SECURITY ANALYSIS OF SIGNAL ANDROID DATABASE PROTECTION MECHANISMS

Kamil KACZYŃSKI

Military University of Technology, Faculty of Cybernetics, Institute of Mathematics and Cryptology, Warsaw
e-mail: kamil.kaczynski@wat.edu.pl
Poland

Abstract: Signal is an instant messenger recognized by the international community as one of the most secure among other competitors. Signal apps for iOS, Android, and desktop have a protection mechanism for locally stored data – the encryption of the internal database. This paper presents the methods of recovering the encryption key protecting the Android app database, its storage locations, and threats to the users resulting from it. Recommendations for the software authors that allow improving the overall security of the solution and removing already-found vulnerabilities were also discussed here.

Key words: Signal, cryptography, Android, security

1. INTRODUCTION

Signal is a cross-platform application for exchanging encrypted text messages, multimedia messages, and voice and audio-visual conversations. The exchange of messages is carried out using the Internet, allowing the exchange of individual and group messages. Voice and video calls can only be made in an individual mode. The application uses the users' phone numbers as identifiers in the service.

Signal offers end-to-end encryption communication using its own communication protocol – the Signal protocol. Its components include the Double Ratchet algorithm, the triple Diffie-Hellman using the Curve25519, the AES-256 algorithm, and the HMAC-SHA256 algorithm. The cryptographic strength of the solution has been determined at the level of 128 bits [1]. Provided security level for data transmission security is sufficient for commercial data, and no attacks have been presented that allow for the practical interception of data sent via the Internet.

There is no practical method of decrypting data transmission, thus the basic method of obtaining data from the Signal messenger is the analysis of devices that store the full history of correspondence. The Signal messenger does not offer data

protection mechanisms like external password. Instead of this, it uses mechanisms provided by the operating system on which the application runs. There are also no mechanisms that would allow to wipe the conversation history, e.g. remotely, using a special code / access method, or by blocking the possibility of making too many access attempts using system's mechanisms. These restrictions indicate the possible existence of significant gaps in securing the local Signal database which is the subject of this paper.

The mobile devices are particularly vulnerable to the possibility of being lost, which requires software vendors to use security features that prevent access to confidential communication. Bypassing the security introduced by operating system vendors is not difficult and no longer impossible. An example of such threat may be the use of software such as Cellebrite UFED Ultimate [2]. Access methods for data stored in a KeyStore have also been the subject of several scientific papers, including [3] and [4]. An important aspect is also the fact that the applications leave selected data in the device's memory, even after uninstalling them [5]. In the last 2 years there were several papers have been published about the extraction and the analysis of the data in the instant messenger apps, especially for Google Hangouts [6], Viber and Telegram [7].

This paper presents the methods of gaining access to the local Signal application database operating under the control of Android operating system. Various methods of gaining access have been indicated, and each of them results in a total disclosure of conversations conducted using a compromised device.

2. ANDROID APPS DATA STORAGE

An application working under the Android operating system has several data storage options. Each of the possibilities differs in the amount of space available for the application, as well as in the level of rights necessary to gain access to the data. Following four ways of storing the application data can be distinguished:

- *Internal storage* that allows storing private application data in the device's file system. Access to the data requires root user privileges.
- *External storage*, allowing the storage of large amounts of data in the shared memory area. Used by applications for storing, for example, photos, backups, etc. The external memory should only store the external data, the leakage of which will not endanger the security of the application.
- *Shared preferences*, allowing the storage of private application data in the form of a key-value.
- *Databases* that allow storing private application data in the form of a database with a fixed structure. The Android system uses SQLite databases.

For the purpose of this paper, only the application's private data has been analyzed. The separated zone in which the application can store its data is not accessible from other applications. In this context, its analysis requires having

access to the privileges of the root user. In the case of standard devices, it is a sufficient protection of the application data, allowing it to be protected against the least complicated attacks made using amateur methods.

In Android system, the application data is stored in a dedicated directory located under the following path */data/data*. The Signal messenger also applies this rule and the directory tree is as follows:

- org.thoughtcrime.securesms
  - app_parts
  - app_webview
  - cache
    - log
  - databases
  - no_backup
  - shared_prefs

To analyze the data stored by the Signal application, it is crucial to analyze the contents of the *databases* and *shared_prefs* folders. The databases folder contains the databases used by the application, in particular the *signal.db* database that stores the history and the content of conversations. The remaining files in this directory are auxiliary and are not subject for further analysis. The *shared_prefs* directory contains a number of xml files that store the local settings of the application. The most important of them to be considered are:

- *SecureSMS-Preferences.xml*
- *org.thoughtcrime.securesms_preferences.xml*

These files are stored in plaintext, and they contain all the application settings, including those that are responsible for access using a screen lock or storage of curve25519 private keys assigned to the user. The content of the files will be discussed later in this paper.

3. ANALYSIS OF SIGNAL APP STORAGE

The signal.db database is a SQLCipher database that uses a page size of 4096 bytes and one iteration of the PBKDF2 function. These data was obtained after the analysis of the source code of the application - SQLCipherOpenHelper.java class [8]. The source code fragment initiating the database looks like this:

```
public SQLCipherOpenHelper(@NonNull Context context, @NonNull DatabaseSecret
databaseSecret) {
    super(context,     DATABASE_NAME,     null,     DATABASE_VERSION,     new
SQLiteDatabaseHook() {
      @Override
      public void preKey(SQLiteDatabase db) {
        db.rawExecSQL("PRAGMA cipher_default_kdf_iter = 1;");
        db.rawExecSQL("PRAGMA cipher_default_page_size = 4096;");
      }
```

```
      @Override
      public void postKey(SQLiteDatabase db) {
        db.rawExecSQL("PRAGMA kdf_iter = '1';");
        db.rawExecSQL("PRAGMA cipher_page_size = 4096;");
      }
    });
    this.context        = context.getApplicationContext();
    this.databaseSecret = databaseSecret;
  }
```

In addition to specifying the work parameters, an access to the database requires knowledge of the key used to encrypt it. Obtaining it depends of the used hardware platform type. For API 22 and older, the key is stored in an explicit form in *shared_preferences*, for API 23 and above, the key is stored in the Android KeyStore, and *shared_preferences* holds only its ciphertext. The source code fragment responsible for creating the base encryption key is included in the DatabaseSecretProvider.java file [9] and its form is as follows:

```
private    DatabaseSecret    createAndStoreDatabaseSecret(@NonNull    Context
context) {
    SecureRandom random = new SecureRandom();
    byte[]       secret = new byte[32];
    random.nextBytes(secret);
    DatabaseSecret databaseSecret = new DatabaseSecret(secret);
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
      KeyStoreHelper.SealedData              encryptedSecret              =
KeyStoreHelper.seal(databaseSecret.asBytes());
      TextSecurePreferences.setDatabaseEncryptedSecret(context,
encryptedSecret.serialize());
    } else {
      TextSecurePreferences.setDatabaseUnencryptedSecret(context,
databaseSecret.asString());
    }
    return databaseSecret;
  }
```

The Samsung GT-i9505 with the Android operating system version 5.0.1 was used as the test platform for API <23. Signal version 4.37.2 was installed on the device which was used to collect test data. The database files and shared preferences have been downloaded from the device. Analysis of the *org.thoughtcrime.securesms_preferences.xml* file showed the existence of the following entries:

```
    <boolean name="pref_needs_message_pull" value="false" />
        <string
name="pref_database_unencrypted_secret">1959be1ea5e35d043c8e49d5de4c6507fb3f
3440598f12a16f064480c1e5fbd2</string>
        <boolean name="pref_toggle_push_messaging" value="true" />
```

The value of *database_unencrypted_secret* is the explicit, hexadecimal form of the encryption key used to encrypt the application database. The recovered key and initialization data were entered into the SQLiteStudio 3.2.1 application, which resulted in the recovery of the entire structure and the content of the database.

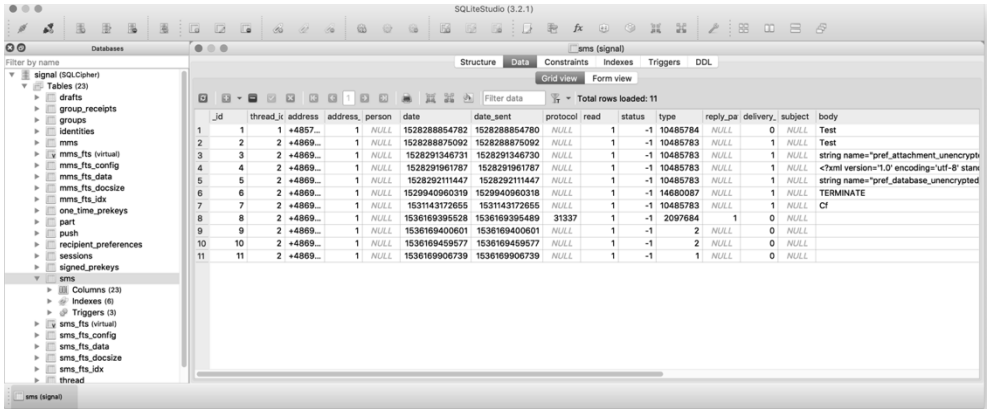

Fig. 1. Recovered signal.db structure and its content.

The database stores the full history of conversations, related contacts, and stored media. The database security used is, therefore, only illusory, not providing any security. In the case of application access protection, the user must confirm his or her identity in the same way as when unlocking the device. Unfortunately, this additional layer does not improve the security of the application, because the security can be easily removed from the application by changing the value of the *pref_android_screen_lock* key in *org.thoughtcrime.securesms_preferences.xml* to false. The security, therefore, has only a logical character and is located in a place easily accessible to the attacker.

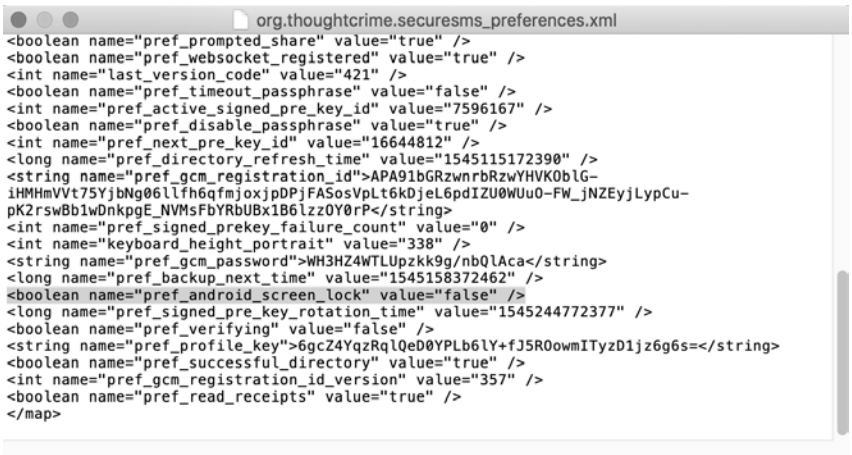

Fig. 2. Shared preferences key for disabling screen lock.

The test platform for the API >= 23 was the Google Pixel emulator operating in the API 25 system - Android 7.1. The same way of storing KeyStore data applies to all version of Android starting from API 23 [10]. All application data was obtained from the device, which indicates a changed way of storing the

database security key on the device. The appropriate entry in *org.thoughtcrime.securesms_preferences.xml* has the form:

```
…
<boolean name="pref_unauthorized_received" value="false" />
    <string
name="pref_database_encrypted_secret">{&quot;data&quot;:&quot;NVi5QQb79zetNj
7j2McyO8mCKIrtT3u7qpQDs7ZlF5OaXHE6Q4NXjCMlFM7RWUE3&quot;,&quot;iv&quot;:&quo
t;diUHbrh+BOh+AAS1&quot;}</string>
    <boolean name="pref_successful_directory" value="true" />
…
```

Recovering the encryption key requires knowledge of the secret stored in the user's device. For the Google Pixel API 25 emulator, the secret is stored in the */data/misc/keystore/user_0* directory. The recovered *10085_USRSKEY_Signal-Secret* file contains binary data of the cryptographic key used to encrypt the database key.

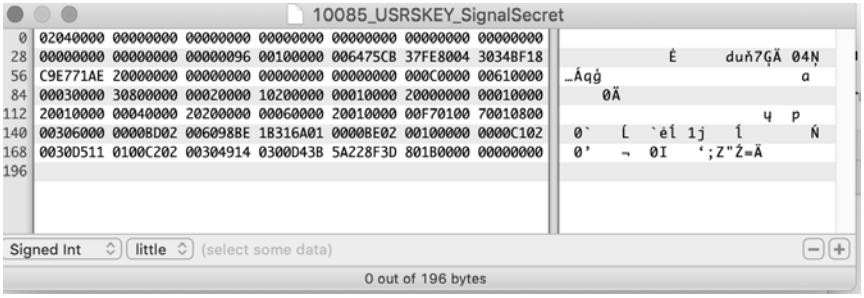

Fig. 3. Content of 10085_USRSKEY_SignalSecret file.

Recovering the encryption key can be carried out in several ways. The first one is transferring all the application data to a new device - e.g. a simulator. The application will be launched with a complete set of data. The second method is to create a malicious application that fetches the secret from the renamed file - the name is created according to the scheme *<app_id>_USRSKEY_<key_alias>*, so the *app_id* change can be considered sufficient. Recently, this method is used to analyze the recovered *USRSKEY* and the data it contains to recover the key which encrypts the database key. Each of the methods presented above is not complicated and can be performed by people with average cryptanalytic abilities. Therefore, it should be noted that the database security used, also in the case of API >= 23, is insufficient and illusory.

The method of securing cryptographic keys - stored in the *SecureSMS-Preferences.xml* file should also be considered as insufficient. The private key being stored is encrypted; however, using an easily identifiable secret - the password "unencrypted" is hardcoded. The above can be confirmed in the Signal source code - the following excerpts present the KeyCachingService.java [11] and MasterSecretUtil.java [12] class codes:

```
…
public static synchronized @Nullable MasterSecret getMasterSecret(Context
context) {
    if (masterSecret == null &&   TextSecurePreferences.isPasswordDisabled
(context) && !TextSecurePreferences.isScreenLockEnabled(context))) {
      try {
        return MasterSecretUtil.getMasterSecret(context, MasterSecretUtil.
UNENCRYPTED_PASSPHRASE);
      } catch (InvalidPassphraseException e) {
        Log.w("KeyCachingService", e);
      }
    }
    return masterSecret;
  }
…
public class MasterSecretUtil {
  public static final String UNENCRYPTED_PASSPHRASE  = "unencrypted";
  public  static  final  String  PREFERENCES_NAME              =  "SecureSMS-
Preferences";
  private   static   final   String   ASYMMETRIC_LOCAL_PUBLIC_DJB          =
"asymmetric_master_secret_curve25519_public";
  private    static    final    String    ASYMMETRIC_LOCAL_PRIVATE_DJB       =
"asymmetric_master_secret_curve25519_private";
…
```

The acquisition of the cryptographic keys allows for impersonating the service user and for intercepting incoming correspondence. The above points to a serious software vulnerability that should be removed as soon as possible.

4. CONCLUSION

The research conducted and presented in this paper indicates the presence of many threats to the Signal users who currently do not have any mechanisms to protect the stored data, except those offered by the operating systems. This situation allows not only for reconstructing the history of conversations but also for impersonating one of the users participating in the communication using the application. Such an attack will not be detected by the other side, because common identifiers will not change. This attack can be performed on any Android system, regardless of using the systemic KeyStore to store the secret key that encrypts the database. Such a serious vulnerability should be eliminated by introducing an additional secret, independent of operating system's mechanisms. It is therefore worth considering, that application developers should return to the access password offered in the versions older than Signal-Android 4.21.

REFERENCES

[1] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., & Stebila, D. (2017, April). A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (pp. 451-466). IEEE.

[2] Cellebrite UFED Ultimate. Available: https://www.cellebrite.com/en/ufed-ultimate/

[3] Sabt, Mohamed, and Jacques Traoré. Breaking into the keystore: A practical forgery attack against Android keystore, *European Symposium on Research in Computer Security*. Springer, Cham, 2016, pp. 531-548.

[4] Cooijmans, Tim, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on android. *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014, pp. 11-20.

[5] Zhang, Xiao, et al. "Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android." *NDSS*. 2016.

[6] Scrivens, N., & Lin, X. (2017, May). Android digital forensics: data, extraction and analysis. In *Proceedings of the ACM Turing 50th Celebration Conference-China* (p. 26). ACM.

[7] Kaczyński, K., Gawinecki J. "Security analysis of passcode mechanisms in Telegram and Viber." *Biuletyn Wojskowej Akademii Technicznej* 67.4 (2018).

[8] SQLCipherOpenHelper.java class of Signal Android. Available: https://github.com/signalapp/Signal-Android/blob/060bed8559692cdbfc579100e49bba0382bd2c/src/org/thoughtcrime/securesms/database/helpers/SQLCipherOpenHelper.java

[9] DatabaseSecretProvider.java class of Signal Android. Available: https://github.com/signalapp/Signal-Android/blob/28dc477b54dac1e86d52214db459bf3f99b761b3/src/org/thoughtcrime/securesms/crypto/DatabaseSecretProvider.java

[10] Android keystore system. Available: https://developer.android.com/training/articles/keystore

[11] KeyCachingService.java class of Signal Android. Available: https://github.com/signalapp/Signal-Android/blob/652306edd0df7e880b3da4897484f5686a3dada7/src/org/thoughtcrime/securesms/service/KeyCachingService.java

[12] MasterSecretUtil.java class of Signal Android. Available: https://github.com/signalapp/Signal-Android/blob/652306edd0df7e880b3da4897484f5686a3dada7/src/org/thoughtcrime/securesms/crypto/MasterSecretUtil.java

Information about the author:

Kamil Kaczyński – Military University of Technology, R&D assistant, Cryptography, Steganography, Blockchain, Cryptanalysis, Steganalysis, Mobile applications, Internet of Things (IoT)

Manuscript received on 10 September 2019