International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

AN ANDROID FRAMEWORK FOR ENHANCING THE SECURITY OF SENSITIVE PERSISTENT DATA

Sambu Patach Arrojula
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 cryptographic options are available for such situations, not all developers are familiar or comfortable with complex cryptographic details, and there may be OEM-specific cryptographic solutions unknown to standard Android developers. This paper explores an Android system/framework that offers a simple and easy interface for enhancing the security of sensitive persistent data within applications. Applications requiring protection for their sensitive persistent data can utilize this service with minimal integration effort. 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. 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, Key-Store, Android framework, Content Provider, User Authentication, Symmetric Keys, separate key-aliases for encryption and decryption

I. 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.

Though there are options/provisions where application developer can use and integrate with certain services from AOSP android to protect his sensitive persistent data, it requires significantly different skillset than usual android development such as design and handling of cryptographic keys, integration with Android Key Store and handling encryption and decryption processes. Additionally, some OEM frameworks/systems may provide additional cryptographic features, such as non-Android Key Store hardware-backed cryptography, which are not available through the Android SDK and application developers cannot access through regular Android f/w or SDK. In this paper, we discuss and explore an approach in which an Android Framework (e.g., a fork of AOSP) can provide a solution while abstracting the cryptographic details from application developers

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

II. 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 Content Provider 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 expose a generic interface for protection of sensitive data in a Content Provider that any interested application can utilize.
- In this paper, we primarily focus on First-party and Second-party applications, as the approach we discuss here is most suitable for these types of apps. While the approach can also be applied to Third-party apps, these developers may have less incentive to adopt platform-specific solutions compared to First and Second-party developers.
- We also consider Android Key Store 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

III. APPROACH

- The design goal here is to have an android framework to expose a generic interface/solution to its applications using which they could enhance security of their sensitive data. 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).
- Framework will rely on Android Key Store or OEM specific cryptographic solutions for creating and maintaining the crypto keys.
- Effectively, the framework will create two symmetric key-entries per Content Provider. 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)
- Application will integrate with the framework solution and pick its own column(s) as its sensitive data and framework seamlessly encrypts and decrypts them. And the clients of that Content Provider 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 before accessing his financial info.

IV. DETAILS

1. Access to the solution

Here we are proposing two ways framework can expose its solutions/classes to application

Through android framework and custom SDK

One can implement the solution in android framework and build a custom android SDK and can expose that android SDK to first and second party apps. Applications will develop using that custom android SDK where they get compiled with our stubbed solution and in runtime get linked with actual implementation of the solution.

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

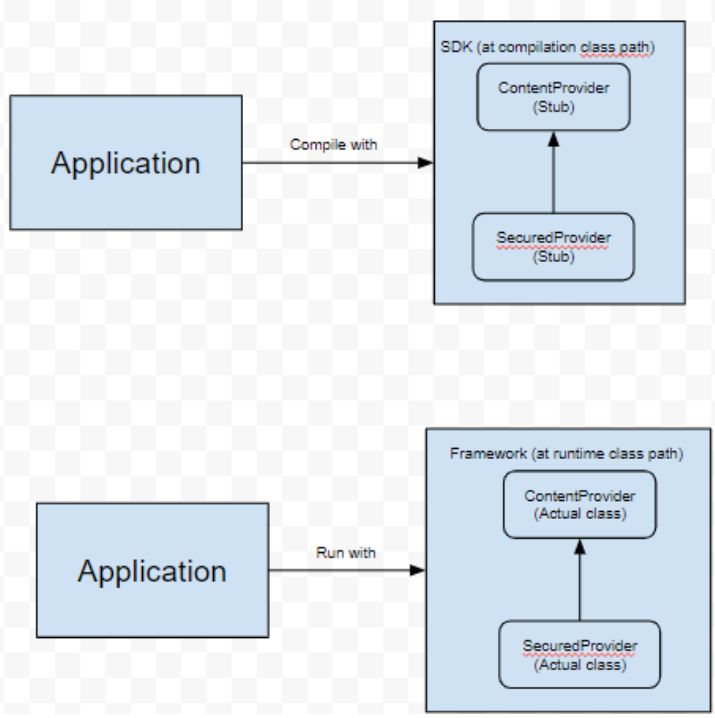


Illustration of approach

While this is a complicated approach it provides flexibility for the framework to have platform specific implementation and applications which are integrated already don't have to worry about this and would get the same service as expected.

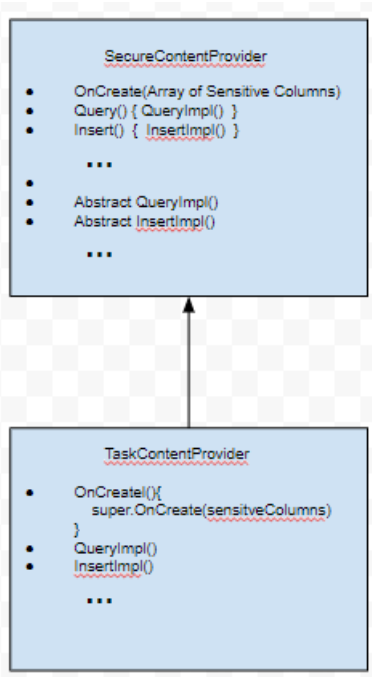
As a support library

As a simple and straight approach, we can export the solution as a library and application can compile and package with it. This approach provides flexibility for deployment of solutions but applications have to be compiled every time the library changes. Here in this paper we demonstrate the support library approach.

2. *Interface of the solution*

To minimize integration complexity for applications, we propose abstracting cryptographic details into a superclass and allow the subclass to declare its sensitive column(s), Secure Content Provider super class handles encryption and decryption processes. However, the subclass method calls must be intercepted in a specific order. For instance, the insert () call should first be intercepted by the superclass to encrypt incoming data, while the query () call should be intercepted by the superclass after the subclass to facilitate data decryption. To achieve this, we rely on template methods in the superclass and enforce the implementation of corresponding methods by the subclass.

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510



Class diagram of Content Provider

3. *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. Secure Content Provider utilizes the Android Keystore to generate and maintain cryptographic keys. When a sensitive data owner application starts for the first time, Secure Content Provider generates a random AES key outside the Keystore and then imports it into the Android Keystore under two separate aliases. The first alias, without access restrictions, is used for encryption operations, while the second alias, with user authentication access restrictions, is used for decryption operations. From that point on, the application uses these keys for secure operations on its sensitive data.

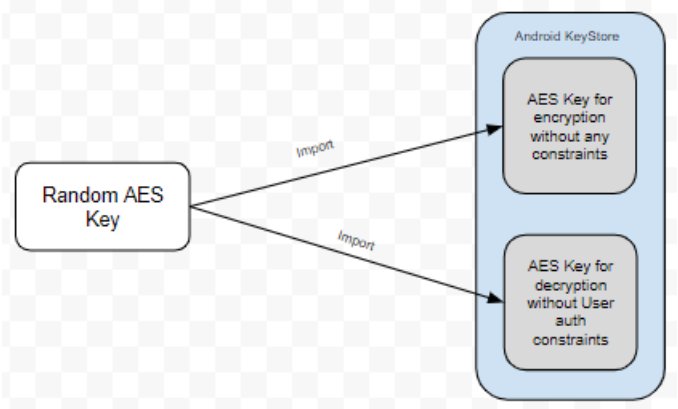


Illustration of key setup in Android Keystore

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

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 Android Key Store - 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);
    }
  }
```

Cryptographic key steup in KeyStore

4. Secure Content Provider that deal with sensitive data

Now that we have the crypto keys set up, the Sensitive Content Provider can use them to encrypt and decrypt. First whenever Content Provider gets new data for insert/update etc Secure Content Provider will check if the content has any sensitive data or not. Sensitive column info will be provided by subclass in On Create, then Secure Content Provider can simply check if the new incoming data has any of those specified column(s) if yes it will encrypt the data first then submit updated row to actual insert implementation in subclass Content Provider.

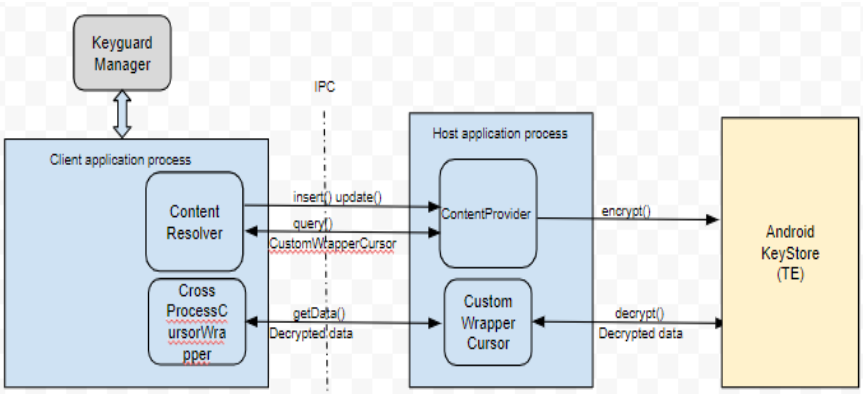
Here is an example for insert () flow encrypting a text column defined by subclass Content Provider in.

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

```
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.DEFAULT);
    } 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;
    }
  }
```

Utility functions for cryptographic key handling and operations

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510



Enhanced Security system for Sensitive persistent data protection

```
@Override // SecureContentProvider
  final public Uri insert(Uri uri, ContentValues contentValues){
    String data = contentValues.getAsString( mSensitiveColumn);
    if(data != null){
      contentValues.put(mSensitiveColumn,getEncryptedDataAES(data)
);
    }
    return insertImpl(uri, contentValues);
  }
```

insert() implementation in SecuredContentProvider

```
@Override  //Subclass like TasksContentProvider
  public Uri insertImpl(Uri uri, ContentValues contentValues){

    if (uriMatcher.match(uri) != TaskContract.TASKS_LIST){
      throw new IllegalArgumentException("Invalid URI: "+uri);
    }

    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);
  }
```

insertImpl() implementation in actual ContentProvider

Now for reading flow, Secure Content Provider will call the subclass query () function where it will fetch the requested data into a Cursor as usual. Secure Content Provider relies on a Custom Cursor which is a Cursor Wrapper (which itself is a wrapper for an actual Cursor object and wraps the Cursor result/object that the subclass returns. This Custom Cursor overrides needful functions to inspect if the fetching column is of sensitive one then this Custom Cursor will decrypt that and returns

International Journal of Core Engineering & Management
Volume-6, Issue-3, June-2019, ISSN No: 2348-9510

```
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
   public String getString(int i) {
     if(mSensitiveColumn.equals(getColumnName(i))){
       String data = super.getString(i);
       return decryptData(data);
     }else {
       return super.getString(i);
     }
   }
}
```

CustomWrapperCursor implementation

In query() flow at the client side Android framework (Content Resolver etc) will wrap our returned cursor again in Cross Process Cursor Wrapper before passing to client application. so that all the calls client application make on that final cursor will seamlessly be performed as IPC calls onto our cursor object in the Content Provider process where our Custom Wrapper cursor will handle decryption of sensitive data.

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

// subclass will implement queryImpl() as usual as per its query specifications
```

Query() function implementation in SecureContentProvider

5. Client Integration

Now Client integration would be the same as usual Content Provider except that client would know about sensitive data (the specific column(s)) in the contract/interface so that whenever it tries to access those sensitive column(s) client will ask user to authenticate and then calls 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, null);
      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.");
        }
      }
   }
```

Client Integration

V. CONCLUSION

The proposed Android framework enhances the security of sensitive persistent data by addressing key vulnerabilities that exist after device boot and user authentication. By abstracting cryptographic complexities from application developers, this framework allows seamless encryption and decryption of sensitive data while maintaining performance and usability. Key takeaways include:

1. **Improved Security:** The framework significantly strengthens data security by leveraging hardware-backed cryptographic services and enforcing user authentication constraints.
2. **Developer Convenience:** By abstracting cryptographic operations into a generic interface, developers can focus on application logic without worrying about complex cryptographic implementations.
3. **Performance and Usability:** The solution selectively encrypts and decrypts critical data, ensuring minimal performance overhead while maintaining ease of use for both developers and end-users.
4. **Extendable Design:** This framework can be adapted for different cryptographic backend (e.g., OEM-specific implementations) and can evolve to cover more threat scenarios.

This solution offers a practical and effective approach for securing sensitive persistent data in Android applications, particularly in scenarios where the device is booted but remains locked or unattended. Future research could explore extending this approach to additional threat vectors and integrating it with broader security frameworks.

VI. LIMITATIONS

- **Limited Scope to Persistent Data:** The proposed framework focuses solely on persistent data stored on disk, not covering other types of sensitive data such as data in memory or shared between applications. This leaves some threat vectors unaddressed, especially for applications that need comprehensive data protection across multiple layers.
- **Manual Calibration for User Authentication:** While the framework provides user authentication for decryption, handling this on a per-access basis might introduce additional complexity for developers and might not be suitable for all application use cases. It also requires careful consideration to ensure a seamless user experience.
- **Dependence on Cryptographic Services:** The effectiveness of this framework is reliant on the availability of Android Keystore or OEM-specific cryptographic services. In cases where these services are unavailable or limited, the security guarantees may be compromised. Additionally, transitioning to different cryptographic implementations (such as non-Android Keystore solutions) could introduce new integration challenges.

VII. 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 Content Provider 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 use cases.

REFERENCES

1. Muzammil Hussain, A.A. Zaidan, B.B. Zidan, S. Iqbal, M.M. Ahmed, O.S. Albahri, A.S. Albahri, Conceptual framework for the security of mobile health applications on Android platform, Telematics and Informatics, Volume 35, Issue 5, 2018, Pages 1335-1354
2. Shao, Yuru & Ott, Jason & Chen, Qi Alfred & Qian, Zhiyun & Mao, Zhuoqing. (2016). Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. 10.14722/ndss.2016.23046.
3. Ricardo Neisse, Gary Steri, Dimitris Geneiatakis, Igor Nai Fovino, A privacy enforcing framework for Android applications, Computers & Security, Volume 62, 2016, Pages 257-277
4. Muzammil Hussain, Ahmed Al-Haiqi, A.A. Zaidan, B.B. Zaidan, M. Kiah, Salman Iqbal, S. Iqbal, Mohamed Abdulnabi, A security framework for mHealth apps on Android platform, Computers & Security, Volume 75, 2018, Pages 191-217
5. Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. 2014. Android security framework: extensible multi-layered access control on Android. In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14). Association for Computing Machinery, New York, NY, USA, 46–55.
6. D. Feth and A. Pretschner, "Flexible Data-Driven Security for Android," 2012 IEEE Sixth International Conference on Software Security and Reliability, Gaithersburg, MD, USA, 2012, pp. 41-50, doi: 10.1109/SERE.2012.14.
7. Smalley, Stephen Dale and Robert Craig. “Security Enhanced (SE) Android: Bringing Flexible MAC to Android.” Network and Distributed System Security Symposium (2013).APIs with Enterprise Systems." IEEE Transactions on Software Engineering. In press.