Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2019, 6(11):94-103

Research Article ISSN: 2394 - 658X

An Android Framework for Seamless Enhanced security of Sensitive Persistent Data

Sambu Patach Arrojula

email: sambunikhila@gmail.com

ABSTRACT

Although Android (AOSP) offers various security mechanisms for application developers to protect their persistent data, such as Linux-based process isolation and full disk encryption, these measures may not comprehensively address the spectrum of potential threats. Notably, once the device completes the boot process and the user is authenticated, all data is decrypted, making it vulnerable to sophisticated attacks that can access application data even when the device is locked. While AOSP provides some cryptographic options for such cases, not all applications can implement these changes. For instance, pre-loaded apps often avoid modifications due to a rigorous approval process, and integrating OEM-specific solutions can be challenging. This paper explores an Android system/framework that offers a seamless method for enhancing the security of sensitive persistent data not only for first and second party apps but even third party applications. Applications requiring protection for their sensitive persistent data can utilize this service without changes for their application. This framework is specifically designed to address scenarios where the device transitions between locked and unlocked states while handling sensitive data, providing solutions for applications and when integrated an application can significantly reduce the threat surface for its sensitive data in situations where the device has booted successfully but the user is not present or the device remains locked.

Note: This paper focuses exclusively on persistent data stored on disk, not on data loaded into memory or shared with other applications. This persistent data is particularly vulnerable if the device is stolen or lost but not rebooted.

Keywords: Android, KeyStore, Android framework, seamless solutions, OEM specific cryptography, pre-loaded apps, ContentProvider, User Authentication, SymmetricKeys, separate key-aliases for encryption and decryption

---

INTRODUCTION

Android offers a variety of tools for application developers to secure persistent sensitive data. However, the default security measures are not sufficiently sophisticated, leaving application data vulnerable when the device is lost or stolen after successfully booting and the user has been verified. Consequently, developers must utilize these tools and implement additional security measures to achieve higher standards of data protection.

Although AOSP provides some cryptographic options for such cases, not all applications can accommodate these changes due to various constraints. For instance, pre-loaded apps often resist modifying their code because of the stringent approval processes they must undergo. This rigorous approval procedure ensures that any changes meet specific standards and criteria, which can be time-consuming and complex. Additionally, if there is an OEM-specific solution in place, integrating it becomes even more challenging. OEM-specific solutions often involve proprietary technologies and custom implementations that require significant effort and expertise to adapt, making it more difficult for developers to integrate such solutions into their applications seamlessly. In this paper, we discuss and explore an approach in which an Android Framework (e.g., a fork of AOSP) can provide a similar solution while not only abstracting the cryptographic details from application developers but also provides the service without changing neither the data provider app nor its client/consumer app.

USE CASE

Android offers numerous methods for applications to store persistent data in turn, applications have a number of use cases and scenarios in this context. following are some use cases we are considering in this paper

• use case of ContentProvider which is popular and endorsed by Android for user data and also for sharing such data hence this approach we discuss here should be feasible for many applications in order to secure their sensitive persistent data.

• We consider Android framework that an app can rely on where the framework is designed to provide a seamless protection of sensitive data in a Contetnt Provider that any interested application can opt for without changing their app which would not expect any change for their client applications as well.

• In this paper we are assuming that applications would be able to do off-the-band opt-in for this service and the framework has the knowledge about which applications have opted-in and what their corresponding sensitive columns are. There are various ways a framework can accomplish this but not the subject of this paper.

• For now in this paper, we consider only pre-opt-in cases for an app where the app-developer would enroll into such service before the first usage of the app. i.e. we are not considering migration of existing app data. which should be doable when we have the core functionality in place but just not a subject of this paper.

• We also consider AndroidKeyStore as an implementation cryptography to demonstrate the concept but which can be upgraded to any OEM/framework specific cryptographic solutions provided they offer such services discussed here.

• We considered Android 7.1.2 for our experimentation to prove the concept.

APPROACH

• The design goal here is to have an android framework that provides a seamless secure solution to its applications when opt-in these applications could enhance security of their sensitive data without changing their code. and this f/w solution is to secure an app-selected sensitive persistent data even after device boot and make it accessible only when device is unlocked and more precisely with user authentication (per access).

• The main approach here is to intercept ContentProvider calls in framework and if that is a sensitive ContentProvider divert that call to framework owned component where it will take care of

- Cryptographic details for encryption and decryption of sensitive data
- Authenticating user when needed i.e. when decrypting sensitive data

• Framework components will rely on AndroidKeyStore or OEM specific cryptographic solutions for creating and maintaining the crypto keys.

• Effectively, the framework will create and maintain two symmetric key-entries per sensitive-ContentProvider. one for encryption and without any access constraints and other for decryption with user-authentication constraint. (assuming framework have such cryptographic solution with these features)

• applications, neither the sensitive ContentProvider nor the user application of that sensitive ContentProvider will not integrate with the framework solution. i.e. no code change is needed for those applications. Rather sensitive ContentProvider app developer would enroll into such service off-the-band and pick its own column(s) as its sensitive data and framework seamlessly sync that info into devices and will take care of securing them. And the clients of that ContentProvider are agnostic to this process framework will take care of authenticating the user when necessary i.e. before decrypting sensitive data.

DETAILS

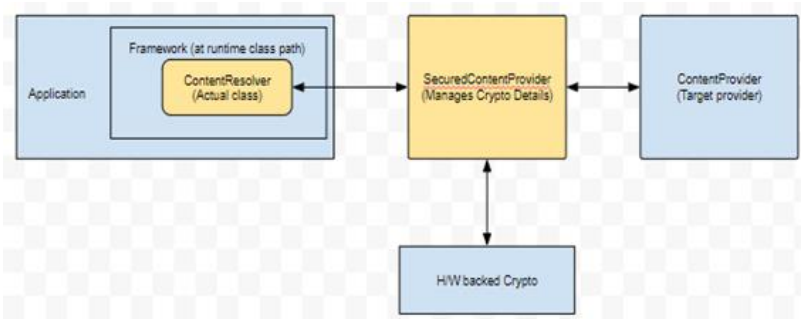
A. Enroll into service

An application developer can opt-in to this service off-the-band. There are various ways to do this, for example hardcode the package names & their sensitive columns into the framework and upgrade the list through OTA updates. Another way is to have secured remote APIs. For this paper, we are assuming that the framework has these sensitive packages & their sensitive columns information readily available. We are also not considering the migration between regular and sensitive data which would be the case when applications enroll and unroll from this service dynamically, which again is not the case we are discussing in this paper.

B. Access to the service

Applications dont have to worry about how to access this service, they just develop a regular ContetnProvider app as per regular Android SDK. The main approach we follow for this solution is to modify the android framework, which will be in the runtime classpath of every application. As the framework has the knowledge of which content provider (authority) is sensitive it can intercept any applications calls to those providers. We listen to these calls in ContentResolver in the android framework.

ContentResolver in the case of sensitive provider access, will divert the call to a dedicated SecuredContentProvider which is part of the system and whitelisted to get access to any other provider in the system.



ContentResolver changes:

```
public final @Nullable Cursor query(final @RequiresPermission.Read @NonNull
Uri uri,
        @Nullable String[] projection, @Nullable String selection,
        @Nullable String[] selectionArgs, @Nullable String sortOrder,
        @Nullable CancellationSignal cancellationSignal) {
      Preconditions.checkNotNull(uri, "uri");
      IContentProvider unstableProvider = acquireUnstableProvider(uri);
      if (unstableProvider == null) {
        return null;
      }
// as a basic measure, make sure calling app has access permission, should be
able to get Provider
     if(mSensitiveAuths.contain(uri.getAuthority()){
       // append actual uri to sensitive provider uri and divert query() to sensitive
provider
        Uri.parse("content://com.example.sensitive.provider/process#" +
              Base64.encodeToString(uri.toString.getBytes(StandardCharSets.UTF_8),
Base64.DEFAULT );
          unstableProvider = acquireUnstableProvider(uri);

     }
```

```
public final @Nullable Uri insert(@RequiresPermission.Write @NonNull Uri url,
          @Nullable ContentValues values) {
      Preconditions.checkNotNull(url, "url");
      IContentProvider provider = acquireProvider(url);
      if (provider == null) {
        throw new IllegalArgumentException("Unknown URL " + url);
      }
// as a basic measure, make sure calling app has access permission, should be
able to get Provider
     if(mSensitiveAuths.contain(uri.getAuthority()){
       // append actual uri to sensitive provider uri and divert insert() to sensitive
provider
        Uri.parse("content://com.example.sensitive.provider/process#" +
              Base64.encodeToString(uri.toString.getBytes(StandardCharSets.UTF_8),
Base64.DEFAULT );
          provider = acquireProvider(uri);

     }
```

C. Secured Content Provider

SecureContentProvider, is a system owned ContetnProvider (doesn't necessarily have SYSTEM UID) which is a secured and whitelisted process in the system which will be able to access any content provider installed in the system. i.e. even if a Content Provider requires some specific permissions to be granted by the User. As a security measure SecuredContentProvider verifies calling applications access rights to the target provider before it actually reaches it. System will be hard coded with package-name:signature so that it can verify the authenticity of the package existing before adding that package into the whitelist. This signature based white listing not only improves security but also lets the system late-install the white listed packages.

aquireProvider() will eventually check with ActivityManagerService where the permissions will be verified, here we will check the white list and grant access for SecuredContentProvider to access any provider.

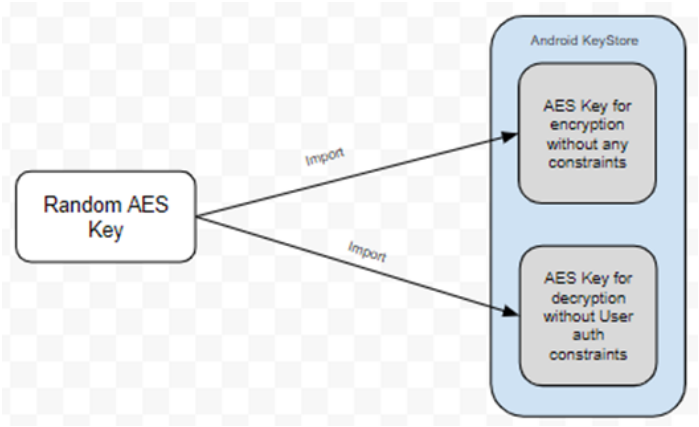
```
// ActivityManagerService.java
private final String checkContentProviderPermissionLocked(
        ProviderInfo cpi, ProcessRecord r, int userId, boolean checkUser) {
    final int callingPid = (r != null) ? r.pid : Binder.getCallingPid();
    final int callingUid = (r != null) ? r.uid : Binder.getCallingUid();
// Check if calling package is sensitive provider package if so grant permission
    String checkPkg = mContext.getPackageManager().getNameForUid(callingUid);
    if(mWhitelistPkgnames.contains(checkPkg){
      return null; // grant permission
    }
 ...
}
```

Later system will check target ContentProvider itself for READ and WRITE permission checking, i.e. ContentProvider base class in android framework in the target provider process will verify the access permission of the calling process. Here again we check if the calling process is whitelisted and grant access for SecuredContentProvider.

```
// ContentProvider/java
protected int enforceReadPermissionInner(Uri uri, String callingPkg, IBinder callerToken)
        throws SecurityException {
    final Context context = getContext();
    final int pid = Binder.getCallingPid();
    final int uid = Binder.getCallingUid();
// Check if calling package is sensitive provider package if so grant permission
    String checkPkg = getContext().getPackageManager().getNameForUid(uid);
    if(mWhitelistPkgnames.contains(checkPkg){
      return MODE_ALLOWED; // grant permission
    }
 ...
}

protected int enforceWritePermissionInner(Uri uri, String callingPkg, IBinder callerToken)
        throws SecurityException {
    final Context context = getContext();
    final int pid = Binder.getCallingPid();
    final int uid = Binder.getCallingUid();
// Check if calling package is sensitive provider package if so grant permission
    String checkPkg = getContext().getPackageManager().getNameForUid(uid);
    if(mWhitelistPkgnames.contains(checkPkg){
      return MODE_ALLOWED; // grant permission
    }
 ...
}
```

D. Setting up Cryptographic keys

Frameworks and OEMs can offer various cryptographic services, including hardware-backed solutions. In this paper, we focus on the Android Keystore, an Android-exposed, hardware-backed trusted execution environment. SecureContentProvider utilizes the Android Keystore to generate and maintain cryptographic keys. When a sensitive content provider being accessed for the first time, SecureContentProvider generates a random AES key outside the Keystore and then imports it into the Android Keystore under two separate aliases and these two key-aliases will be maintained by SecureContentProvider per sensitive ContentProvider. Among those two aliases the first alias would be, without access restrictions, and used for encryption operations, while the second alias would be, with user authentication access restrictions, and used for decryption operations. From that point on, the application uses these keys for secure operations on its sensitive data.



We have some other options for the key setup like using RSA key pairs one for encryption and decryption but RSA cryptographic operations are costlier than AES. Then another option is to have an intermediate AES key for actual encryption and decryption and this master key can be secured by AndroidKeyStore - while this gives a performance gain as it refuses the trusted execution but opens up high risk because the actual key is in a real execution environment.

Another point to consider here is to destroy the non-keystore AES key after we import it into the keystore. but we keep this out of the scope of this paper for now.

```
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);
    }
  }
```

E. SecureContentProvider that deal with sensitive data

Now that we have the crypto keys set up, the SensitveContentProvider can use them to encrypt and decrypt. First whenever it gets new data for insert/update etc SecureContetnProvider will encrypt the data first then submit

updated row to actual target ContentProvider for insert. Target uri can be extracted from the provided uri from the client application.
Here is an example for insert() flow encrypting a text column defined by subclass ContentProvider in.

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

            KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.Secret-
KeyEntry) keyStore.getEntry(ENCRYPT_KEY_ALIAS+authority, null);
            return secretKeyEntry.getSecretKey();
        } catch (Exception e) {
            Log.e(TAG,"Exception while getKey() :"+e.getMessage());
            e.printStackTrace();
            return null;
        }
    }
    public static SecretKey getDecryptKey(String authority) {
        try {
            KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
            keyStore.load(null);

            KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.Secret-
KeyEntry) keyStore.getEntry(DECRYPT_KEY_ALIAS+authority, 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, String authority)
{
        try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
            cipher.init(Cipher.ENCRYPT_MODE, getEncryptKey(authority));
            byte[] iv = cipher.getIV();
            byte[] encryptedData = cipher.doFinal(data.get-
Bytes(java.nio.charset.StandardCharsets.UTF_8));

            return Base64.encodeToString(Bytes.concat(encryptedData,iv),
Base64.DEFAULT);
        } catch (Exception e) {
            Log.e(TAG,"Exception while getEncryptedDataAES() :"+e.get-
Message());
            e.printStackTrace();
            return null;
        }
    }
 public static String decryptData( String codedDataStr, String authority) {
        try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
            byte[] codedData = Base64.decode(codedDataStr, Base64.DE-
FAULT);
            byte[] encryptedData = new byte[codedData.length - 16];
            byte[] iv2 = new byte[16];
            System.arraycopy(codedData, 0, encryptedData, 0, encrypt-
edData.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(authority),
paramSpec);
            byte[] decryptedData = cipher.doFinal(encryptedData);
            return new String(decryptedData, java.nio.charset.Standard-
Charsets.UTF_8);
        } catch (Exception e) {
            Log.e(TAG,"Exception while decryptData() :"+e.getMessage());
            e.printStackTrace();
            return null;
        }
    }
```

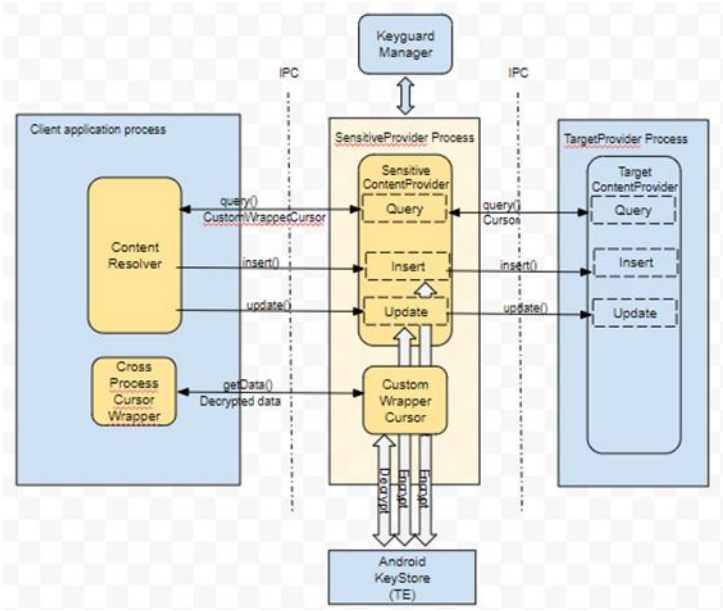
```
private Uri getTargetUri(Uri uri){
    String targetUriString = uri.getFragment();
    if(targetUriString != null) {
      Log.d(TAG, "queryImpl:targetUriString64:" + targetUriString);
      targetUriString = new String((Base64.decode(targetUriString,
Base64.DEFAULT)), StandardCharsets.UTF_8);
      Log.d(TAG, "queryImpl:targetUriString:" + targetUriString);
      return Uri.parse(targetUriString);
    }
    return uri;
  }
```

```
@Override // SecureContentProvider - assume sensitive column info available in
mSensitiveColumn
  final public Uri insert(Uri uri, ContentValues contentValues) {
    String data = contentValues.getAsString( mSensitiveColumn);
    if(data != null){
      contentValues.put(mSensitiveColumn,getEncryptedDataAES(data,
uri.getAuthority()) );
    }
    uri = getTargetUri(uri);

    return getContext().getContentResolver().insert(uri,contentValues);

  }
```

Now for reading flow, SecureContentProvider will get the actual target url first, then calls query() function on target ContentProvider where it will fetch the requested data into a Cursor as usual. Then SecureContentProvider relies on a CustomCursor which is a CursorWrapper (which itself is a wrapper for an actual Cursor object) and wraps the above Cursor result/object that the target ContentProvider returns. This CustomCursor overrides needful functions to inspect if the fetching column is of sensitive one then it will decrypt and return. CustomWrapperCursor also handles user authentication before decrypting the sensitive data as shown below. Though we have just considered basic usecase to prove the concept here we could do proper concurrency handling to prevent multiple keyguard launches when multiple read happens in parallel.

```
public class CustomWrapperCursor extends CursorWrapper {
  private KeyPair keyPairMapRSA;
  private static String TAG = "CustomWrapperCursor";
  public CustomWrapperCursor(Cursor cursor) {
    super(cursor);
    AESHelperNew.getKeyGenerator();
    keyPairMapRSA = RSAHelper.generateKeyPair();
  }


  @Override
// assume such info available in mSensitiveColumn & mAuth & mContext
  public String getString(int i) {
    if(mSensitiveColumn.equals(getColumnName(i))){
      CountDownLatch latch = new CountDownLatch(1);
    AuthActivity.LatchHolder.latch = latch;
    mContext.startActivity(new Intent(mContext,
AuthActivity.class).addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP));
    try {
      latch.await();
    } catch (InterruptedException e) {
      Log.w(TAG,"Exception while waiting on latch");
      e.printStackTrace();
      throw new RuntimeException(e);
    }

      String data = super.getString(i);
      return decryptData(data, mAuth);
    }else {
      return super.getString(i);
    }
  }
}
```

```
public class AuthActivity extends Activity {
  private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1;
  public static class LatchHolder{
    public static CountDownLatch latch;
  }
  private void showAuthenticationScreen() {
    KeyguardManager mKeyguardManager = (KeyguardManager)
getSystemService(Context.KEYGUARD_SERVICE);
    Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null,
null);
    if (intent != null) {
      startActivityForResult(intent,
REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
    }
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS) {
      //Challenge completed, proceed with using cipher

      if (resultCode == RESULT_OK) {
        Log.d(TAG, "Authentication SUCCESS.");
      } else {
        Log.e(TAG, "Authentication failed.");
      }
      LatchHolder.latch.countDown();
      finish();
    }
  }
  @Override
  protected void onResume() {
    Log.d(TAG,"onResume()");
    showAuthenticationScreen();
    super.onResume();
  }
}
```



When a client app calls query() its Android framework (ContentResolver etc) will wrap our returned cursor again in CrossProcessCursorWrapper before passing it to the client application. so that all the calls client applications make on that final cursor will seamlessly be performed as IPC calls onto our cursor object in the ContentProvider process where our CustomWrapper cursor will handle decryption of sensitive data.

```
@Override  // SecureContentProvider
   final public Cursor query(Uri uri, String[] strings, String s, String[] strings1, String s1)
{
      uri = getTargetUri(uri);
      return new
CustomWrapperCursor(getContext().getContentResolver().query(uri,strings,s,
strings1, s1));
   }
```

Even when a background client tries to access the sensitive provider, SecuredContentProvider would be able to throw a notification indicating the so-called-client-app is trying to access the so-called-provider and ask him to Authenticate. Which could launch above AuthActivity to authenticate the user.

```
private void sendNotification(String title, String clientPkg, String targetPkg) {


      Intent intent = new Intent(getContext(), AuthActivity.class);
      intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
      PendingIntent pendingIntent = PendingIntent.getActivity(getContext(), 0 /*
Request code */, intent, PendingIntent.FLAG_ONE_SHOT);
      Uri defaultSoundUri =
RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
      NotificationCompat.Builder notificationBuilder = new
NotificationCompat.Builder(getContext())
            .setSmallIcon(R.drawable.ic_launcher)
            .setContentTitle(title)
            .setContentText(clinetPkg + "Accessing" + targetPkg)
            .setAutoCancel(true)
            .setSound(defaultSoundUri)
            .setContentIntent(pendingIntent);

      NotificationManager notificationManager = (NotificationManager)
getContext().getSystemService(Context.NOTIFICATION_SERVICE);

      notificationManager.notify(0 /* ID of notification */,
notificationBuilder.build());
   }
```

F. Client and Target ContentProvider Integration

Neither Target ContentProvider nor Client applications are expected to change code for integrating with this solution. The enrollment of Target Content Provider as sensitive data will be done off the band and the system has various methods to sync that information into it.

CONCLUSION

This paper presents an Android framework designed to enhance the security of sensitive persistent data within applications, particularly focusing on data stored on disk. By leveraging the proposed framework, application developers can protect their data from sophisticated attacks that exploit the device's state transitions between locked and unlocked conditions. This is achieved without requiring any modifications to the application code, making it an attractive solution for developers who seek to enhance their security measures without significant redevelopment efforts. The framework intercepts ContentProvider calls, handles cryptographic details using the AndroidKeyStore or OEM-specific solutions, and ensures that decryption occurs only when the device is unlocked and the user is authenticated.

The implementation of this framework addresses a critical security gap in the current Android ecosystem, where existing measures may fall short in protecting data after the device has successfully booted and the user is verified. By providing a seamless method for securing sensitive data, this framework not only enhances the security posture of applications but also simplifies the process for developers. The use of symmetric keys with separate key aliases for encryption and decryption, coupled with user authentication requirements for decryption, significantly reduces the threat surface. This solution, tested on Android 7.1.2, demonstrates the feasibility and effectiveness of the approach, offering a robust security enhancement for applications handling sensitive data.

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.
- Handling of existing applications and their user data in case that application onboards this enhanced security service

● improve security measures in SecureContentProvider: instead of checking the calling client permissions in android framework, its more secure to check again in the SecureContentProvider process as well.
● Handle parallel calls for accessing sensitive data and make sure only one call should ask the user for authentication.

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_and_secure_computation_in_Android.pdf
[7]. 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. Jellybean hardware-backed credential storage. http://nelenkov.blogspot.nl/2012/07/jelly-bean-hardware-backed-credential. html