Android Security Practices: AES Data Encryption

CoRide

Ahmed Nesar Tahsin Choudhury (02)
Mehrajul Abadin Miraj (20)
Jannatin Tajri (42)

Introduction

Android security practices are crucial in Android development to safeguard user data and privacy. Adhering to these practices helps prevent vulnerabilities, malware, and unauthorized access, ensuring a secure user experience. It also builds trust among users, making your app more reliable and competitive in the marketplace.

Data encryption is a pivotal component of Android security practices. It plays a vital role in protecting sensitive information within Android apps. By encrypting data, such as user credentials and personal details, developers ensure that even if unauthorized access occurs, the data remains indecipherable, adding an extra layer of defense against data breaches and enhancing the overall security of the Android application.

There are a lot of encryption systems that are typically used in android application development that include AES, RSA, ECC and many more. In this tutorial, we are going to discuss how to integrate AES Encryption System in an Android Application using Java.

Prerequisites

- Basic Idea about Android Studio
- Java

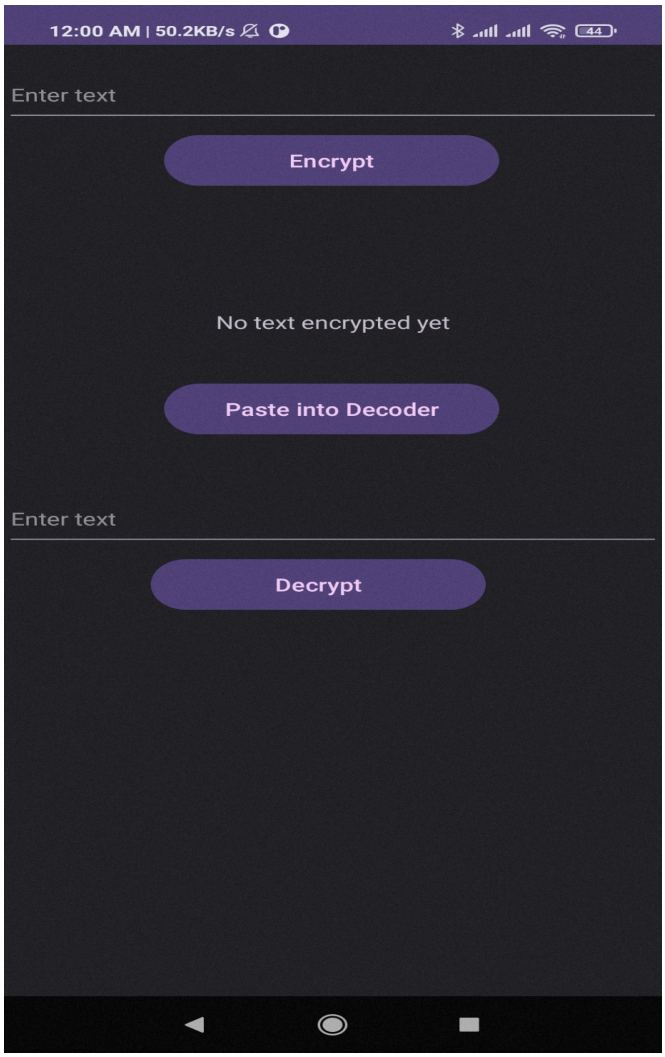
AES Encryption System

Advanced Encryption Standard (AES) is a widely adopted symmetric-key encryption algorithm used to secure data. It was established as a standard by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES uses variable key lengths of 128, 192, or 256 bits, making it highly versatile for different security requirements. The algorithm is known for its efficiency and strong encryption capabilities, making it a preferred choice for protecting sensitive information in various applications, including data transmission, storage, and secure communication. AES operates through a series of substitution, permutation, and mixing operations to ensure robust protection against

cryptographic attacks. We are not going to go into the intricacies of the system. Java provides built-in support for this system. Here, we are going to see how we can use Java's built-in support to accomplish our task.

Creating the layout

We need two EditTexts for writing the message to be encrypted, and the cipher-text to be decrypted. Two TextViews are needed to display the encrypted message (cipher-text) and the decrypted message. We also need three buttons, one for encrypting, one for copying and pasting the cipher text into the EditText for decryption, and the last one for decryption. The layout we are trying to create is:



To get the XML Code for this layout, check out the following link: https://github.com/tahsinchoudhury/AndroidSecurityAESDataEncryption/blob/main/app/src/main/res/layout/activity_main.xml

You can play around with the layout if you like.

Creating the AESEncoder Class

Now, we need to create a class called AESEncoder that will help us encode and decode messages. The Java built-in methods will be used here to create the cipher-text and decrypt it.

First, we need to generate a key for encrypting. We write a method called init to generate the key and store it in a member variable called “key”. The method is given below:

```
private SecretKey key;
1 usage  tahsinchoudhury
public void init() throws Exception {
    KeyGenerator generator = KeyGenerator.getInstance( algorithm: "AES");
    generator.init(KEY_SIZE);
    key = generator.generateKey();
}
```

This key needs to be stored because it is going to be useful later, both while encrypting and decrypting.

Now, we need to create a method to encrypt a message. Before we do that, we need a method to convert strings to byte[] arrays, since the methods that do the encryption deal

with byte arrays. Here goes the method:

```
1 usage  tahsinchoudhury
private String encode(byte[] data) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        return Base64.getEncoder().encodeToString(data);
    }
    return null;
}
```

We also need a method to convert a byte array to a string. Here it goes:

```
1 usage  tahsinchoudhury
private byte[] decode(String data) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        return Base64.getDecoder().decode(data);
    }
    return null;
}
```

Now, we are going to implement a method called “**encrypt**” that will take a string as a parameter, which is the message to be encrypted, and will return a string, which is the cipher-text.

The code is given below and described later:

```
1 usage   tahsinchoudhury
public String encrypt(String message) throws Exception {
    byte[] messageInBytes = message.getBytes();
    encryptionCipher = Cipher.getInstance( transformation: "AES/GCM/NoPadding");
    encryptionCipher.init(Cipher.ENCRYPT_MODE, key);
    byte[] encryptedBytes = encryptionCipher.doFinal(messageInBytes);
    return encode(encryptedBytes);
}
```

Note that before using the code, you have to declare a member variable called encryptionCypher, which is of type Cypher. If you are confused, just type “private Cipher encryptionCipher” after the class definition.

Let us now go through the code line by line:

1. In the first line, the String is converted to a byte array.
2. The second line creates a Cipher object named encryptionCipher for performing encryption operations. It specifies the encryption algorithm and mode. In this case, it's using AES in GCM mode with no padding. GCM is a mode of operation that provides both confidentiality and authenticity. The Cipher.getInstance method is used to obtain a cipher instance based on the specified algorithm and mode.
3. The third line initializes the encryptionCipher in encryption mode (Cipher.ENCRYPT_MODE) with a given encryption key (key), which was generated using the init method. This sets up the cipher to use the provided key for encrypting data.
4. The fourth line performs the actual encryption. It takes the messageInBytes and encrypts it using the initialized encryptionCipher. The result is stored in the encryptedBytes byte array.
5. The fifth line simply converts the byte array into a string, as required.

Next, we need to implement the “**decrypt**” method.

Before using it, we need to set two constants, as shown below:

```
1 usage
private final int KEY_SIZE = 128;
1 usage
private final int T_LEN = 128;
4 usages
```

The method is provided below:

```
1 usage  tahsinchoudhury
public String decrypt(String encryptedMessage) throws Exception {
    byte[] messageInBytes = decode(encryptedMessage);
    Cipher decryptionCipher = Cipher.getInstance( transformation: "AES/GCM/NoPadding");
    GCMParameterSpec spec = new GCMParameterSpec(T_LEN, encryptionCipher.getIV());
    decryptionCipher.init(Cipher.DECRYPT_MODE, key, spec);
    byte[] decryptedBytes = decryptionCipher.doFinal(messageInBytes);
    return new String(decryptedBytes);
}
```

Let us now go through the code:

1. The first line converts the string to a byte array.
2. The next line creates a new Cipher object named decryptionCipher for performing decryption operations. It specifies the decryption algorithm and mode, which is "AES/GCM/NoPadding," matching the encryption mode used.
3. This line creates a GCMParameterSpec object named spec. It is used to provide additional parameters required for GCM decryption, such as the "T_LEN" (the tag length) and the IV (Initialization Vector) obtained from the encryptionCipher used during encryption. The IV is necessary for GCM decryption because it is used in the GCM mode to ensure the authenticity of the ciphertext.
4. The fourth line initializes the decryptionCipher in decryption mode (Cipher.DECRYPT_MODE) using the specified key and the spec parameter. This sets up the cipher to use the provided key and IV for decrypting data.
5. This line performs the actual decryption by calling doFinal on the decryptionCipher. It takes the messageInBytes, which represents the encrypted

data, and decrypts it using the initialized decryptionCipher. The result is stored in the decryptedBytes byte array.

6. The next line converts the decryptedBytes back into a String, assuming that the decrypted data represents a textual message, and returns it as the output of the decrypt method. This line effectively transforms the binary decrypted data into its original text form.

This completes the AESEncoder class. If you are confused about anything, take a look at the entire class here:

https://github.com/tahsinchoudhury/AndroidSecurityAESDataEncryption/blob/main/app/src/main/java/com/example/dataencryptiontutorial/AESEncoder.java

Creating the MainActivity

After entering the onCreate method, we need to bind the views to variables. The code is:

```
Log.d(TAG, msg: "onCreate: initializing");
EditText editText = (EditText) findViewById(R.id.editText);
EditText cipherEditText = (EditText) findViewById(R.id.editText2);

Button encryptButton = (Button) findViewById(R.id.encryptButton);
Button pasteButton = (Button) findViewById(R.id.pasteButton);
Button decryptButton = (Button) findViewById(R.id.decryptButton);

TextView encryptedTextView = (TextView) findViewById(R.id.encryptedText);
TextView decryptedTextView = (TextView) findViewById(R.id.decryptedText);
```

When a button is pressed, we need to remove the soft keyboard from the screen. For that, the following method can be used:

```
2 usages  tahsinchoudhury
private void hideKeyboard(View view) {
    if (view != null) {
        InputMethodManager imm = (InputMethodManager)getSystemService(Context.INPUT_METHOD_SERVICE);
        imm.hideSoftInputFromWindow(view.getWindowToken(), flags: 0);
    }
}
```

In case of any error while encrypting or decrypting, we can set error message on a textview using the following method:

```
2 usages  new *
private void displayErrorMessage(TextView textView, String message) {
    textView.setText(message);
    textView.setTextColor(Color.RED);
}
```

Next, when the encrypt button is clicked on, we need to collect data from the EditText, instantiate an AESEncoder object, initialize it and call the encrypt method. Then we need to fill the text view. The code is:

```
encryptButton.setOnClickListener(view -> {
    hideKeyboard(view);
    String message = editText.getText().toString();
    try {
        encoder = new AESEncoder();
        encoder.init();
        encodedMessage = encoder.encrypt(message);
    } catch (Exception e) {
        Log.d(TAG, msg: "onCreate: Encryption error: " + e.getMessage());
        displayErrorMessage(encryptedTextView, message: "Sorry. Could not be encrypted.");
        return;
    }
    encryptedTextView.setText("Encrypted Text: " + encodedMessage);
});
```

The paste button simply puts the encoded message onto the EditText for cipher-texts:

```
pasteButton.setOnClickListener(view -> {
    cipherEditText.setText(encodedMessage);
});
```

The decrypt method is very similar to the encrypt method. So, only the code is provided:

```
decryptButton.setOnClickListener(view -> {
    hideKeyboard(view);
    String cipherText = cipherEditText.getText().toString();
    try {
        decryptedMessage = encoder.decrypt(cipherText);
    } catch (Exception e) {
        displayErrorMessage(decryptedTextView, message: "Sorry. Could not be decrypted.");
        Log.d(TAG, msg: "onCreate: Decryption error: " + e.getMessage());
        return;
    }
    decryptedTextView.setText("Original Text: " + decryptedMessage);
});
```

GitHub Link

For any kind of confusion, check out the following repository:
https://github.com/tahsinchoudhury/AndroidSecurityAESDataEncryption/tree/main

References

1. https://github.com/tugbaguler/Text-And-Image-Encryption-Decryption-App/tree/main
2. https://www.youtube.com/watch?v=J1RmZZEkN0k&t=436s
3. https://medium.com/@deepak.sirohi9188/java-aes-encryption-and-decryption-1b30c9a5d900
4. https://developer.android.com/topic/security/best-practices