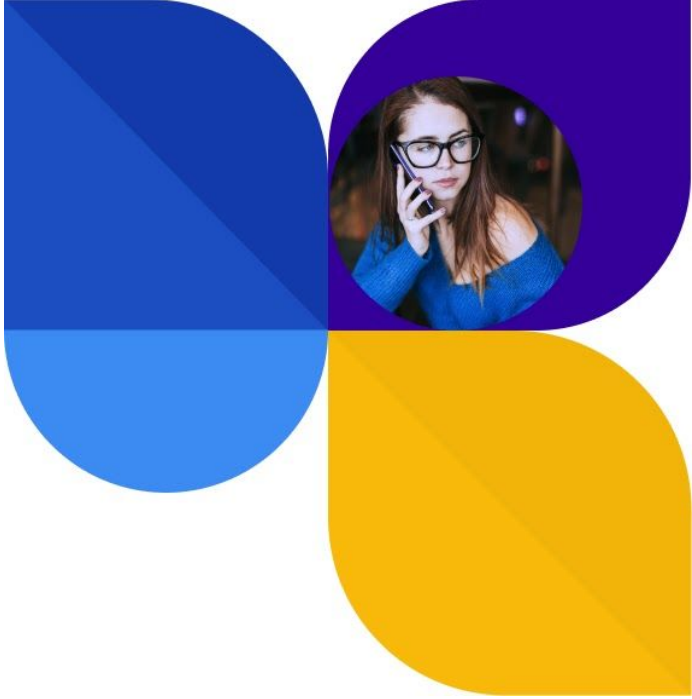


LEMON

The COVID-19 Tracing-App

Security Overview

10/09/20

Introduction

The purpose of this document is to explain the security processes that are implemented within the LEMON project. Due to the sensitive nature of some of the information that is stored and communicated, it is of the utmost importance that this information is secured so that only those authorised to have access to the information will be able to read it. This also aligns with the strict privacy policies that the app follows, meaning that the user is in total control of their data and must give express permission in order for anyone else to access this data.

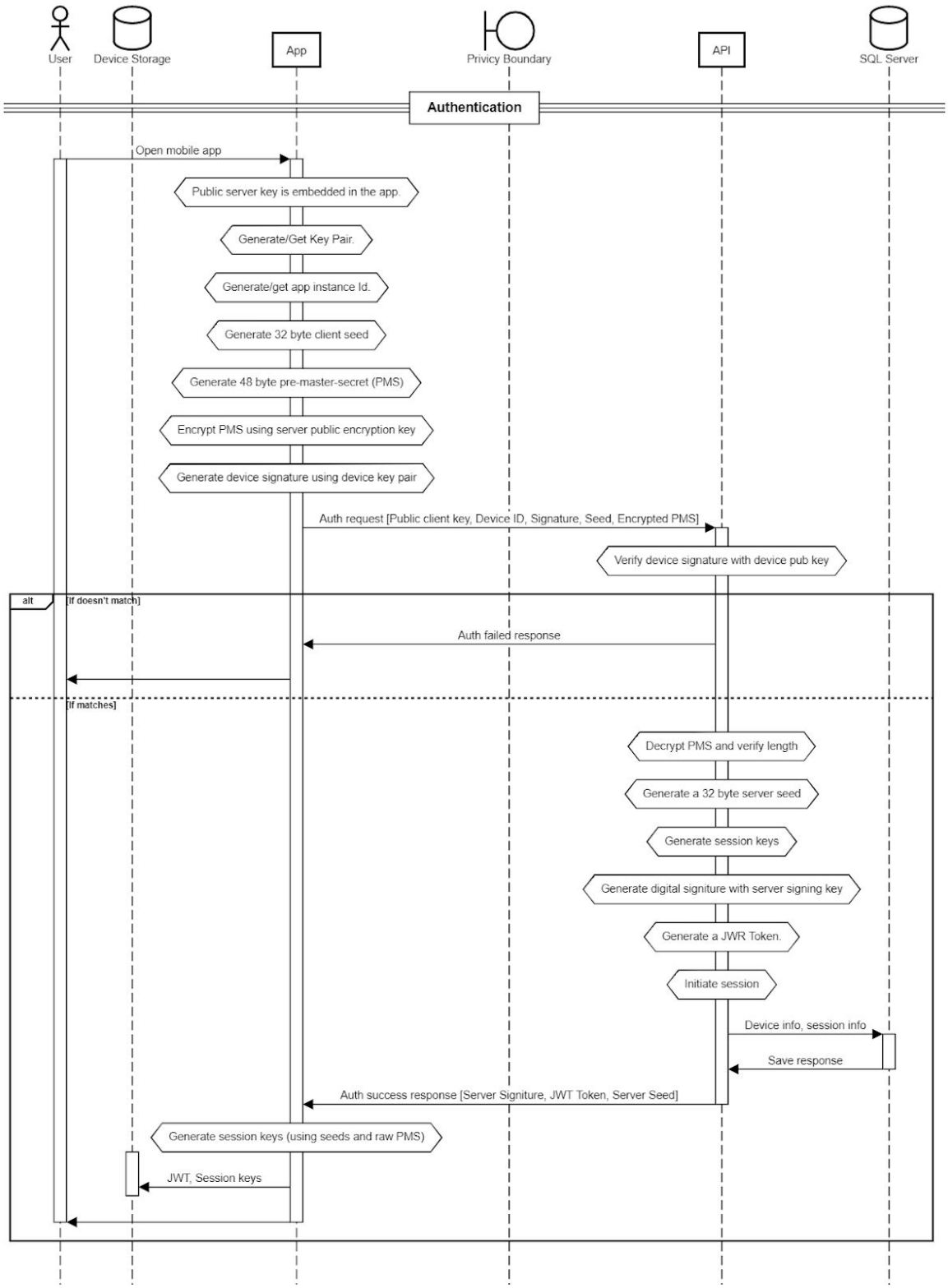
Please note that any reference to appId in this document refers to a randomly generated Hashed App Instance ID rather than any unique application identifier that can be used to trace sensitive information.

Document Scope

This document focuses on the security on the application, on the server as well as the communication between the two. It must be noted that this does not document any security associated with the Bluetooth contact tracing as that is handled independently by the DP-3T contact tracing solution. It will, however, detail how LEMON securely interfaces with the DP-3T system.

Authentication Overview

In order to keep simplicity and reduce risk the auth flow is a one-step process. This process is as detailed in the below sequence diagram:


User
Device Storage
App
Privicy Boundary
API
SQL Server
Authentication
Open mobile app
Public server key is embedded in the app.
Generate/Get Key Pair.
Generate/get app instance Id.
Generate 32 byte client seed
Generate 48 byte pre-master-secret (PMS)
Encrypt PMS using server public encryption key
Generate device signature using device key pair
Auth request [Public client key, Device ID, Signature, Seed, Encrypted PMS]
Verify device signature with device pub key
alt
[If doesn't match]
Auth failed response
[If matches]
Decrypt PMS and verify length
Generate a 32 byte server seed
Generate session keys
Generate digital signiture with server signing key
Generate a JWR Token.
Initiate session
Device info, session info
Save response
Auth success response [Server Signiture, JWT Token, Server Seed]
Generate session keys (using seeds and raw PMS)
JWT, Session keys

In summary, there are two main parts to this flow:

- **Authentication of both server and client using asymmetric signatures**
- **Establishing a session with session keys for secure communication.**

Note that the server public key is embedded on the app in order to establish trust for the app. By doing this it ensures that the app is indeed talking to the server, and not an imposter attempting to glean sensitive information.

Registration

Upon registration, the app generates a key pair that will be used from here on out to uniquely identify the application to the server as well as secure sensitive information on the device.

Upon initial authentication, there is some additional device information that is required for the functionality of the system. These include:

- **App public key**
- **Device operating system**
- **App language**
- **App push token (optional)**

Therefore these fields need to be included upon the initial registration call. On subsequent authentication calls, these values are not required, in fact, some must not be included at all. Push token, operating system and the public key cannot be updated. This is done in order to maintain security and protect user information.

In order to verify that the public key received by the server is the key that was sent by the app, signature verification takes place on this call. This ensures that the data was not corrupted or altered during transport. Should an attacker be able to get in the middle of communications and update both the public key and signature to their own, all they will achieve is registering their own device.

Payload Structure

Request Structure

```
{
        "appId": "",
        "publicKey": "",
        "operatingSystem": "",
        "pushToken": "",
        "language": "",
        "seed":"",
        "preMasterSecret":"",
        "signature":{
                "plainTextData": "",
                "signedData": ""
        }
}
```

- **AppId**: A required field containing a unique identifier for the application. This identifier is random and contains no device or user-specific information.
- **PublicKey**: Contains a base64 encoded public key. This key must correspond to the key pair used to generate the digital signature.
- **OperatingSystem**: The operating system of the mobile device. Must be IOS or Android.
- **PushToken**: A unique identifier that allows the server to push notifications to the app. This identifier contains no information about the device or user. This is optional.
- **Language**: This indicates the user's preferred language for the mobile application. All text will be displayed to the user in this language.
- **Seed**: A cryptographically secure random 32-byte string in base64 format.
- **PreMasterSecret**: A cryptographically secure 48-byte array that has been encrypted with the server public key. This is in base64 format.
- **Signature**: Digital signature performed using the client key pair. This is verified against the public key that is also contained in the payload.

Response Structure

```
{
        "data": {
                "accessToken": "",
                "accessTokenExpiry": "",
                "seed": ""
                "Signature": {
                        "plainTextData": "",
                        "signedData": ""
        },
        "meta": {
                "success": <bool>,
                "code": <int>,
                "message": ""
        }
}
```

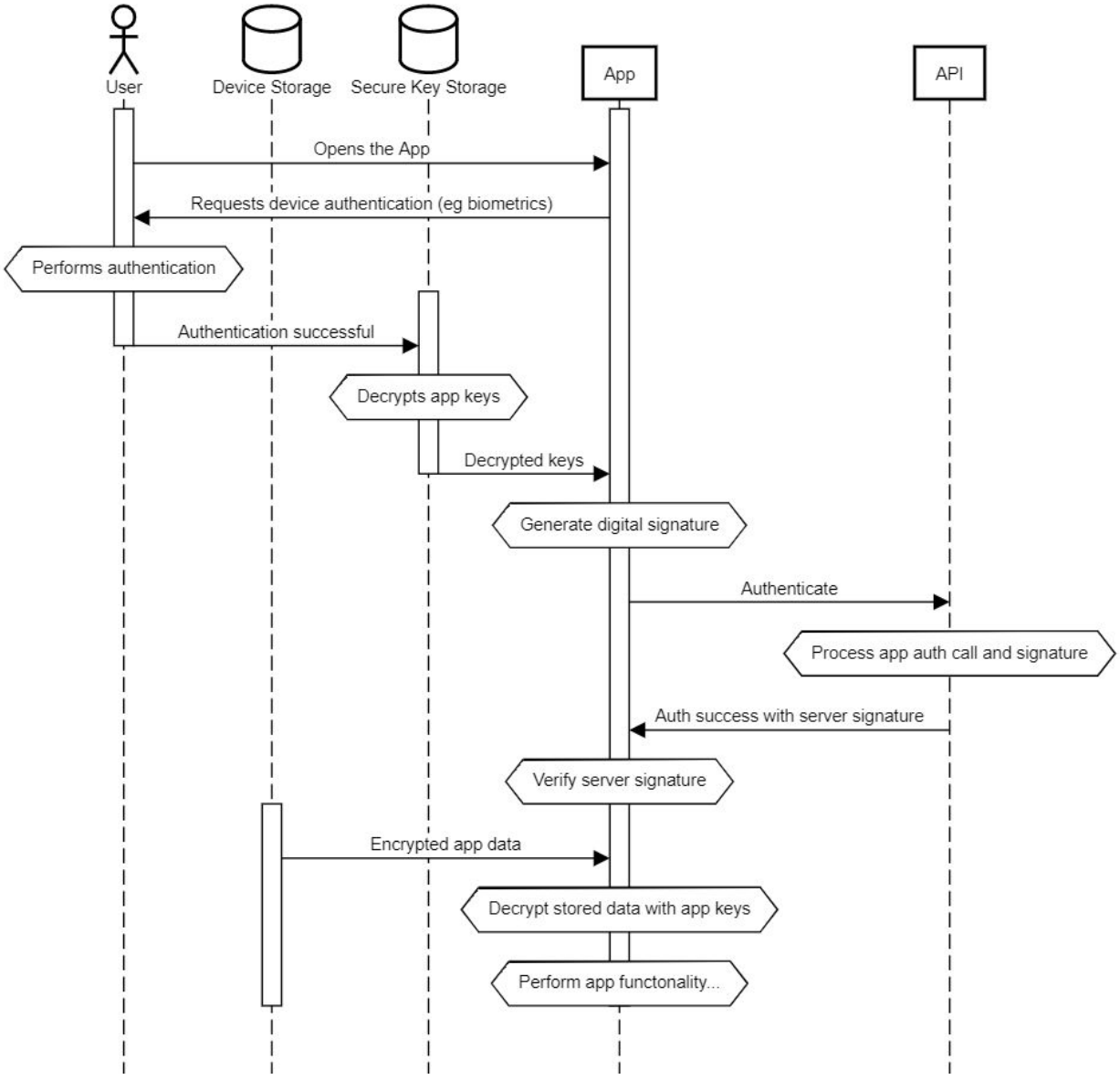
- **AccessToken**: JWT used to authenticate all API calls for the session.
- **AccessTokenExpiry**: The DateTime indicating when the accessToken will no longer be valid.
- **Signature**: Contains the server's digital signature. This is verified by the mobile app using an embedded public key in the mobile application source code.
- **Seed**: A cryptographically secure random 32-byte string in base64 format generated by the server.
- **Meta**: Additional metadata on the API call. Indicates the success of the call and any reasons for a call failure.

Login

Login uses the same end-point as registration, however, the required payload is structured slightly differently. As you can see from the request structure below that several fields are not allowed during authentication.

Mobile Application Flow

Once the key pair has been generated it is important that access to them is restricted. In order to gain access to both the key pair and the stored data on the application the following flow has been implemented:



Payload Structure

Request Structure

```
{
        "appId": "",
        "language": "",
        "seed":"",
        "preMasterSecret":"",
        "Signature":{
                "plainTextData": "",
                "signedData": ""
        }
}
```

- **AppId**: A required field containing a unique identifier for the application. This identifier is random and contains no device or user-specific information.
- **Language**: This indicates the user's preferred language for the mobile application. All text will be displayed to the user in this language. Language is optional.
- **Seed**: A cryptographically secure random 32-byte string in base64 format.
- **PreMasterSecret**: A cryptographically secure 48-byte array that has been encrypted with the server public key. This is in base64 format.
- **Signature**: Digital signature performed using the client key pair. This is verified against the public key that the server has stored for the specified appId. Verifies that the device owns the key pair of the app it is trying to authenticate as.

Response Structure

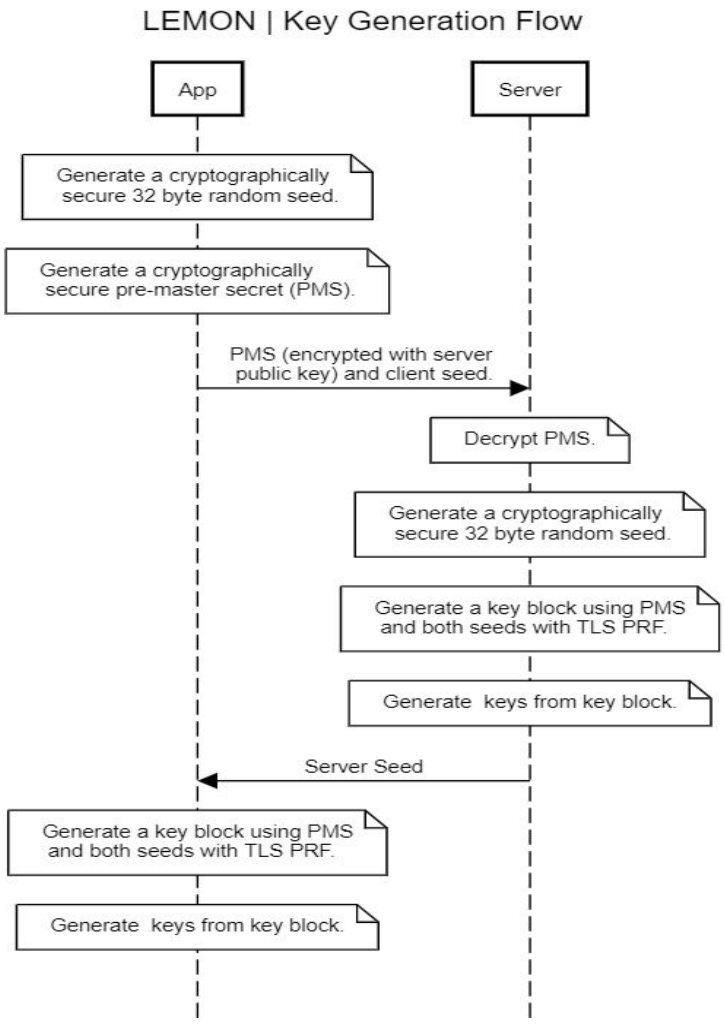
```
{
        "data": {
                "accessToken": "",
                "accessTokenExpiry": "",
                "Signature": {
                        "plainTextData": "",
                        "signedData": ""
        },
        "meta": {
                "success": <bool>,
                "code": <int>,
                "message": ""
        }
}
```

- **AccessToken**: JWT used to authenticate all API calls for the session.
- **AccessTokenExpiry**: The DateTime indicating when the accessToken will no longer be valid.
- **Signature**: Contains the server's digital signature. This is verified by the mobile app using an embedded public key in the mobile application source code.
- **Meta**: Additional metadata on the API call. Indicates the success of the call and any reasons for a call failure.

Session Key Generation

In order to minimise security risk, it was decided to use a well-established process for key generation. The process followed is based upon that used in TLS 1.2. Essentially this relies on information provided by both the client and the server in order to generate a common key block. From this key block, the specific keys required as per the encryption algorithm can be derived. This process works identically on the client and the server, meaning that no keys ever have to be communicated, eliminating any risk of exposure to malicious sources.

The key generation flow works as follows:



The Pseudo-Random Function (PRF)

In order to see a detailed breakdown of the algorithm for the TLS 1.2 PRF see the official RFC Documentation. The project implementation uses this exact flow to ensure that the key generation process follows a well-established and secure method.

Test Vectors

When building a system such as this it is very important to test that it works as expected. In order to do this test vectors are required to ensure that when your system is provided with a known input, it produces the expected output as provided by a trusted source. Test vectors were taken from this link.

Session Security

Once the session has been started using the authentication process described previously, it is important to maintain session security, confidentiality and client-server trust. This process is done using the following techniques:

- **JSON Web Tokens (JWT)**
- **End to end encryption**

JSON Web Tokens

JWT is a very well established, industry standard for maintaining session security. It is beyond the scope of this document to explain the process in detail but further information can be found here.

LEMON's JWTs contain the application Id of the app that performed the authentication process to establish the specific application that is communicating with the server. All endpoints require a valid JWT in order to be accessed. An unauthorised response is returned should a call be made with a missing or invalid JWT.

End-to-end Encryption

All information passed to the server in a request body, and from the server in a response body will be encrypted using the session keys generated during authentication. This end to end encryption uses symmetric encryption to secure the secrecy of the data transmitted as well as a MAC to secure the integrity of the transmitted data. Should any anomalies be picked up during the MAC verification or payload decryption, an unauthorised response will be issued to the client making the request.

Session Termination

All sessions and their session keys have a limited lifespan. Currently, on LEMON this is set to a 24 hour period. Once the token has expired the application must perform the authentication procedure again to re-establish a session with new session keys. By rotating keys regularly it limits the time an attacker has to break the encryption to the length of the session.

As authentication takes place in the background of the mobile application it was deemed unnecessary to use refresh tokens to refresh the session.

Cryptographic Specifications

In order to conform with well established secure algorithms, the auth flow uses the following encryption algorithms:

- **Asymmetric Encryption:**
  - **RSA-2048**
- **Symmetric Encryption:**
  - **AES-128-CBC**
- **MAC for data integrity:**
  - **HMAC-SHA256**

These are all very well established and widely used cryptographic algorithms and widely acclaimed to be not only secure but leaders in their respective fields for security.

Asymmetric Signing

Asymmetric encryption is solely used for the purpose of authentication. The key pairs are used to uniquely identify and secure device-specific information. This authentication process is done using RSA2048's built-in signature generation and verification.

Payload Structure

The signature that is used to authenticate a device is laid out as follows:

```
"Signature":{
        "plainTextData": "",
        "signedData": ""
}
```

Symmetric Encryption

In this project symmetric encryption is used to secure communication sessions between the apps and the API. In order to secure communications, it is integral to protect both data integrity as well as encrypting all information communicated. To achieve this the project uses an encrypt then mac process using AES-128-CBC and HMAC-SHA256.

Note that AES in CBC mode was chosen over GCM mode as we already have an authentication mechanism in place and do not need to use the authentication built into GCM on top of this as well.

Payload Structure

The payload structure is as follows:

```
<16 byte IV><Cipher Text><32 byte MAC>
```

- **IV**: This is the initialisation vector. It is a 16 byte, cryptographically randomised array that is required for decryption. It must be different for each payload.
- **Cipher Text**: This is the actual plain text data that has undergone encryption.
- **MAC**: This is a 32-byte array that is used to verify the integrity of the payload and ensure that it has not been altered due to errors in communication or malicious attacks. The MAC is calculated over *IV + Cipher Text*

Key Storage

This section details how the various keys utilised by the system are stored. Currently, the only keys used are:

- **Server signature key pair**
- **Server Encryption key pair**
- **App signature key pair**
- **Session keys**
    - **Encryption key**
    - **MAC key**

Server Signature Key Pair

The server signature key pair is stored using AWS Key Management Service (KMS). The server key is generated by the software and the private key is never exposed. All Cryptographic actions occur using the service meaning that the private key never has to be shared, not even to the developer that generated it.

Due to the nature of public keys, it is not essential that this is kept secret. Although it is embedded into the mobile application this is simply used so that the app can confirm that it is indeed talking to the LEMON server, and not an imposter. This process is not compromised by a leaking of the public key, therefore it is not essential to keep this secret.

Server Encryption Key Pair

The server encryption key pair is also stored using AWS KMS. As with the server signature key pair all cryptographic operations occur within the service such that the private key is never exposed.

App Signature Key Pair

The app key pair is stored in secure storage that is dependent on the device platform. These locations are:

- IOS: keychain
- Android: Shared Preferences

In both of these instances, these storages are encrypted and only accessible with successful authentication by the device's user. The authentication method used is what is specified on the mobile device for standard device authentication. This will most likely be biometrics the majority of the time although can default back to a device pin or passcode.

Should a user have set up their device to have no authentication then the only security level to external access is the possession of the phone. It was deemed to be out of the scope of the project to try to protect users that have specifically turned off device authentication.

Session Keys

The session keys are stored onto the database against the corresponding application Id. Due to the short lifespan and rotating nature of the keys, this was deemed to be secure enough. The database that they are stored in is encrypted so it is likely unnecessary to add another layer of encryption on top of this.

In spite of this though, a future consideration would be to generate an additional server encryption key that is used to encrypt the session keys before storing them in the database. Although currently deemed unnecessary, when time allows it will add an additional layer of security that can only help.

Data Storage

Due to the sensitive nature of some of the information that is being handled by the system, it is integral that all sensitive information is stored securely. This section details how this is done by the LEMON project.

Due to the decentralized nature of the system, all sensitive information is stored on the device as opposed to in a central server. This in itself adds a strong level of security inherently as there is no central store of data that can be compromised. Should an attacker be able to gain access to a device, they will only be able to see data for one user, all other user data remains safe.

Mobile Application Storage

On top of this, it is essential to maintain data security on a device for each individual user. This aims to secure both the data on the phone as well as protect falsified data from being uploaded by securing the application to a specific user, not just a device.

Encryption

All sensitive data that is accumulated by the app and stored on the device is encrypted for storage. The cryptographic algorithm used for this is AES-128-GCM. The key used for this encryption is generated and stored alongside the device asymmetric key pair. Therefore, all data stored by the application can only be accessed once key access has been granted, through the device authentication mechanism as explained earlier in the document.

Device Rooting

In order to protect any stored data on the device, should the app detect that the device has been rooted, the app will not open. Should the device be rooted it compromises the security of the application and therefore blocks it.

DP-3T Tracing Integration

The project uses the DP-3T SDKs without any changes to the code basis. On top of this, the DP-3T backend is hosted independently and does not sit behind any additional project security. This is done to allow for interoperability once it is implemented. It is taken that the DP-3T project as a whole is secure in itself and has undergone the required testing, meaning that it is not necessary to make any changes other than the required configuration updates.

The configuration changes that are specific to our project include:

- JWT signing configuration
- Database setup

In order to maintain security, new JWT signing keys are generated and replace the demo keys in the DP-3T repositories. The algorithm used to sign the JWTs is the Elliptic Curve Digital Signature Algorithm (ECDSA). This is specified in the DP-3T documentation as one of the recommended standards.

As was recommended by the documentation, the project was set up to use a PostgreSQL database that is hosted alongside the back-end. The database is set up in such a way that only the DP-3T backend has access to it.