CS763 Project 3

Implementing Crypto

Alessandro Allegranzi

Table of Contents

Github Repo

Repository: https://github.com/1-8192/CS763_project

Files I added/changed to implement added security:

Encryption Utility Class

String Attribute Converter

Double Attribute Converter

JWT 4 Hour Expiration Change

Report

Security Feature

The Trackr application already featured JWT tokens for authentication and hashed passwords using bcrypt for persistent storage, so I implemented database encryption.

Database encryption is an important security measure employed to safeguard sensitive information stored in databases, like bank account information or identifying information, by converting it into unreadable code that can only be deciphered with the appropriate cryptographic key [3]. The process involves applying encryption algorithms to the data before it is stored, rendering it effectively indecipherable to unauthorized entities. DB encryption protects sensitive data, such as personal information, financial records, and intellectual property, from unauthorized access, ensuring confidentiality and integrity [3].

DB encryption can mitigate system breaches because in the event of a security breach, encrypted data remains incomprehensible to unauthorized parties, limiting the potential damage and protecting the privacy of individuals. Additionally, database encryption aids in compliance with various data protection regulations and standards, such as the General Data Protection Regulation (GDPR) and the Health Insurance Portability and Accountability Act (HIPAA), which mandate the protection of sensitive information [3].

Very small change, but, additionally, I shortened the expiration date for the JWT token from the default (which I think is 20 minutes) to 10 minutes, to decrease the potential vulnerabilities in that feature.

Implementation Details

I implemented database encryption using JPA Attribute converters [1]. The Trackr application was built using an H2 database and JPA ORM, so I decided to leverage the AttributeConverter interface to implement encryption.

Before Implementing the attribute converters, I first built out a utility class to handle the encryption. The class file is viewable in GH here; I'll also include the code below:

```
/**
 * Encryption utility class to encrypt data for database storage.
 */
@Component
public class EncryptionUtility {

    // Bringing in the encryption key and initialization vector from
application context.

    @Value("${encrypt_key}")
    private String key;

    @Value("${init_vector}")
    private String vector;

    // Using the AES symmetric algorithm for encryption. Used the standard
Cipher transformation example from documentation
    // for AES 128 bit encryption.
    private final String cryptoAlgo = " AES/GCM/NoPadding ";
    private final String algo = "AES";

    /**
     * Encrypt string types.
     * @param value String value we want to encrypt
     * @return encrypted string.
     * @throws RuntimeException
     */
    public String encryptString(String value) throws RuntimeException {
        try {
            IvParameterSpec iv = new
IvParameterSpec(vector.getBytes(StandardCharsets.UTF_8));
            SecretKeySpec spec = new
SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), algo);

            Cipher cipher = Cipher.getInstance(cryptoAlgo);
            cipher.init(Cipher.ENCRYPT_MODE, spec, iv);

            byte[] encrypted = cipher.doFinal(value.getBytes());
            return Base64.encodeBase64String(encrypted);
        } catch (NoSuchPaddingException | NoSuchAlgorithmException |
InvalidKeyException
                | InvalidAlgorithmParameterException |
IllegalBlockSizeException | BadPaddingException e) {
            throw new RuntimeException(e);
        }
```

```
    }

    /**
     * Method to decrypt string.
     * @param encrypted encrypted string
     * @return decrypted String
     * @throws RuntimeException
     */
    public String decryptString(String encrypted) throws RuntimeException {
        try {
            IvParameterSpec iv = new
IvParameterSpec(vector.getBytes(StandardCharsets.UTF_8));
            SecretKeySpec spec = new
SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), algo);

            Cipher cipher = Cipher.getInstance(cryptoAlgo);
            cipher.init(Cipher.DECRYPT_MODE, spec, iv);

            byte[] original = cipher.doFinal(Base64.decodeBase64(encrypted));
            return new String(original);
        } catch (NoSuchPaddingException | NoSuchAlgorithmException |
InvalidKeyException
                | InvalidAlgorithmParameterException |
IllegalBlockSizeException | BadPaddingException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Encrypt double types.
     * @param value double to encrypt.
     * @return encrypted string.
     */
    public String encryptDouble(double value) {
        return encryptString(Double.toString(value));
    }

    /**
     * Decrypt double.
     * @param encrypted encrypted string.
     * @return double that was decrypted.
     */
    public double decryptDouble(String encrypted) {
        String decryptedValue = decryptString(encrypted);
        if (decryptedValue != null) {
            return Double.parseDouble(decryptedValue);
        }
        return Double.NaN; // Return NaN for failure or invalid input
    }
}
```

Some notes on the above code:

1. I am injecting the encryption key and initialization vector from application context, so as not to sore the plain string in the file. (See ptifalls section below about the key)

2. I decided to use the AES128 symmetric algorithm for encryption because it is a standard option in the Ciper class [2]. It is also a widely used, secure standard for encryption that is stronger than DES or Triple DES [4].
3. I am using the Cipher class [2] to run the actual encryption. It is a Java standard and is the core of the Java Cryptographic Extension (JCE) framework [2].
4. Because the application also stores account balances and other sensitive numeric information as double types, I had to implement double encryption and decryption methods as wrappers around the string methods to convert types and make it work.

I then created the Attribute Converters. Here is a link to the String converter. I also had to create a separate Attribute Converter for the double type. Here is the code for one the classes:

```
/**
 * Class to encrypt string attributes.
 */
public class EncryptionStringConverter implements
AttributeConverter<String,String> {
    @Autowired
    EncryptionUtility encryptionUtility;

    @Override
    public String convertToDatabaseColumn(String s) {
        return encryptionUtility.encryptString(s);
    }

    @Override
    public String convertToEntityAttribute(String s) {
        return encryptionUtility.decryptString(s);
    }
}
```

This was pretty simple as I just had to override two methods from the AttributeConverter interface to convert attributes to columns and vice versa. Those methods are basically wrappers around calls to the encryption utility class I shared above.

The final step was then adding an annotation to the existing Trackr Entities. Here are 2 examples, one for a string type and one for a double type, from the BankAccount entity.

```
@Column(nullable = false, length = 20)
@NotNull(message = CommonConstants.BLANK_ACCOUNT_TYPE)
@Enumerated(EnumType.STRING)
@Convert(converter = EncryptionStringConverter.class)
private ACCOUNT_TYPE accountType;
```

```
@Column(nullable = false, precision = 2)
@PositiveOrZero(message = CommonConstants.INVALID_BALANCE_VALUE)
@Convert(converter = EncryptionDoubleConverter.class)
private double balance;
```

Through the magic of Spring I just had to include the @Convert annotation and specify the converter. The framework handles the rest and applies the encryption to those database columns.

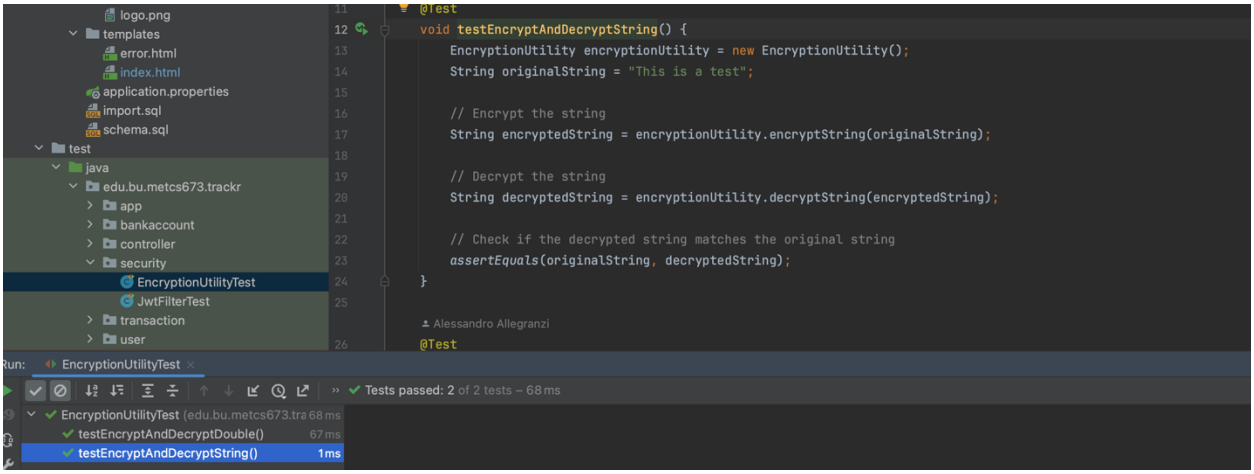
The main pitfall of this implementation is the exposed encryption key. Due to time constraints, I wasn't able to look into storing the key in Hashicorp Vault, or another secret manager. Storing the key in plaintext in the Github repository is a very big issue and would not be a production-ready implementation of the feature. I included it in the application.properties file to inject into context for now, but ideally I would be loading that from a key manager rather than hard coding the string. Long-term, I would look into storing the key in vault and injecting it into application configs. Another negative aspect is I used the standard Cipher class 128 bit AES algo. Using AES256 would have been a more secure option, and for storing user bank account information, the safest option possible would be best.

I also made a small change to the existing JWT token implementation to set the expiration time to 10 minutes rather than the standard 20. It was a one line change, but easy wins are still wins. The JWT Util class was also injecting an application property strangely, so I cleaned that up by using the @Value annotation to reference the variable stored in the application file rather than the string itself. I also removed the redundant constructor.

```
@Value("${jwt_secret}")
public String jwtSecret;
```

Testing

I added unit tests for the encryption classes for some automated test coverage. Below is a screenshot of the tests passing, showing encryption works. I also tested the application locally manually, and database interaction still worked well.



I also ran a ZAP DAST scan as part of a github action with my code merge. The full scan is available as a downloadable zip file here. The results do not show any sql injection vulnerabilities, which is a good sign. ZAP technically does alert SQL injection and other DB vulnerabilities, but it may not be a great tool for identifying database vulnerabilities mitigated by encryption [5]. Screenshot of the summary below:

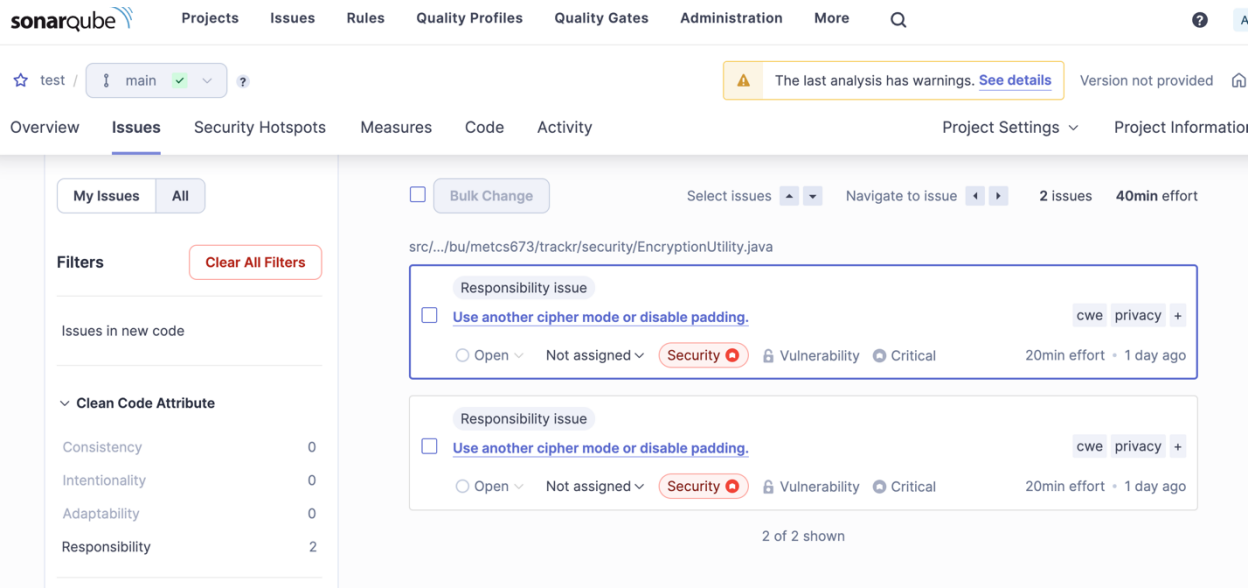
Summary of Alerts

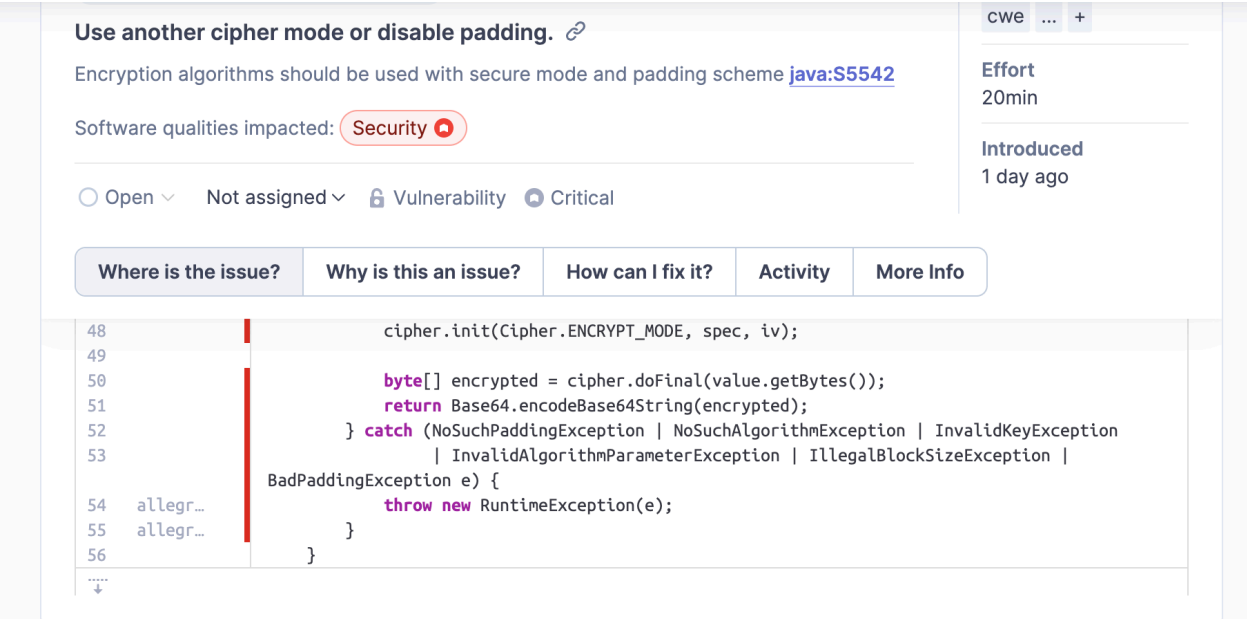
Risk Level	Number of Alerts
High	0
Medium	2
Low	3
Informational	10
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	1
Missing Anti-clickjacking Header	Medium	1
Cookie No HttpOnly Flag	Low	1
Cookie without SameSite Attribute	Low	1
Permissions Policy Header Not Set	Low	2
Base64 Disclosure	Informational	1
Information Disclosure - Suspicious Comments	Informational	13
Loosely Scoped Cookie	Informational	2
Modern Web Application	Informational	1
Non-Storable Content	Informational	4
Sec-Fetch-Dest Header is Missing	Informational	3
Sec-Fetch-Mode Header is Missing	Informational	3
Sec-Fetch-Site Header is Missing	Informational	3
Sec-Fetch-User Header is Missing	Informational	3
Session Management Response Identified	Informational	3

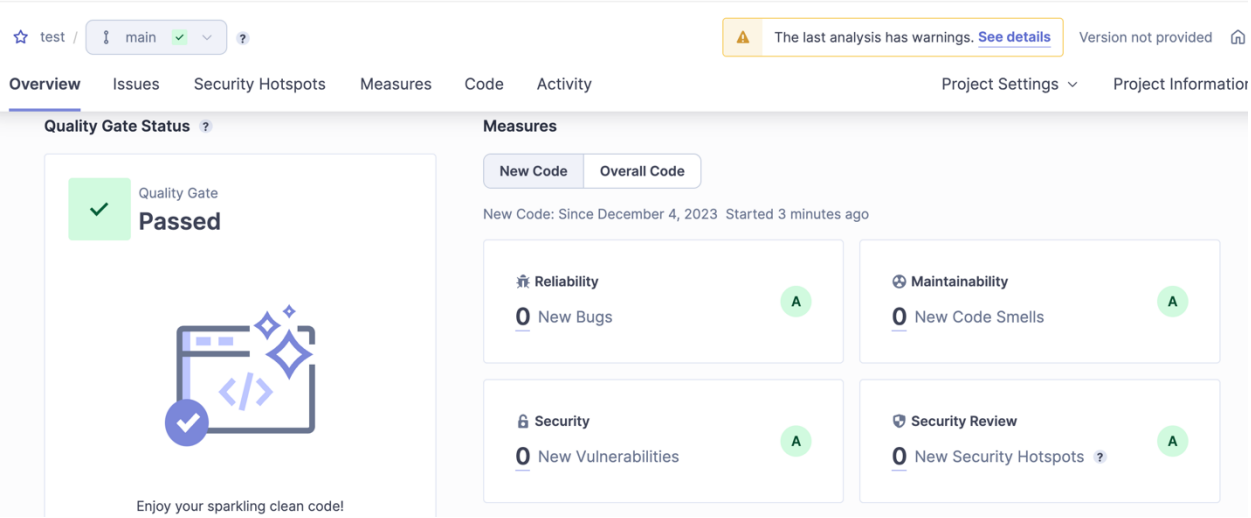
I ran a Sonarqube SAST scan locally, which showed a vulnerability related to the original encryption algorithm I was using.





I followed Sonarqube's suggestion and swapped out the "AES/CBC/PKCS5PADDING" algorithm for "AES/GCM/NoPadding" crypto algorithm for Cipher.

The successive scan showed the vulnerabilities were resolved.



References

[1] “JPA Attribute Converters” *Baeldung*. https://www.baeldung.com/jpa-attribute-converters

[2] “Class Cipher” *Oracle Docs*. https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html

[3] Gulen, Kerem. “Cracking the code: How database encryption keeps your data safe?” *Dataconomy*. https://dataconomy.com/2023/04/11/what-is-database-encryption-types-methods/ April 11, 2023.

[4] “Advanced Encryption Standard (AES)” *GeeksforGeeks*. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

[5] “SQL Injection” *ZAP Docs*. https://www.zaproxy.org/docs/alerts/40018/