Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2019, 6(2):287-292**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Android Application with Enhanced Security for Sensitive Persistent Data

Sambu Patach Arrojula

Email: sambunikhila@gmail.com

Abstract Although Android (AOSP) provides various security mechanisms for application developers, including Linux-based process isolation and full disk encryption, these measures may not fully address the entire spectrum of potential threats. Notably, once the device completes the boot process and the user is authenticated, all data is decrypted, making it susceptible to sophisticated attacks that can access application data even when the device is locked. This paper explores the design of an application aimed at enhancing the security of its sensitive data, especially in scenarios where the device is either locked or unlocked. This approach significantly mitigates the threat surface in situations where the device has booted successfully but the user is not present or the device remains locked.

Note: It is important to note that this paper focuses solely on the persistent data stored on disk, rather than data loaded into memory or shared with other applications. This persistent data is particularly vulnerable in cases where the device is stolen or lost but not rebooted.

Keywords Android, KeyStore, ContentProvider, User Authentication, SymmetricKeys, separate key-aliases for encryption and decryption

1. Introduction

Android gives enough tools for the application developer to secure his persistent sensitive data, but by default the security it provides to application's data is not sophisticated enough and application's data is still at risk when device booted successfully. So app developers need to use the tools and set up extra security for their user data if they have to aim for higher standards of security for their user-data.

Besides the above problem, applications in general don't consider their entire persistent data as sensitive and would like to set up extra security measurements only for the sensitive data (which they think) for all practical and performance purposes.

Then, this paper proposes and evaluates a design that an app can rely on for securing its users sensitive (persistent) data.

- Sensitive data encrypted even after device booted - i.e until user not present/authenticated
- Allows Writing new data into Sensitive Data
- Dynamic decryption of data - i.e. on actual read of data.

This design is only concerned about the application's persistent data and not the data that is loaded into memory or the data that is being shared with other applications or other systems like the backend. These data are of different contexts and need much more sophisticated methods and support needed from the platforms.

2. Use Case

Android offers various methods for applications to store persistent data on disk. This paper considers the use case of ContentProvider, a widely adopted and Android-endorsed component for managing and sharing user

data. Consequently, the approach discussed herein is applicable to numerous applications aiming to secure their sensitive persistent data through ContentProvider.

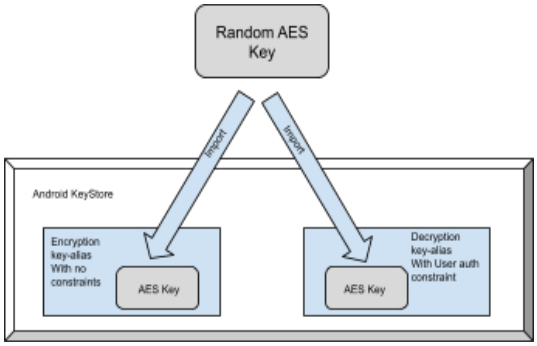
3. Approach

- The design goal here is to secure app-selected sensitive persistent data even after device boot and make it accessible only when the user is present precisely, per access user authentication.
- We rely on AndroidKeyStore for creating and maintaining the crypto keys.
- Application will create two symmetric key-entries. one for encryption and without any access constraints and other for decryption with user-authentication constraint.
- Application will use ContentProvider and pick its own column(s) as its sensitive data and seamlessly encrypts and decrypts them. And the clients of that ContentProvider are agnostic to this process except they are supposed to reach device owners to authenticate whenever client-apps access the sensitive data. It's like a Financial app asks the user to authenticate bef accessing his financial info.

4. Details

A. Setting up Cryptographic keys

The Android Keystore is a hardware-backed trusted execution environment that we utilize to generate and manage cryptographic keys. When the application handling sensitive data is launched for the first time, it generates a random AES key outside of the Keystore, which is then imported into the Android Keystore und two separate aliases. The first alias, without any access restrictions, is used for encryption operations. The second alias, which requires user authentication, is designated for decryption operations. From that point onward, the application uses these keys for secure operations on its sensitive data. This key setup allows the encryption key alias to be accessible even when the device is locked, enabling the writing of sensitive data However, decryption is only possible when the user authenticates, due to the access restrictions we applied the decryption key alias.



We have several other options for key setup, such as using RSA key pairs for encryption and decryption. However, RSA cryptographic operations are more computationally expensive than AES. Another approach involves using an intermediate AES key for actual encryption and decryption, with this master key being secured by the AndroidKeyStore. This method offers performance benefits by offloading the trusted execution but introduces higher risks as the actual key resides in the regular execution environment.

Another crucial point is the destruction of the non-keystore AES key after importing it into the keystore. Currently, there seems to be no reliable method to achieve this, though we hope future Android versions will support it. For now, we proceed with the existing setup.

B. ContentProvider that deal with sensitive data

Now that the cryptographic keys are set up, the ContentProvider can utilize them for encryption and decryption. When the ContentProvider receives new data for insertion or update, it checks whether the content contains any sensitive data. The application defines what constitutes sensitive data by specifying certain columns. If the incoming data includes these specified sensitive columns, the ContentProvider will encrypt the data before storing it in the actual storage (such as a database).

```
public static void getKeyGenerator() { try {
        KeyGenerator keyGenerator =
KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES); keyGenerator.init(128);
        SecretKey secretKey = keyGenerator.generateKey();
        KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore"); keyStore.load(null);

        if (!keyStore.containsAlias(ENCRYPT_KEY_ALIAS) ) { keyStore.setEntry(
                ENCRYPT_KEY_ALIAS,
                new KeyStore.SecretKeyEntry(secretKey), new
                KeyProtection.Builder(KeyProperties.PURPOSE_ENCRYPT)
                        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
                    .build());
        }else {
           Log.w(TAG,"ENCRYPT_KEY already exists");
        } if( !keyStore.containsAlias(DECRYPT_KEY_ALIAS)) { keyStore.setEntry(
                DECRYPT_KEY_ALIAS,
                new KeyStore.SecretKeyEntry(secretKey), new
                KeyProtection.Builder(KeyProperties.PURPOSE_DECRYPT)
                        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
                    .setUserAuthenticationRequired(true) .build());
        }else{
           Log.w(TAG,"DECRYPT_KEY already exists"); }
      } catch (NoSuchAlgorithmException e) {
        Log.e(TAG,"Exception while creating KeyGenerator :"+e.getMessage());
      } catch (CertificateException e) { throw new RuntimeException(e);
      } catch (KeyStoreException e) { throw new RuntimeException(e);
      } catch (IOException e) { throw new RuntimeException(e); }
    }
```

Here is an example for encrypting the text column TASK in the insert() flow.

```
public static SecretKey getEncryptKey() { try {
            KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
            keyStore.load(null);

            KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.SecretKeyEntry)
keyStore.getEntry(ENCRYPT_KEY_ALIAS, null); return secretKeyEntry.getSecretKey();
        } catch (Exception e) {
            Log.e(TAG,"Exception while getKey() :"+e.getMessage()); e.printStackTrace();
            return null;
        }
    }
    public static SecretKey getDecryptKey() { try {
            KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
            keyStore.load(null);
            KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.SecretKeyEntry)
keyStore.getEntry(DECRYPT_KEY_ALIAS, null); return secretKeyEntry.getSecretKey();
        } catch (Exception e) {
            Log.e(TAG,"Exception while getKey() :"+e.getMessage()); e.printStackTrace();
            return null;
        }
    }
    public static String getEncryptedDataAES(String data) { try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
            cipher.init(Cipher.ENCRYPT_MODE, getEncryptKey()); byte[] iv = cipher.getIV();
            byte[] encryptedData =
cipher.doFinal(data.getBytes(java.nio.charset.StandardCharsets.UTF_8));

            return Base64.encodeToString(Bytes.concat(encryptedData,iv), Base64.DEFAUL
        } catch (Exception e) {
            Log.e(TAG,"Exception while getEncryptedDataAES() :"+e.getMessage());
            e.printStackTrace(); return null;
        }
    } public static String decryptData( String codedDataStr) { try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding"); byte[] codedData
            = Base64.decode(codedDataStr, Base64.DEFAULT); byte[] encryptedData = new
            byte[codedData.length - 16]; byte[] iv2 = new byte[16];
            System.arraycopy(codedData, 0, encryptedData, 0, encryptedData.length);
            System.arraycopy(codedData, encryptedData.length, iv2, 0, iv2.length);

            //int ivIndex = codedData.length - 16;
            //IvParameterSpec paramSpec = new IvParameterSpec(codedData, ivIndex, 16)
            IvParameterSpec paramSpec = new IvParameterSpec(iv2);
            cipher.init(Cipher.DECRYPT_MODE, getDecryptKey(), paramSpec);
            byte[] decryptedData = cipher.doFinal(encryptedData);
            return new String(decryptedData, java.nio.charset.StandardCharsets.UTF_8);
        } catch (Exception e) {
            Log.e(TAG,"Exception while decryptData() :"+e.getMessage());
            e.printStackTrace(); return null;
        }
    }
```

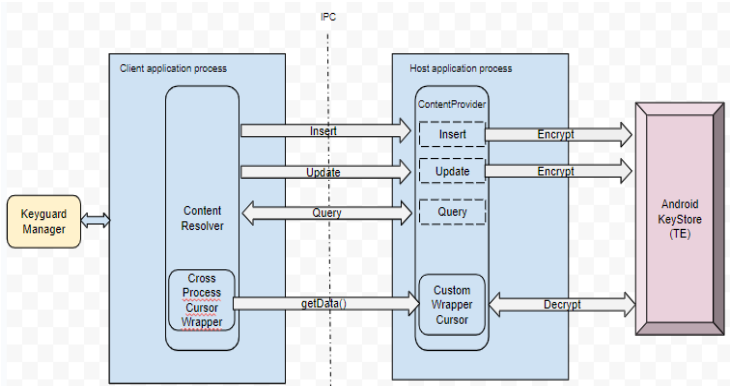
```
@Override

public Uri insert(Uri uri, ContentValues contentValues) {

      if (uriMatcher.match(uri) != TaskContract.TASKS_LIST) { throw new
         IllegalArgumentException("Invalid URI: "+uri);
      }
      String data = contentValues.getAsString( TaskContract.Columns.TASK); // sensiti
if(data != null){ contentValues.put(TaskContract.Columns.TASK,
AESHelper.getEncryptedDataAES(data) );
      }
      long id = db.insert(TaskContract.TABLE,null,contentValues);

      if (id>0) { return ContentUris.withAppendedId(uri, id);
      }
      throw new SQLException("Error inserting into table: "+TaskContract.TABLE); }
```

In the query() function, we wrap the final cursor with our CustomWrapperCursor and return it. On the client side, the Android framework (via ContentResolver, etc.)



will wrap our returned cursor in a CrossProcessCursorWrapper before passing it to the client application. This ensures that all calls made by the client application on the final cursors are seamlessly executed as IPC (Inter-Process Communication) calls onto our cursor object in ContentProvider process where our CustomWrapperCursor will handle decryption of sensitive data.

```
@Override public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionAr
String sortOrder) {
      SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
      qb.setTables(TaskContract.TABLE);

      switch (uriMatcher.match(uri)) { case TaskContract.TASKS_LIST: break;

        case TaskContract.TASKS_ITEM:
          qb.appendWhere(TaskContract.Columns._ID + " = "+
          uri.getLastPathSegmen break;

        default:
              throw new IllegalArgumentException("Invalid URI: " + uri);
      }

      Cursor cursor = qb.query(db,projection,selection,selectionArgs,null,null,null);

      //return new cursor;
      return new CustomWrapperCursor(cursor); }
```

C. Client Integration

Now Client integration will be similar steps/process as of standard ContentProvider integration, with the exception that the client will be aware of the specific columns containing sensitive data as defined in the contract/interface and consequently, whenever the client attempts to access these sensitive columns, it will prompt the user for authentication like before calling the query() function.

```
private static final int REQUEST_CODE_QUERY = 1; private void
    showAuthenticationScreen() {
        KeyguardManager mKeyguardManager = (KeyguardManager)
getSystemService(Context.KEYGUARD_SERVICE);
        Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null,
        nu if (intent != null) { startActivityForResult(intent, REQUEST_CODE_QUERY
        );
        }
    }

    @Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) { if
(requestCode == REQUEST_CODE_QUERY ) { // Challenge completed, proceed
with using cipher if (resultCode == RESULT_OK) {
this.getContentResolver().query(TaskContract.CONTENT_URI,null,null,null,null);
            } else {
                Log.e(TAG, "Authentication failed."); }
        }
    }
```

5. Conclusions

While Android provides foundational security mechanisms for protecting application data, the default measures are insufficient for securing sensitive data once the device has completed the boot process. The proposed approach addresses this gap by leveraging the AndroidKeyStore for robust key management and implementing user authentication constraints for decrypting sensitive data. By integrating this within the ContentProvider framework, applications can seamlessly encrypt and decrypt sensitive data columns, ensuring that such data remains protected even after the device has booted.

This design not only enhances the security of persistent data but also maintains usability and performance by allowing applications to selectively secure only the most critical information. Future work may extend this approach to cover additional threat vectors and explore its integration with broader security frameworks. Overall, this system offers a practical and effective solution for developers aiming to meet higher security standards for their applications' sensitive data.

6. Next Steps

- Proper destruction of the random secret key, because that's the actual/root key using which the master encryption/decryption keys can be derived easily.
- User authentication can be handled by ContentProvider itself as actual decryption is being done there. This should help client applications not to worry about user authentication. but has to be evaluated further if it can break any other usecases.

References

[1]. Android application security.
[2]. Android full-disk encryption
[3]. Vulnerabilities of android(aosp) disk encryption
[4]. disk encryption helps when device turned-off
[5]. Securing key access with user present/authentication https://cypherpunk.nl/papers/spsm14.pdf
[6]. Secure Key Storage and Secure Computation in Android https://www.cs.ru.nl/masters-theses/2014/T_Cooijmans___Secure_key_storage_a secure_computation_in_Android.pdf
[7]. Nikolay Elenkov. Accessing the embedded secure element in Android 4.x. Aug.2012. http://nelenkov.blogspot.nl/2012/08/accessing-embedded-secureelement-in.html.

[8]. Nikolay Elenkov. Jelly Bean hardware-backed credential storage. http://nelenkov.blogspot.nl/2012/07/jelly-bean-hardware-backed-credential. html