurity/data-security/data-encryption-standard/

BIS. (n.d.). Export administration regulations (EAR). https://www.bis.doc.gov/index.php/regulations/export-administration-regulations-ear

Dang. (2012). Recommendation for Applications Using Approved Hash Algorithms, NIST Computer Security Information Technology Laboratory. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-107r1.pdf

Garfinkel, S., & Spafford, G. (2002). Web security, privacy, and Commerce. O’Reilly.

GDPR.EU. (n.d.). General Data Protection Regulation (GDPR). GDPR Archives. https://gdpr.eu/tag/gdpr/

Hossain, I. (2023, September 27). Spring boot whitelabel error handling: Under the Hood. Medium. https://blog.stackademic.com/spring-boot-whitelabel-error-handling-under-the-hood-df9376897e37

Manico, J., & Detlefsen, A. (2015). Iron-clad java: Building secure web applications. McGraw-Hill Education.

Medley, J. (2017). Deprecations and removals in Chrome 58. Chrome for Developers. https://developer.chrome.com/blog/chrome-58-deprecations/

Newton, G. E. (2015, August 7). The evolution of encryption. Wired. https://www.wired.com/insights/2013/05/the-evolution-of-encryption/

NIST. (2017). NIST policy on hash functions - hash functions. Computer Security Resource Center. https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions

(NIST, 2021) Advanced Encryption Standard (AES). https://www.nist.gov/publications/advanced-encryption-standard-aes

NIST. (2023, October 17). Glossary. COMPUTER SECURITY RESOURCE CENTER. https://csrc.nist.gov/glossary/

Oracle. (2023, May). Secure Coding Guidelines for Java SE. Secure coding guidelines for java SE. https://www.oracle.com/java/technologies/javase/seccodeguide.html

OWASP. (n.d.). Error handling cheat sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Error\_Handling\_Cheat\_Sheet.html

Scanigma. (n.d.a). Cipher Suite: TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256. KNOWLEDGE BASE. https://scanigma.com/knowledge-base/tls/ciphersuite/tls-ecdhe-ecdsa-with-aes-128-gcm-sha256

Scanigma. (n.d.b). Cipher Suite: TLS\_AES\_128\_GCM\_SHA256. KNOWLEDGE BASE. https://scanigma.com/knowledge-base/tls/ciphersuite/tls-aes-128-gcm-sha256

Spring.io. (n.d.). Developing web applications. https://docs.spring.io/spring-boot/docs/1.4.3.RELEASE/reference/html/boot-features-developing-web-applications.html

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017). The First collision for full SHA-1. Shattered. https://shattered.it/static/shattered.pdf

Tink. (2023a). What is Tink? | tink | google for developers. Google. https://developers.google.com/tink/what-is

Tink. (2023b). I want to encrypt data | tink | google for developers. Google. https://developers.google.com/tink/encrypt-data

Walker, J., Kounavis, M., Gueron, S., & Graunke, G. (2009). Recent Contributions to Cryptographic Hash Functions. Intel Technology Journal, 13(2), 80–95.

Wang, X., & Yu, H. (2005). How to break MD5 and other hash functions. http://merlot.usc.edu/csac-f06/papers/Wang05a.pdf