Faculty 3: Mathematics and Computer Science
Computer Science Bachelor

Low-Level Security Patterns for Android Apps controlling IoT devices

Bachelor Thesis

Jannis Fink
fink@uni-bremen.de
Matr.-No.: 4005044

First examiner Dr. Karsten Sohr
Second examiner Prof. Dr. Ute Borman

January 17, 2019

Contents

Acknowlegements

I want to thank a few persons in advance, for helping me write this thesis. First and foremost here is Dr. Karsten Sohr, who has supervised this thesis' writing excellently. I would also like to thank Martin Schröer, who always had the time when I had a question and who provided me with advice, when I did not know how to proceed the best.

Additionally, I would like to thank Prof. Dr. Ute Borman, who agreed to take on the role of the second examiner.

Lastly, there are all the people I have asked for advice, who have proofread this work, provided me with valuable feedback and motivated me, when things did not work as planned.

Thank you!

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Bremen, January 17, 2019

Jannis Fink

1. Introduction

The number of devices additionally to computers, tablets and smartphones connected to the internet is constantly growing. According to a recent Gartner study, there will be more than 20 billion devices connected to the internet in 2020 [31]. In 2009, only 900 million devices were, which represents a 20-fold increase over the course of eleven years [32]. This trend is backed up when looking at the Google trends for the term Internet of Things (IoT), which show a massive increase in interest since January 2014 with the current high being in November 2017 [36]. The graphs for similar topics such as "smart home" show similar developments [56], albeit not as drastically as IoT.

There are already more devices connected to the internet than humans alive [31]. Amazon recently published that it has sold more than 100 million devices that come with its voice assistant named *Alexa* [5]. With more and more smart gadgets that are present in more and more peoples' homes, their safety requirements are increasing. Companies like nest are producing internet connected alarm systems and smoke detectors [42]. Should their security be compromised and the devices rendered useless as a result, the consequences may even be fatal.

1.1. Contemporary State of Smart Home and IoT Security

A big danger that is coming from the internet of things is the fact, that billions of those devices are owned and maintained by individuals, who often have no idea in how to configure them securely. When their security is compromised, they become dangerous assets in the hands of cyber criminals. Recent examples of these devices include the Mirai botnet, which was used on a wide spread attack on Domain Name System (DNS) provider Dyn in late 2016, for example. The attacks on Dyn have resulted in many popular and big sites such as Netflix, eBay and reddit being offline. This botnet was even used to successfully attack Liberias internet infrastructure resulting in short periods of time where the whole countries internet was disrupted [40]. To infect such a large number of devices, the Mirai botnet used the fact that many manufacturer were using easy to guess default credentials for the admin/root user of their products [43].

1.2. Selection of specific use cases

This thesis has been written in the context of the project "Secure Smarthome Apps" within the Center for Computing Technologies (TZI) at the University of Bremen[1]. The problems that are discussed in this thesis have been faced during the discussion that took place in the context of this project. Each of these problems will be dealt with in a separate chapter of this thesis, with the goal of developing a working prototype for each one, a pattern. While this worked out for the first two patterns, which deal with storing secrets to disk and implementing OAuth securely, the third chapter, which is about encrypting all network traffic that is generated by Real Time Streaming Protocol (RTSP), could only be implemented partially.

Every problem has been chosen because of its immediate relevance for the mobile application environment. Sites such as the Open Web Application Security Project are maintaining lists of common security problems, where the categories these problems fit into have been featured in the past [48], too. Additionally, because storing secrets and implementing the OAuth flow securely are common problems with well defined requirements, they looked like good examples of problems where one (or a few) standard implementation could seriously improve security. When this implementation could be used easily by most developers, it may discourage them from reinventing the wheel in a probably insecure way.

As a contrast to the first two problems, which are somewhat isolated in a way that they do not require the interoperation of many different components but instead offering a simple Application programming interface (API) for the developer, encrypting RTSP traffic is not as isolated. Here both client and server have to work with each other in order to make the encryption work. RTSP is a protocol commonly used for live streaming of video and audio. While the probably most used protocol for communication over the internet, Hypertext Transport Protocol (HTTP), has a very matured way of encrypting its traffic, there does not seem to be a simple solution for RTSP, so this thesis tries to solve this problem. As already mentioned, this has not worked out as planned, and just a part of the RTSP traffic has actually been encrypted. A detailed description about what is left to do has been written, instead.

A fourth topic, which was planned to be worked on in this thesis has been left out, because it has been worked on as part of another bachelor thesis in the context of the same project. This topic was how to manage Transport Layer Security (TLS) certificates in a local network, where domain names are not necessarily unique and where the servers are not usually administered by professionals. Tobias Osmers has looked at this topic in his 2018 thesis "Sicherheitsanalyse der TLS-Client-Implementierung von Android-Anwendungen bezüglich ihrer Kommunikation mit einem Gerät im lokalen Netz" [44].

[1] https://www.informatik.uni-bremen.de/~sohr/SecureSmartHomeApp/index_e.htm

2. Method

This chapter will describe how this thesis is laid out and how the use cases described in section 1.2 have been implemented and hardened against potential security gaps. While chapters 4 and 5 have been implemented as a working prototype, chapter 6 has no implemented solutions. So in contrast to the first two chapters, this chapter gives a detailed overview of what needs to be done in order to actually implement RTSP securely.

To develop the Android prototypes for this thesis, Android Studio in version 3.1.4 was used[1]. At the beginning of this thesis, the most recent Software development kit (SDK) version (27) of that time was used. With version 28 of the SDK released in June 2018, the SDK version used for this thesis has been updated, too, in order to utilize the added feature of saving a security key inside of secure hardware [22].

While chapter 6 does not have any implementation, the demo application developed for this thesis contains an activity capable of playing an RTSP stream. For this, the user must start the activity, paste a valid RTSP URL into the input field and click the button labelled "OK". This activity was implemented to verify that Android's `VideoView` can play RTSP. Additionally, this chapter uses both `ffmpeg` and `live555`, which are explained in detail in section 6.2. The `ffmpeg` version selected was the most recent one available within the Ubuntu 18.04 package repository at the time of writing, 3.4.4[2]. `live555` was integrated by downloading the source code and compiling it manually. At first, version 2018.10.17 and updated later to the newest version, 2018.12.14m which can be downloaded on the projects homepage[3]. There was no particular reason for this other than using the newest software, which worked without any changes to the existing source code. While both `ffmpeg` and `live555` should run on Windows, it has not been tested, since everything, including the Android library, was developed and tested on Ubuntu 18.04.1.

In order to make sure that the resulting prototypes have no known security flaws, the internet standards used have been read and understood. As an example, the Request for Comments (RFC) for OAuth itself already has some security protocols build in [34], while other attack vectors for OAuth are explained and mitigated using additional methods defined in other RFCs [52]. Additionally, when there are multiple possible ways of implementing a specific solution, publications from third party security professionals have been consulted, in order to choose the most secure way for the actual implementation.

[1] `https://developer.android.com/studio/`
[2] `https://packages.ubuntu.com/bionic/ffmpeg`
[3] `http://www.live555.com/liveMedia/public/`

These publications were mainly published by the german Federal Office for Information Security [35].

During development, the "Android Virual Device Manager" which is integrated into Android Studio has been used to manage the virutal Android devices that were used for development. This tool integrates well into Android Studio and made debugging very easy. During the development for chapters 4 and 5, this was very useful, since many issues that occurred during development could be easily spotted and fixed this way. Additionally, the program Wireshark, a network protocol analyzer has been used to inspect the RTSP handshake to get a deeper unterstanding of how the protocol works.

2.1. Interface SecureContextInitializer

After setting out the tools that have been used, this section will describe the interface `SecureContextInitializer`. This interface was designed for this thesis as a common interface for all Android solutions developed in the different chapters with the goal of providing a similar way of initializing the different solutions. It provides three methods the developer must implement.

2.1.1. SecureContextInitializer.initialize

This method expects the activity calling it as its single parameter. It must take care of initializing everything. It returns a boolean indicating whether the initialization was completed or is still pending because of some asynchronous work or user interaction to wait for. When it returns `true`, the instance is fully initialized.

2.1.2. SecureContextInitializer.isInitialized

When `SecureContextInitializer.initialize` returned `true`, this method must return that, too. However, when the initialization was not completed after the call, this method must return `false` as long as the instance is actually being initialized.

2.1.3. SecureContextInitializer.setOnInitializeListener

If `SecureContextInitializer.initialize` is asynchronous, the user of this class can register a listener with this function. This listener is called when the initialization is completed. It is only possible to set a single listener object, setting another will override the one previously set. This listener is guaranteed to be only called once when the initialization is complete. If the initialization is complete at the time when the listener is set, it is called immediately.

3. Background

This chapter will describe the basic APIs that are used in this thesis. While some of the basics are described in this chapter, some more specific APIs are described in the chapter they have eventually been used. Because this thesis focuses strongly on the Android operating system, its main components and API will be described here, as well as the common interface that was developed to implement the solutions for the different chapters.

Android is an open source Linux based operating system built and maintained by Google [25]. On top of the Linux kernel, Androids Hardware Abstraction Layer (HAL) provides standard interfaces for the higher-level Java-API. An image of how these components are arranged within Android can be seen in Figure 3.1.

Apps for Android are typically written in Java and run by the Android Runtime (ART). In Android 5.0, ART replaced Dalvik as the Android runtime. While apps can be written in a language other than Java, this has not been used in this thesis. This thesis uses the current API level as of October 2018, which is API level 28.

3.1. Android Manifest

The android manifest is placed in a file named `AndroidManifest.xml` and contains all meta information about an app or library. It lists all permissions an app or library may use and request along with all activities and intents. Listing 3.1 shows an example for such a manifest file.

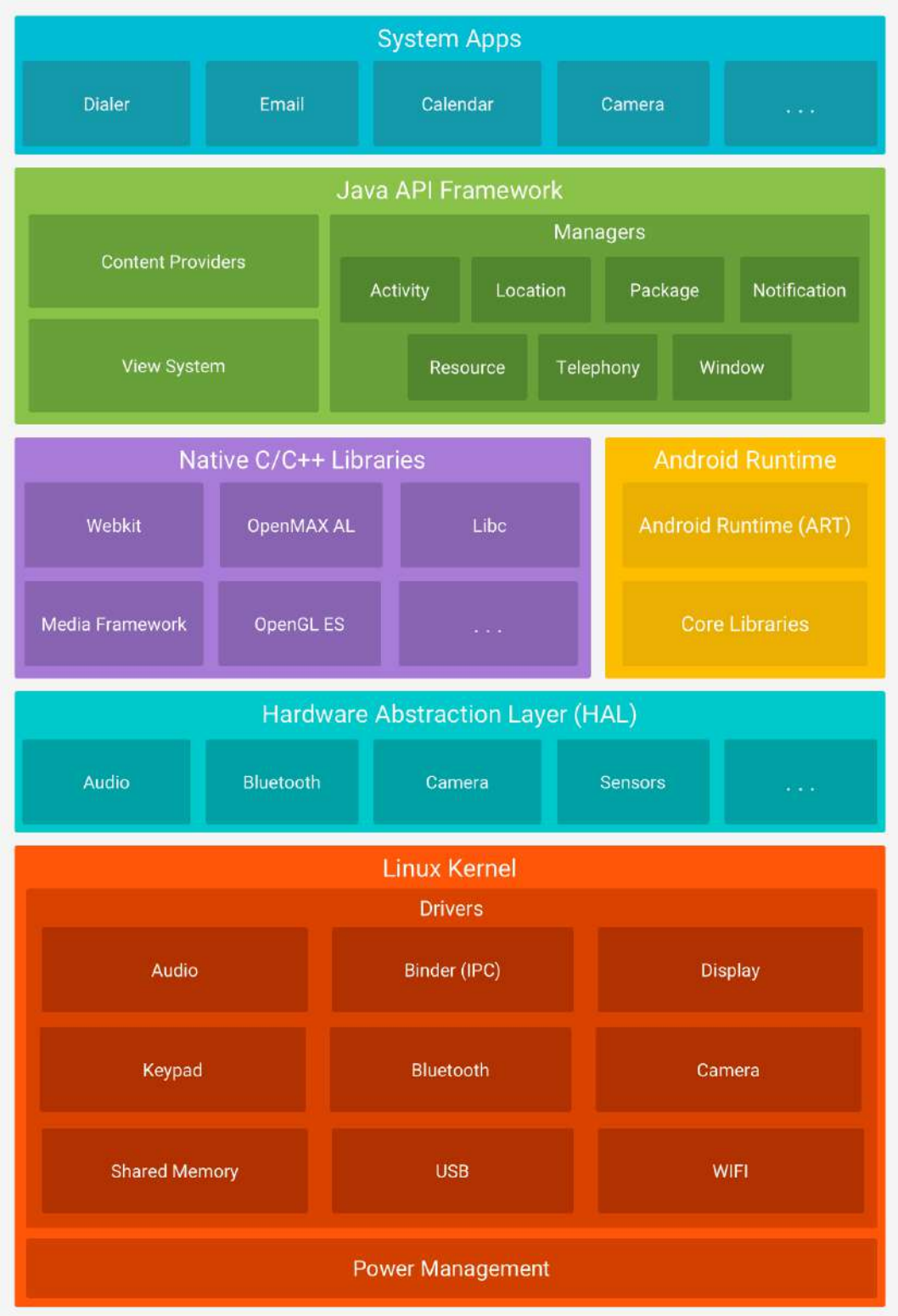

Figure 3.1.: Android software stack [25].

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.bremen.uni.jannisfink.androidsecurity">
    <uses-permission android:name="android.permission.INTERNET"/>

    <application>
        <activity android:name=".oauth2.activities.
    AuthorizationGrantConsumerActivity">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/
    >
                <category android:name="android.intent.category.BROWSABLE
    "/>
                <data android:scheme="oauthschemecallback"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 3.1: Example of an AndroidManifest.xml file

3.1.1. Permissions

All permissions used at runtime must be defined in the android manifest. As an example, the android manifest in listing 3.1 contains the permission `"android.permission.INTERNET"` which is neccessary for the application to communicate via the internet. Prior to android 6.0, all permissions listed in the manifest file must be accepted by the user in order to install the app. With android 6.0 and upwards, permissions are requested at runtime and can be denied by the user [24].

3.2. Activity and Context

The `Context` class is an interface to the global information about the application environment. It allows to access to the application resources as well as launching activities and receiving intents [15].

All activities in android are subclasses of `Context`. They are mainly used to interact with the user by creating a window which contains the user interface (UI) and can control the UI elements in that window as well as listen to specific user interactions. The UI itself is typically described in an Extensible Markup Language (XML) file which is loaded by the activity [10]. Each UI component in this XML file must have a unique id and can be accessed through it. Listing 3.2 shows an example of such an XML file, while listing 3.3 shows how to interact with it.

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <Button
            android:id="@+id/button_id"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="The text"/>
    </LinearLayout>
</android.support.constraint.ConstraintLayout>
```

Listing 3.2: Example of a XML file describing a UI

```java
import android.util.Log;
import android.view.View;
import android.widget.Button;

Button button = findViewById(R.id.button_id);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.v("button", "clicked");
    }
});
```

Listing 3.3: Example of a UI controller class

3.3. Intent

To open an activity from another, a developer must create an Intent. Additionally to invoking other activities, an intent can also carry data across activities. Figure 3.4 shows how to create a new intent and save some extra data inside of it, while Figure 3.5 shows how to read this data from within the invoked activity.

```java
Intent intent = new Intent(InvokingActivity.this, OtherActivity.class);
intent.putExtra("string", "some string");
intent.putExtra("bool", true);
```

```
4 startActivity(intent);
```

Listing 3.4: How to create a new Intent and store data inside of it

```
1 Intent intent = getIntent();
2
3 String s = intent.getStringExtra("string");
4 Boolean b = intent.getBoolenExtra("bool");
```

Listing 3.5: Read the extra data from an intent

4. Secure storage of credentials

When dealing with sensitive information in an application, it is crucial to store them in a way they cannot be read by entities other than the application dealing with them. One example of such information is the access token needed in OAuth 2.0, which is discussed later in this thesis. This problem or storing sensitive information securely is a well known problem, being featured in the "Top 10 Mobile Risks" list of the OWASP Mobile Security Project both in 2014 and 2016 [48].

One way of achieving this is to encrypt the information before saving them to disk. But in order for the encrypted information to be secure, the key (or key pair) used for encryption must also be stored securely. This chapter will take a look at how to achieve this and explore one rejected idea which turned out to be not as secure as initially thought.

To encourage use of this chapters API, it is designed in a way the developer using it does not need to know anything about encryption or the way the encryption keys are stored. Listing 4.1 shows the initial API design with Listing 4.2 showing the way it was eventually solved. The final API uses the interface developed in section 2.1.

```
1 CredentialSaver saver = new CredentialSaver();
2
3 // save secret
4 saver.storeSecret("key", "secret value");
5
6 // restore secret
7 String secret = saver.getSecret("key");
```

Listing 4.1: Initial API design for saving credentials

```
1 final CredentialSaver saver = new KeystoreCredentialSaver();
2 saver.initialize(activity);
3
4 // save secret
5 saver.storeSecret("key", "secret value");
6
7 // restore secret
8 String secret = saver.getSecret("key");
```

Listing 4.2: Final API design for saving credentials

4.1. Background

This section explains the Android APIs, features and security mechanisms used to implement the solution.

4.1.1. Shared Preferences

The Shared Preferences on Android provide a simple API for storing data on and retrieving it from disk [27]. The example in Listing 4.3 shows how one can save the string `"some data"` under the key `"the key"` in the shared preferences.

```
SharedPreferences preferences = context.getSharedPreferences("shared
    preferences name", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = preferences.edit();
editor.putString("the key", "some data");
editor.apply();
```

Listing 4.3: Shared Preferences example for storing data

The example in Listing 4.4 shows how the data stored for the key `"the key"` can be read. The string `"default value"` is the default value for the case the key does not exist.

```
SharedPreferences preferences = context.getSharedPreferences("shared
    preferences name", Context.MODE_PRIVATE);
String data = preferences.getString("the key", "default value");
```

Listing 4.4: Shared Preferences example for retrieving data

Shared Preferences are stored as XML files on disk. The second parameter of the `getSharedPreferences` method is the mode. When it is set to `Context.MODE_PRIVATE`, as it is in this example and the final implementation, the file created for storing the preferences can only be accessed by the application that created the file or any application that shares the same user id [16].

4.1.2. Android KeyStore

By the Android documentation, the keystore is described as this: "The Android Keystore system lets you store cryptographic keys in a container to make it more difficult to extract from the device. Once keys are in the keystore, they can be used for cryptographic operations with the key material remaining non-exportable." [11]. Keys stored in the Android keystore are not stored in the application processes memory, the cryptographic operations are carried out by a specialized system process, so the key is not accessible from within the application process. Additionally, the key material might be bound to

secure hardware so that an attacker cannot extract the keys from the device even if he has access to the devices internal storage [26]. Section 4.4 explains this further.

The Android keystore is one specific instance of the Java keystore. To access it, one must use the function `KeyStore.getInstance("instance name")` and pass `"AndroidKeyStore"` as the instances name. To access a key in the keystore, the keystore must be loaded first. Then, it is possible to access different keys by their alias.

The example in Listing 4.5 shows how to access a specific key in the keystore and use it to encrypt a string `"data to encrypt"` using Rivest–Shamir–Adleman, a public-key cryptosystem (RSA). This example assumes that a key for the alias `"key alias"` already exists. If it would not exist, the result of `keyStore.getEntry("key alias", null)` would be `null`.

```
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);

// load the key and cast it correctly
KeyStore.Entry entry = keyStore.getEntry("key alias", null);
KeyStore.PrivateKeyEntry privateKeyEntry = (KeyStore.PrivateKeyEntry)
    entry;

// initialize the cipher
Provider provider = Security.getProvider("AndroidKeyStore");
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPPadding", provider);
cipher.init(Cipher.ENCRYPT_MODE, privateKeyEntry.getCertificate().
    getPublicKey());

// the actual encryption
byte[] encryptedData = cipher.doFinal("data to encrypt".getBytes());
```

Listing 4.5: How to encrypt data using a public key off the android keystore

4.1.3. KeyPairGenerator

The `KeyPairGenerator` is the API used to generate a new key pair. Similarly to the `KeyStore`, one can get an instance of it by calling `KeyPairGenerator.getInstance("instance name")`. The instance named `"AndroidKeyStore"` is capable of creating keys for asymmetric encryption and saving them directly into the android keystore.

To create a new key pair, the class `KeyGenParameterSpec` is needed additionally. It defines all key properties like the alias (the name under which the key is saved), the key size, whether the key is used for encryption, decryption, signing and/or verifying, and many more.

The example in Listing 4.6 shows how to create a new RSA key pair to be used only for encryption and decryption:

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_RSA, "AndroidKeyStore"
);

KeyGenParameterSpec spec = new KeyGenParameterSpec.Builder("key alias",
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT
).build();

keyPairGenerator.initialize(spec);

KeyPair pair = keyPairGenerator.generateKeyPair();
```

Listing 4.6: How to create a RSA key and save it in the android keystore

Instead of using `KeyGenParameterSpec`, it is possible to use the class `KeyPairGeneratorSpec`. While it is deprecated since Android 6.0 and should be replaced with `KeyGenParameterSpec`, there are still many references found using this old implementation. The APIs to construct them are mostly identical, both follow the builder design pattern [30] and have methods named roughly the same. More Information on how to securely store a key pair in the Android keystore can be found in section 4.4.

4.2. Solution using Android Keystore and Shared Preferences

This solution uses the Android keystore for creation and secure storage of a key pair. This key pair is then used to encrypt the input data and store it inside using the SharedPreferences API.

4.2.1. Interfaces

It was initially planned to provide multiple solutions for this problem using an unified interface for all to implement. While just one solution has eventually been implemented, it still uses this interface. Discarted solutions can be found in section 4.3.

The interface is called `CredentialSaver` and consists of two methods. It extends another interface called `SecureContextInitializer` consisting of three additional methods. This interface was also used to implement the solution for the chapter OAuth 2.0 and is explained it detail in section 2.1.

CredentialSaver.storeSecret

This method expects a key-value pair. It must save the value given securely. The key must act as an identifier. Both values must not be `null`.

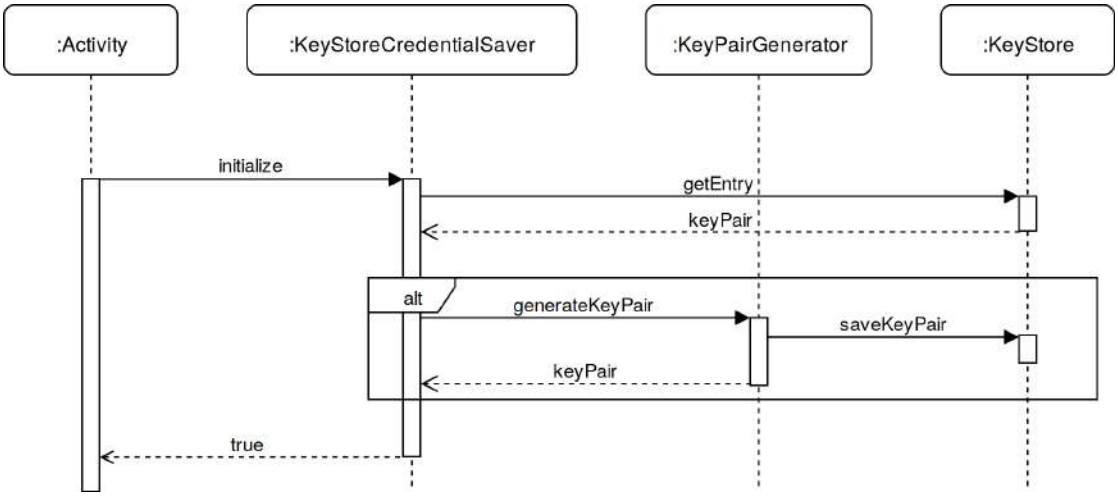

Figure 4.1.: Initialization of the `KeyStoreCredentialSaver`.

CredentialSaver.getSecret

This method is used to get a stored secret. It expects the key as its only parameter, which must not be `null`, either. When there is a secret stored for this key, it will return it. Otherwise, it will return `null`.

4.2.2. Class KeyStoreCredentialSaver

The `KeyStoreCredentialSaver` class is implementing the interface defined in section 4.2.1 as well as the one it is extending. To use it, one must create an instance by using the default constructor with no arguments. Following that, the user must call `initialize`. During initialization the credential saver checks if there is already a certificate to use. The certificate alias is based on the class name of the activity class given in the initialization function. When there already is a certificate, the function retrieves its private and public key and returns. When there is none, a certificate is created synchronously during initialization and stored in the keystore for later use, too. More information on the algorithm used for encryption and decryption can be found in section 4.2.3. The initialization function works completely synchronously, so the credential saver is ready to use after it returns. Figure 4.1 shows the initialization flow with the optional part of creating and storing a key pair in the keystore. The block labeled “alt” is optional and only executed, if no certificate has been generated, yet.

When a user stores information in this credential saver, it is first encrypted using the public key restored from the keystore. The encrypted information is then stored in the shared preferences. The shared preferences, however, can only store primitive types such as integers and booleans alongside strings. The cypher returns the encrypted data as a byte array. In order to save it in the shared preferences, it is first converted into the base64 format and stored as a string. To decrypt the data when the user is requesting

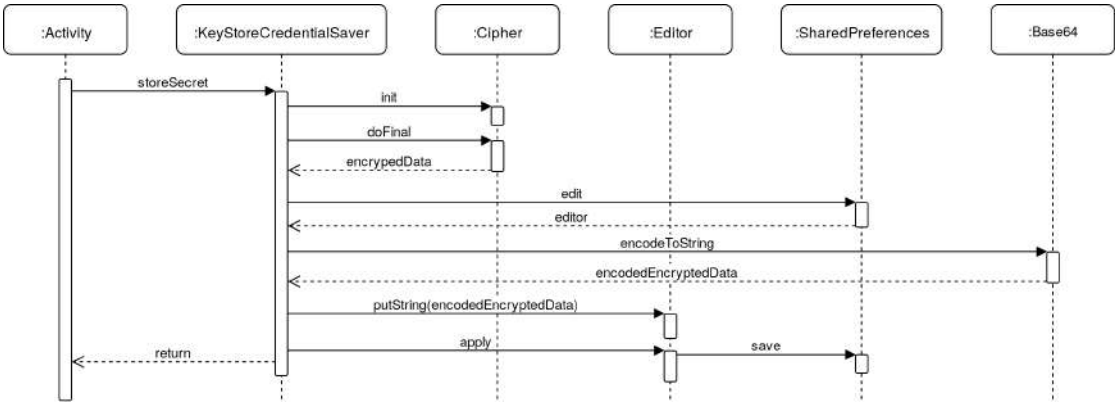

Figure 4.2.: Process of storing a secret within the `KeyStoreCredentialSaver`.

it, it has to be converted back into a byte array. The algorithm used for encryption and decryption is explained in detail in section 4.2.3. Figure 4.2 shows how a secret is encrypted and stored when using the `KeyStoreCredentialSaver`. The static methods for getting the instances of the cipher and the shared permissions have been omitted for clarity.

Both the key pair stored in the keystore and the shared preferences file can only be accessed using a key alias or shared preferences name. This parameter is a string. In order to generate this string dynamically but deterministically, the credential saver uses the canonical class name of the activity given in the initialization function for it. This way, there are separate key pairs and shared preference files for every activity using the credential saver. This design has its own advantages and flaws. One major advantage is that, if a single key pair is compromised, it does not necessarily compromises all credentials saved by an application. A major disadvantage is that the user cannot easily delete or rename an activity that is using the credential saver and that he cannot easily access the same secrets from different activities. Section 7.1 takes a closer look on this problem.

4.2.3. Encryption and Decryption algorithm used

The Java API for encryption and decryption is located at `javax.crypto.Cipher`. To get an instance of a cypher, one can use the function `Cipher.getInstance`. This function expects a string describing the algorithm to use as its first parameter and a provider which can provide this exact algorithm as its second parameter. The first parameter is necessary because a provider may be able to implement several algorithms. The second parameter is necessary because one algorithm may be implemented by several providers. It can be omitted, in this case, the cipher will choose an unspecified provider which supports the algorithm wanted.

The algorithm used in this thesis is specified by the string `"RSA/ECB/OAEPPadding"`. This

string consists of three parts. The first part is `"RSA"`. It describes the algorithm to use. RSA is an asymmetric cipher, meaning that the key required for encryption is different to the decryption key. The key used for encryption is called public key and can be shared publicly. The other key is called private key and should be kept confidential by its owner. Together, those two keys are the key pair.

The second part of the string describes the mode to use for encryption and decryption. In this case, it is `"ECB"`, which stands for "Electronic Codebook". This encryption mode works by divinding the message into blocks and encrypting each block separately.

The last part of the string is `"OAEPPadding"`. It describes the padding to be used if a block to be encrypted is too short. For the padding, Android supports two versions, `"PKCS1"` and `"OAEP"`, with the latter being used in this chapters solution. The German Federal Office for Information Security (BSI) recommends this, too, because there are known attacks for `"PKCS1"` [35].

The Android Keystore is able to generate Advanced Encryption Standard (AES) keys for symmetric and RSA key pairs for asymmetric encryption. While it would be possible to use AES instead of RSA, it was chosen not to. RSA offers the advantage of being able to additionally sign and verify the encrypted secret, so that it cannot be modified. Section 7.1.2 further elaborates on this idea.

4.3. Discarded solution using the Android AccountManager

As the name suggests, the AccountManager provides a centralized registry of the user's online accounts [9]. An account within the AccountManager can be associated with a "bundle" of user data, which can store simple key-value pairs [14].

The idea was to provide a class that implements the same interface (`CredentialSaver`) and can be used the same way as the `KeyStoreCredentialSaver` was. But when storing information within the AccountManager, this information is not encrypted in any way. Its sole protection is the fact that it is only accessible by the root user [17]. When using secure hardware for an encryption key, the keystore solution from section 4.2 is much safer, as discussed in section 4.4. The AccountManager could have been used to store the encrypted data, but the shared preferences API is much simpler to use and more suited for this type of data.

4.4. Protection against illegitimate access to the stored credentials

In this section will be discussed how secure the provided solution is compared to just storing the credentials directly in a file.

When not explicitly readable or writable for the whole system (also known as "world

readable"), a file created by an app and stored in its own directory is only accessible by this app. This is achieved by having every app run with its own user id. It is possible for two apps to share the same user ID, but that is unusual and requires special prerequisites, so it will not be discussed in this thesis [8].

As a result of that approach, any secret stored in a file by an app is stored securely as long as the attacker has no root access to the device. When he has root access, he can access the file containing the secret. For this reason, the focus of this section is not whether the attacker can read the encrypted secret or not, but if he can access and use the key stored on the device to actually decrypt it.

The Android Keystore offers two mechanisms to prevent an attacker from extracting the key from the device. Firstly, when using a key from the Keystore, it is not actually accessible from the application process, but the cryptographic operations are carried out by a specialized system process. Secondly, Android offers a feature to store the key material inside secure hardware. When a key is stored inside secure hardware, an attacker with access to the device may be able to utilize the keys on the device, but is unable to extract them [19]. This feature has to be turned on explicitly by the developer using the Keystore API. To make use of the secure hardware, the developer must explicitly set the correct flag during key generation and use an algorithm and parameters that are supported by the secure hardware. As of API level 28, the supported algorithms include RSA with a 2048 bit long key. Longer keys are currently not supported [21]. For RSA, the BSI currenty recommends keys with a length of at least 2000 bit, so this is fine for now. After 2022, keys should have a length of at least 3000 bit [35]. How to upgrade the algorithm and its parameters within a deployed application is no part of this thesis, but an upgrade path is considered in section 7.1.3. Listing 4.7 shows an example of how to set the flag to store the key in secure hardware. The key generation may fail when no such hardware is available, so the implementation for this chapter's solution provides a fallback solution which uses the old way of storing the key to disk with only the root user being able to access it [6].

```
1  KeyGenParameterSpec.Builder builder = new KeyGenParameterSpec.Builder("
       key alias", KeyProperties.PURPOSE_ENCRYPT | KeyProperties.
       PURPOSE_DECRYPT);
2  builder.setIsStrongBoxBacked(true);
3
4  KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(
5      KeyProperties.KEY_ALGORITHM_RSA, "AndroidKeyStore"
6  );
7  KeyGenParameterSpec spec = builder.build();
8
9  keyPairGenerator.initialize(spec);
10
11 KeyPair pair = keyPairGenerator.generateKeyPair();
```

Listing 4.7: Minimal working example of how to store a key pair in secure hardware

5. OAuth 2.0

OAuth 2.0 is an authorization framework (called just "OAuth" for the rest of this thesis, as older versions have not been looked at). It enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf [34].

OAuth defines the way a resource owner can grant a third party (also called "client") access to a protected resource. During the whole authorization process, the client will never gain access to the resource owners credentials[1]. If the resource owner grants the client access, the client is provided with an access token. This token can be used by the client as an alternative to the resource owners credentials. It may have a limited scope and can be revoked by the resource owner at any time.

This chapter will develop a possible way of implementing the OAuth authorization flow for an android app securely. Additionally, it will take a look at an existing OAuth client library for Android developed by the OpenID foundation in cooperation with Google called "AppAuth-Android".

5.1. Background

This section covers the basic Android APIs used to implement the OAuth authorization flow. Additional information about the Android APIs and other basics used can be found in chapter 3. This section only contains the basics needed for this chapter, while the chapter mentioned above contains the basics necessary to understand the whole thesis.

5.1.1. Displaying web content within an android app

OAuth makes use of the web browser when authenticating the user at the authorization server. The user is presented with a web page, where he would typically have to sign in if he hasn't already done it. After that, he is lead through the OAuth approval flow by the authorization servers user interface. When working with android apps, there are three different ways of displaying web content from within an app. The first one is to just open the web browser configured by the user as his default browser for a specified

[1] There are several different ways for the client to obtain access. One of them, the "Client Credentials Grant" [34], makes the user's credentials accessible to the third party, but this grant type is not discussed in this thesis.

url. By doing so, the user can use his saved logins, but is forced to leave the app and return to it later. Additionally, opening the browser is a potentially slow operation, as the program has to be loaded from disk, if it is not already loaded and the operating system has to switch to the browser's application.

Another way of displaying web content within an app is to use a `WebView`. A web view is a UI component which can be embedded into an activity, so it is very easy to integrate into the application and the user is not forced to leave it. However, a web view has several flaws. For once, it has not stored the same cookies as the browser the user uses normally, so he has to log in separately. Additionally, there are many attacks known for it, as outlined in the article "Attacks on WebView in the Android System" by scientists from the Syracuse University in New York, USA[39]. RFC 8252 additionally explicitly discourages to use an embedded user-agent, as these may be able to record keystrokes, can copy session cookies and pose other security and privacy related threats [7].

The third way and the way used in this thesis is the newest one. It is called "Custom Tab". It is best described as a lightweight browser tab that is displayed in the context of the running app, so it opens much faster than a native browser while offering all of its advantages. Additionally, the app opening the custom tab can set some layout elements such as the color and menu entries [28]. Custom tabs are available for android since May, 2016 and Chrome 45. They are available for all android versions since *Jelly Bean*, which is more than 99% of all android devices [18]. Because of that, a fallback solution, which is recommended by Google, has not been implemented. Listing 5.1 shows how to open a URL in a custom tab. As a custom tab basically is a browser, it conforms to the definition of an external user-agent as recommended by RFC 8252.

```
1  import android.content.Context;
2  import android.support.customtabs.CustomTabsIntent;
3
4  String url = "https://jannisfink.de";
5  Context context = getContext();
6
7  CustomTabsIntent.Builder builder = new CustomTabsIntent.Builder();
8  CustomTabsIntent intent = builder.build();
9
10 intent.launchUrl(context, Uri.parse(url));
```

Listing 5.1: Open a URL in a Custom Tab

5.1.2. AsyncTask

Android prohibits the use of network request on the main (UI) thread, because they are executed synchronously. All network requests must be made in a separate thread, so that they do not block the UI thread. The `AsyncTask` API provides a simple way of managing such short living threads. Its use is not limited to network calls, but the

runtime of an async task should not exceed more than a few seconds [13]. The Listing 5.2 shows a minimal example of how to use the simple task. To create a specific async task, one must extend the class `AsyncTask` and define the three generic type parameters. The first one defines the parameter type, the third the return type. The second generic argument is not used in this thesis and will be ignored. When one of these parameters is not used, it should be set to `Void`.

```
import android.os.AsyncTask;

class MyAsyncTask extends AsyncTask<Parameter, Void, Response> {
    @Override
    protected Response doInBackground(Parameter... parameters) {
        Response response = new Response();

        for (Parameter parameter : parameters) {
            // do stuff
        }

        return response;
    }
}
```

Listing 5.2: Minimal AsyncTask example

To execute an async task asynchronous, one must create an instance of the task and call the `execute` method on it. This method expects zero or more parameters that are passed to the `doInBackground` method and returns the `AsyncTask` itself. Its result can be accessed by the `get` method. Listing 5.3 shows this. Additionally, it catches the exceptions that might be thrown during computation.

```
MyAsyncTask task = new MyAsyncTask();

try {
    Response response = task.execute(/* parameters */).get();
} catch (InterruptedException | ExecutionException e) {
    // deal with exceptions
}
```

Listing 5.3: AsyncTask usage

5.2. OAuth keywords

This section gives an overview about the different roles, servers and tokens involved in the OAuth flow [34].

Client The client is the third party mentioned above. This application wants to gain access to the resource owners protected resources.

Resource Owner The resource owner is the role in power to grant the client access to it's protected resources. If this role is filled out by a human, it is also called an end user.

Authorization Server This server issues access token and refresh token, after the resource owner has successfully authenticated itself and granted the client access.

Resource Server This server stores the protected resources. It is able to respond to queries for these resources by checking the validity of the token issued by the authorization server.

Client Identifier / Client ID A public and unique identifier identifying the client at the resource owner.

Client Secret A shared secret between the authorization server and the client. See section 5.3 for further information.

Authorization Grant The authorization grant is a special token issued by the authorization server on behalf of the resource owner with a short lifespan of usually about 10 minutes. With this, the access and optional refresh token can be negotiated.

Access Token This token is issued by the authorizaion server in exchange for the authorization grant. It is used by the client as a replacement for the resource owners credentials. In contrast to the resource owners credentials, it can have a limited scope, for example granting only read access to the client. It can also have a limited livespan.

Refresh Token This token is issued optionally by the authorization server. When it is issued, it can be used by the client to get a new access token and refresh token.

5.3. Obtaining a new access token

Before the client can start the process of obtaining a new access token, he must first be registered at the authorization server. This server then generates a unique id for the client, which has to be used by the client during the communication in order to identify itself. Based on the type of the client, the server may also generate a client secret or client password for the client. It may be used when obtaining the access token, but must

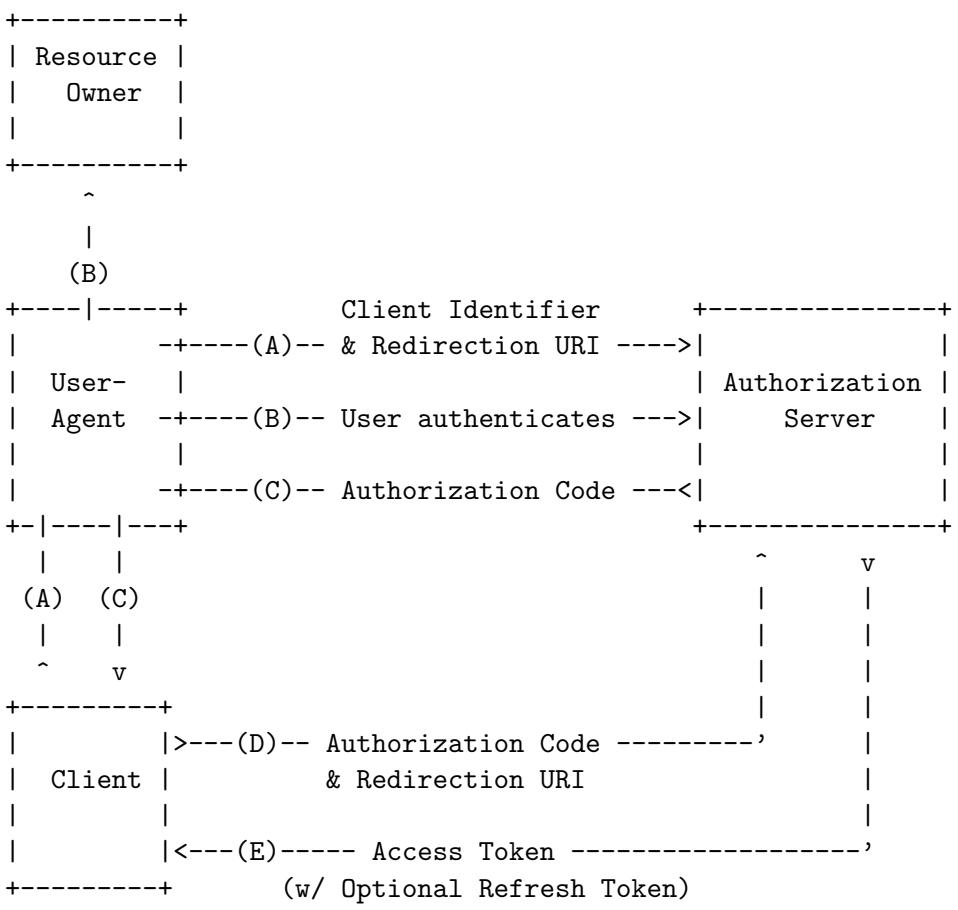

Figure 5.1.: Authorization Code Flow [34].

not be used when obtaining the authorization grant. This secret must only be used when the client is able to keep it confidential, so it is not suitable to be used in an android app, but should be used for web application clients. When using the client password, it should be transferred by using HTTP basic authentication with the client id being the username and the client secret the password. When it is not possible to utilize basic authentication, it is possible for the client to include the parameters in the request body [34].

OAuth specifies several ways to obtain an access token, but this thesis will just cover the authorization code grant, since this grant is recommended for native apps [7]. For this, the OAuth provider must provide two protocol endpoints: the authorization endpoint and the token endpoint. When the client wants to obtain a new access token, it involves both endpoints in two separate steps. The process is shown in Figure 5.1.

First, the client has to get an authorization grant by directing the resource owner to the authorization endpoint (step `A` in Figure 5.1). He has to encode some information into the url parameters in order to successfully obtain an authorization grant. These information are at least `response_type`, which has to be set to the value `code` and `client_id`, which is the id given previously to the client by the authorization server. Additionally, there are two optional parameters: `scope`, which describes the privileges the client wants to have with this token and `redirect_uri`, which is the URL the client wants the resource owner to be redirected after successful authorization or in any error case. When the user/resource owner successfully authorizes the client (step `B` in Figure 5.1), the authorization grant is send as an url parameter to the given redirect url (step `C` in Figure 5.1).

After the client has successfully obtained an authorization grant, it can be used to generate the access token. For this, the client has to make a HTTP POST request to the token endpoint with the following parameters in the request body (step `D` in Figure 5.1):

- `grant_type` with the value set to `authorization_code`
- `code` with the value of the authorization grant from the previous step
- `redirect_uri` with the value being exactly the same as in the previous step
- `client_id` as generated by the resource server

When the request is successful, it returns the access token and an optional refresh token (step `E` in Figure 5.1). The access token then can be used to access the protected resources while the refresh token is used to reissue an expired access token [34].

5.4. API Usage

Similar to the `KeyStoreCredentialSaver` in chapter 4.2, the API developed in this chapter should be easy to use without requiring a broad knowledge about OAuth. Ideally, the same interface as in section 4 (described in chapter 2.1) should be used, with Listing 5.4 showing an example of how to initialize the OAuth client.

```
1 OAuthClient client = new Client();
2
3 client.initialize();
4
5 Object result = client.makeGetRequest("https://some-url.example.com");
```

Listing 5.4: Desired API for the OAUth client

The base class developed for this is called `OAuthClient` and implements this interface. It expects the authorization and token endpoint URLs as the first two parameters, and the client identifier and the scope as the last ones. Because the endpoint URLs are mostly

static, the two OAuth services, Slack and GitLab, which were implemented as examples, have separate subclasses, where those URLs are set automatically. The GitLab client just expects a base URL, since GitLab instances can and are installed on premise. How to use the GitLab client to connect a client to the GitLab instance hosted at the University of Bremen is shown in Listing 5.5. Since GitLab does not support to authenticate clients without a client password, this password is passed to the constructor as well. The access token and optional refresh token that are obtained during the initialization process are stored using the API developed in chapter 4. When they have already been obtained and stored on the device, they are decrypted during initialization and can be used instantly.

```
import java.util.ArrayList;
import de.bremen.uni.jannisfink.androidsecurity.oauth2.GitLabOAuthClient;

ArrayList<String> scope = new ArrayList<>();
scope.add("api");

OAuthClient client = new GitLabOAuthClient(
    "https://gitlab.informatik.uni-bremen.de",
    "the client identifier",
    scope,
    "the client password"
);

client.initialize(activity);
```

Listing 5.5: GitLab OAuth client initialization

How to use the intialized client is shown in Listing 5.6. `OAuthClient` offers several methods to make HTTP calls with all supported request methods (e.g. `GET`, `POST` and `PUT`). These methods take care of serializing the payload as proper JSON, setting the necessary header values and pass the access token within the `Authorization` header. Both GitLab and Slack make use of the bearer token type described in RFC 6750. When implementing this RFC, resource servers must support the access token to be send in the "Authorization" header field with the "Bearer" authentication theme. This means, that the HTTP header "Authorization" has to be set to the value `"Bearer " + accessToken` [37].

```
RequestPayload payload = new RequestPayload();

Response response = client.makeJsonPostRequest("https://gitlab.informatik
    .uni-bremen.de/api/stuff", payload, Response.class);
```

Listing 5.6: How to use the GitLab OAuth client

5.4.1. Internal API structure

These section dives deeper into how the solution of this chapter was implemented.

The only public API provided is the class `OAuthClient`. This classes constructor expects four parameters: the authorization endpoint, the access token endpoint, the client identifier and the scope. For ease of use the scope is expected as an `Arraylist<String>` instead of a simple string with the different scopes being divided by spaces. In the constructor, the code challenge and code verifier explained in section 5.5.1 are calculated here, alongside the `KeyStoreCredentialSaver`, who is being instantiated, but not initialized, yet. Additional data needed for during the OAuth authentication can be passed in as a simple `Map<String, String>` object by calling `setAdditionalData`. As an example, this method is used to pass in the client secret needed for Slack and GitLab. As mentioned in chapter 5.3, this secret should not be used, so its usage has been intentionally been designed to be complicated. An additional error is logged should the client secret be used. Section 5.5.3 explains this further.

When the user calls `initialize` on the `OAuthClient` object, the `CredentialSaver` is being initialized first. After that, the `OAuthClient` tries to access the stored access token. When there is no access token stored, it then initiates the OAuth authorization code flow. For this, an object of type `AuthorizationGrantRequest` is constructed first, which opens the Chrome custom tab for the user to grant his authorization. When the user has successfully granted authorization, access token and refresh token are being requested, which does not require user interaction. Because it is not possible within Android to make a network call on the main thread (as discussed in section 5.1), an object of type `AccessTokenRequest` is constructed as the next step. This object is then passed as a parameter to an instance of the class `GetAccessTokenTask`, which implements the `AsyncTask` mentioned in section 5.1 to fetch the access and optional refresh token. When those tokens are fetched, they are stored within the `KeyStoreCredentialSaver`, so that they can be accessed easily the next time the `OAuthClient` is being initialized, without user interaction.

Figure 5.2 describes this process graphically. The box labeled “fetched” highlights the return path when the access token has already been fetched. Not all function calls have been visualized in order to maintain readability.

5.5. Security concept

When implementing the solution presented in this chapter, the decisions made were based mainly on the RFCs 6749 (“OAuth 2.0”) [34], 6750 (“Bearer Token Usage within OAuth 2.0”) [37], 7636 (“Proof Key for Code Exchange by OAuth Public Clients”) [52] and 8252 (“OAuth 2.0 for Native Apps”) [7]. In this chapter will be explained, what measurements were taken in order to secure the OAuth authorization flow. In order to do so, it first will be explained how the attack works. After that, the way of mitigating the specific attack is set out.

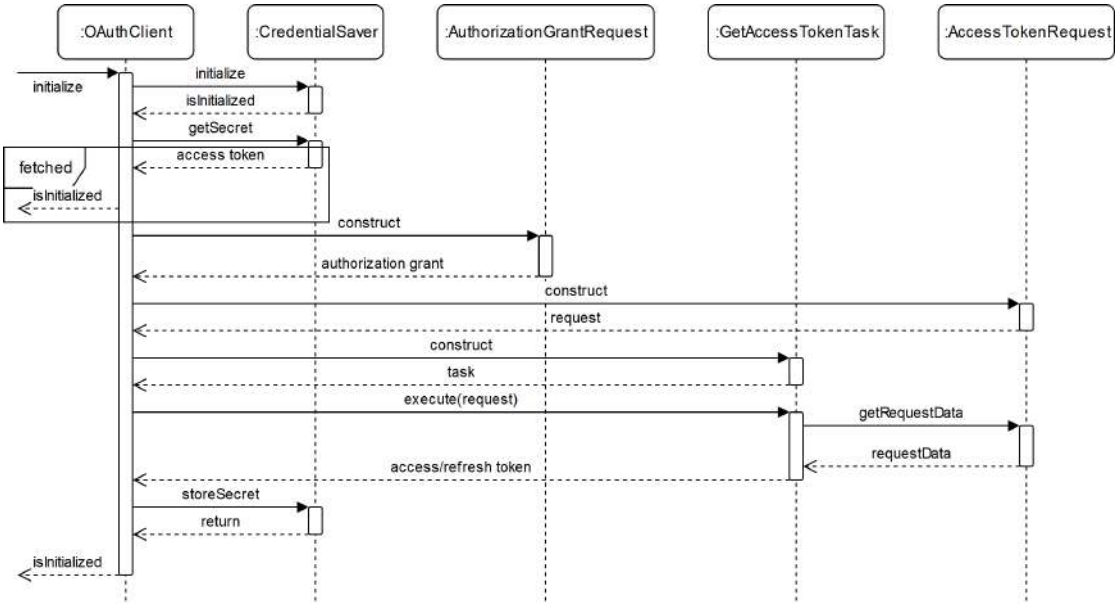

Figure 5.2.: API structure of the `OAuthClient`.

5.5.1. Protect against CSRF

This attack is described in RFC 6749 [34]. With Cross-site request forgery (CSRF), an attacker tries to force an end user to execute unwanted actions on a web application in which they are currently authenticated [47]. A possible attack vector for OAuth would be to trick the user into getting an access token for an account unter the attacker's control. If the user now sends data to the API with this access token, this data is stored in the attackers account instead of his own. As an example, this could include the user's bank account data.

The way the attacker would try to trick the user into using the wrong access token is to trick the user into performing any action that would redirect the user to the callback URL of the OAuth client, with the attackers authorization grant appended as an URL parameter. This action could include clicking on a malicious link on a website under the attackers control. Without protection, the client would process the authorization grant and trade it with the OAuth authorization server for an access token.

To mitigate this attack on the client side, the client generates a random string and appends it as an URL parameter called `"state"` to the authorization request. When the authorization server redirects the user back to the client, it appends this random string to the redirection URL. The client then only accepts the authorization grant, if the `"state"` parameter given as an URL parameter in the redirection URL matches the string previously send to the authorization server. The communication between client and server must be encrypted, so it is very hard for the attacker to gain access to the `"state"` parameter the client choses. As a result, it is not possible for him to trick the client into

accepting his authorization grant. While it should be impossible for the attacker to read the `state` parameter, section 5.5.2 outlines a vulnerability where this is possible.

Because of this and the fact that using the `"state"` parameter for the authorization request is recommended by both RFC 6749 and 8252, the implementation for this chapters solution does that.

5.5.2. Authorization code interception attack

This attack is described in RFC 7636 [52]. With this attack, the attacker tries to intercept the authorization grant that is handed out to the client after the user has authorized him. Clients that use the authorization code grant are vulnerable to this attack. To intercept the code, the attacker exploits the fact that many native apps register a custom scheme for themselves. Examples are for example `mailto:`, which opens the mail client to compose a mail to the given mail adress, `tel:`, which opens the phone app to call the given phone number and many more. Android (as well as IOS) does not prohibit two apps from registering the same custom scheme. When a user has multiple apps installed that have the same custom scheme registered, he gets presented a screen like the one in Figure 5.3. Here he can choose the app that should handle the URL (or in the case of the picture the file type) he tries to open. When the authorization server redirects the user to the callback URL, which might use such a custom scheme, and there are several apps available to handle that scheme, the user might be presented to such a screen where he can choose which app should handle it. When he chooses the attackers app, the attacker gains access to the legitimate clients authorization grant, which he then can use to trade for a valid access token.

The way the OAuth protocol tries to solve this problem is not to prohibit the attacker form getting access to a valid authorization grant, but to prevent him from trading it for an access token. This method is documented in RFC 7636, called "Proof Key for Code Exchange by OAuth Public Clients" [52]. It prevents the attacker from obtaining an access token by utilizing HTTP request parameters. When the (legitimate) client first redirects the user to the authorization server, he includes the parameters `"code_challenge"` and `"code_challenge_method"` as URL parameters. There are two different challenges defined in RFC 7636, but only `"S256"` has been implemented, as it's more secure, the recommended one and mandatory for the server to implement when implementing this RFC. `"plain"` should only be used when it is not possible to use `"S256"`.

To implement the `"S256"` code challenge method, the client must generate two values. The `"code_verifier"` and the `"code_challenge"`. The code challenge is derived from the code verifier, which itself is just a random string. The code verifier should be made up of just ASCII characters. To derive the code challenge from it, it must be hashed using SHA-256 and then be base64 encoded using the URL safe encoding mode. The resulting value is the code challenge. This challenge is then send as an URL parameter with the request to obtain the authorization grant. When the client gets back the authorization grant and wants to make the request to obtain the access token, it puts the code verifier

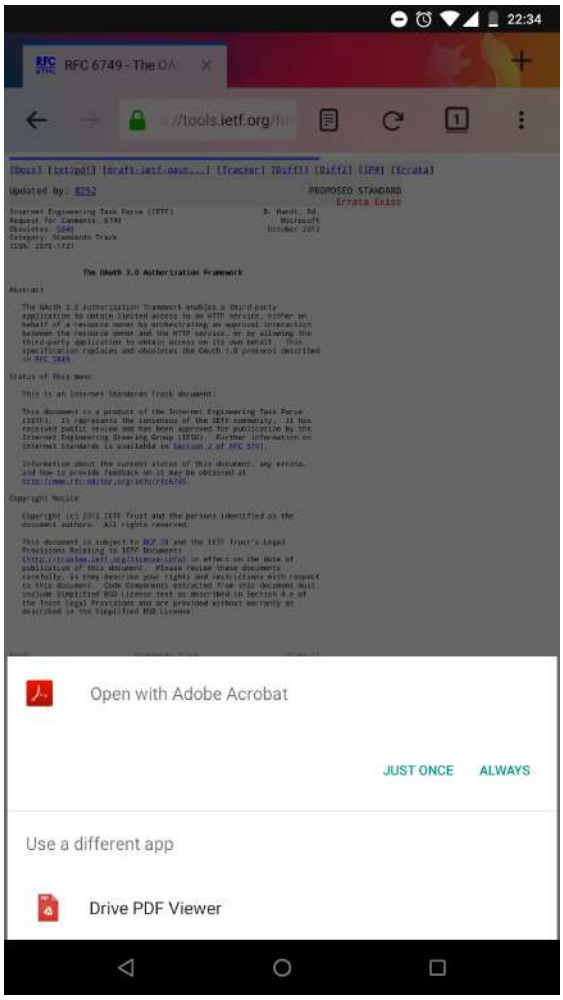

Figure 5.3.: Dialog to select the app that should open a specific file.

with the key `"code_verifier"` in the request body. The server must only grant an access token, when it has verified that the code verifier results in the code challenge when using the same method as the client did before.

The difference between the `"S256"` and `"plain"` challenge method is that with the plain method, the code verifier is the same as the code challenge, so the client sends the same string with both requests. While this method alone already stops the attacker from obtaining an access token when he has intercepted the authorization grant, it does not ensure, that the attacker could not find another way of getting hold of the code challenge. He may be able to do so by reading log files, for example. When the verifier is hashed first, the attacker would have to guess the code verifier, because he would just have access to the hashed one and cannot calculate it from the one has in an easy way. The vulnerability outlined in this paragraph is also a threat to the CSRF attack outlined in section 5.5.1. A possible defence for this is outlined in section 7.1.8.

This protective measure could be implementing without knowing if the authorization server actually supports it. When he does not, he will just ignore the additional request parameters. So code challenge and verifier are always sent along with the authorization grant and access token request.

5.5.3. Complicate the usage of the Client Secret

The client secret involved in the OAuth authorization grant flow must only be used when the client is able to keep it confidential. This is not the case for mobile applications, which are the topic of this thesis. But because the reference servers, Slack and GitLab, are requiring its use, support for it has been implemented. To discourage a developer from using this feature, its API has intentionally been designed cumbersome. Listing 5.7 shows an example of how to use this feature.

```
1 OAuthClient client = new OAuthClient(...parameters);
2
3 Map<String, String> additionalData = new HashMap<>();
4 additionalData.put("dangerous_to_use_client_secret", "secret value");
5
6 client.setAdditionalData(additionalData);
```

Listing 5.7: OAuthClient usage when using a client secret

When using this API, the following error message is logged as well: “Using a client secret is discouraged for mobile applications and should only be considered for web server applications where the client secret can be kept confidential. Try registering your OAuth application as type ’public’ in order to prevent the server to generate a client secret for it. Please refer to RFC 6749, section 2.1 for more information.”.

This way of designing an API for a security relevant feature that should not be used was inspired by the popular Javascript Framework React, which is used for rendering

HTML. A simple example for this framework can be found in Listing 5.8, which also contains a potential security vulnerability React fixes by default. If a developer wants to disable this security feature, he must do so the way that is outlined in Listing 5.9.

```
class TextRenderer extends React.Component {
    render() {
        const user_input = this.props.user_input
        return <p>Hello {user_input}</p>;
    }
}

// <TextRenderer user_input="John"/> renders <p>Hello John</p>

// <TextRenderer user_input="<script>alert('XSS works');</script>"/>
// would render <p>Hello <script>alert('XSS works');</script></p>

// React escapes dangerous characters appropriately. In reality, this
// renders: <p>Hello &lt;script&gt;alert('XSS works');&lt;/script&gt;</p>
```

Listing 5.8: React example containing a potenial XSS vulnerability

```
class TextRenderer extends React.Component {
    render() {
        const user_input = this.props.user_input
        return <p dangerouslySetInnerHTML={{__html: user_input}}>;
    }
}

// <TextRenderer user_input="John"/> renders <p>John</p>

// <TextRenderer user_input="<script>alert('XSS works');</script>"/>
// renders <p><script>alert('XSS works');</script></p>
```

Listing 5.9: React example containing an example of how to disable the security feature

5.5.4. Enforce the use of encrypted HTTPS connections

This solution validates all URLs that are given as parameters. The URLs to access the authorization code and to get the actual access token must use the `https://` scheme. Additionally, the URLs that are given as parameters to the `makeJsonXYZRequest` method of the `OAuthClient` must use this scheme as well. When they are not using encrypted connections, the implementation will fail with a `IllegalArgumentException`.

5.6. Comparison with AppAuth-Android

For this comparison, version `0.7.1` of AppAuth-Android have been used. It can be downloaded from the releases page of the GitHub project[2]. It will be called just "AppAuth" from now on.

Both, the implementation of this chapter and AppAuth are using the best practices set out in RFC 8252. This includes the use of Custom Tabs instead of WebView and the authorization grant scheme in favour of the others.

The main differences of both projects is the amount of work a developer needs to put into the integration of the different projects in order to get them to work. In both cases, the dependency would first have to be declared[3]. After that, it would be just a few lines of code as shown in Listing 5.5 to implement the solution of this chapter. On the other hand, in order to get AppAuth to work, the developer would have to implement the OAuth flow, as is shown in the demo app of the GitHub project[4]. Without diving further into the specific API that a developer would have to use in order to implement this flow using AppAuth, this API requires the developers to know about the specific OAuth flow.

As opposed to AppAuth, this chapters solution requires a few general steps in order to make the OAuth authorization process work:

1. Register a new client at the authorization server.
2. When not already available, create/configure a new subclass of `OAuthClient` for the specific authorization server (this step is mostly for convenience, as this way a user of this subclass would not have to specify the authorization server URLs every time the class is used).
3. Create a new instance of the `OAuthClient` class/subclass specifying the client id and other parameters.
4. Call `initialize`.
5. (Optionally) set a listener object to be notified when the access token has been issued successfully by calling `setOnInitializedListener` with the listener object.

After inspection of all the features that are available in AppAuth, the only feature that is available in AppAuth, but not in this chapters solution, is the Dynamic Client Registration for OAuth defined in RFC 7591. This feature was intentionally not added because the scope of this chapter was just to cover the authorization code grant for granting an access token. However, this feature could be added as an extension to this chapters solution with an API very similar to the existing. Listing 5.10 shows one proposition, which does the client registration and initiates the OAuth authorization code

[2] https://github.com/openid/AppAuth-Android/releases
[3] The solution of this chapter is not available as a Maven package, but is assumed for this compairson.
[4] https://github.com/openid/AppAuth-Android/tree/master/app/java/net/openid/appauthdemo

grant afterwards. While this example would not compile with the current `OAuthClient`, it gives a rough overview of how this goal could be achieved.

```
import de.bremen.uni.jannisfink.androidsecurity.oauth.
    DynamicRegistrationClient;
import de.bremen.uni.jannisfink.androidsecurity.oauth.OAuthClient;
import de.bremen.uni.jannisfink.androidsecurity.SecureContextInitializer;


DynamicRegistrationClient registrationClient = new
    DynamicRegistrationClient(...params);
registrationClient.initialize(...params);
registrationClient.setOnInitializeListener(new SecureContextInitializer.
    OnInitializeListener() {
    @Override
    public void onInitialized(@NonNull SecureContextInitializer
    secureContextInitializer) {
        OAuthClient oauthClient = new OAuthClient(registrationClient.
    getClientId());
        oauthClient.initialize(...params);
        oauthClient.setOnInitializeListener(new SecureContextInitializer.
    OnInitializeListener() {
            // oauthClient is ready to be used
            oauthClient.makeJsonGetRequest(...params);
        });
    }
});
```

Listing 5.10: Proposition for an API to add the Dynamic Client Registration.

6. RTSP 1.0

RTSP/1.0, or real time streaming protocol version 1.0, is a network protocol designed for control over the delivery of data with real-time properties [55]. It is not in charge of the real time data transport itself, but is used to setup and control of the streaming session. While there is already a successor for RTSP/1.0, RTSP/2.0, which has been released several years ago, not many popular RTSP clients or servers have implemented the new version. GStreamer being one of the first to implement it in late 2017 [33], while there has nothing happened with the corresponding ticket for the VLC media player for over four years [59]. Because of the limited support from most popular libraries, this thesis will cover only the old version, RTSP/1.0, and will call it just RTSP.

The desired solution for this chapter is an executable, which can take an unencrypted RTSP stream as input and is able to output an encrypted one. This way, this proxy can easily be added to existing solutions to provide an easy way of migrating to encrypted RTSP. Additionally, the Android VideoView, which can already play unencrypted RTSP, must be extended to support encrypted RTSP, if it cannot do so right now.

Ideally, all that is left to the developer using this solution is to make sure that the unencrypted stream is not accessible from the web any more.

6.1. Protocol components

While RTSP is just in control of the connection setup and controlling the data source, Real Time Protocol (RTP) is usually used for the actual live data streaming. Additionally, Real Time Control Protocol (RTCP) is used to control the RTP stream by primarily providing feedback on the quality of the data distribution[54].

A fourth protocol used inside RTSP is called Session Description Protocol (SDP). Here it is used for interchanging the streaming parameters, such as video and audio codec, framerate, size and other parameters needed to initiate the streaming session.

This section will briefly describe these four protocols.

6.1.1. RSTP

RTSP is a text-based protocol using Transmission Control Protocol (TCP) as its transport layer protocol. It is heavily inspired by HTTP and reuses many of its header names.

In the same way as HTTP does, RTSP reqests and responses are send in plain text with an empty line marking the end of a message.

To begin establishing an RTSP stream between a client and a server, the client could send the RTSP request shown in Listing 6.2 to the server. The RTSP server could respond to that request with the response shown in Listing 6.1. These two messages are real RTSP messages intercepted using Wireshark, a network protocol analyzer. The server responds to the client requesting a description for the media source `"rtsp://184.72.239.149/vod/mp4:BigBuckBunny_175k.mov"` by listing all audio and video sources that make up this media source. SDP is strongly integrated within RTSP and used for the media source description. The client requests the use of SDP for media description by setting the "Accept" header to "application/sdp". The server responds with the response body in SDP format and the "Content-Type" header set to "application/sdp", too.

```
RTSP/1.0 200 OK
CSeq: 2
Server: Wowza Streaming Engine 4.7.5.01 build21752
Cache-Control: no-cache
Expires: Thu, 15 Nov 2018 21:35:02 UTC
Content-Length: 590
Content-Base: rtsp://184.72.239.149/vod/mp4:BigBuckBunny_175k.mov/
Date: Thu, 15 Nov 2018 21:35:02 UTC
Content-Type: application/sdp
Session: 2075693970;timeout=60

v=0
o=- 2075693970 2075693970 IN IP4 184.72.239.149
s=BigBuckBunny_175k.mov
c=IN IP4 184.72.239.149
t=0 0
a=sdplang:en
a=range:npt=0- 596.458
a=control:*
m=audio 0 RTP/AVP 96
a=rtpmap:96 mpeg4-generic/48000/2
a=fmtp:96 profile-level-id=1;mode=AAC-hbr;sizelength=13;indexlength=3;
    indexdeltalength=3;config=1190
a=control:trackID=1
m=video 0 RTP/AVP 97
a=rtpmap:97 H264/90000
a=fmtp:97 packetization-mode=1;profile-level-id=42C01E;sprop-parameter-
    sets=Z0LAHtkDxWhAAAADAEAAAAwDxYuS,aMuMsg==
a=cliprect:0,0,160,240
a=framesize:97 240-160
a=framerate:24.0
a=control:trackID=2
```

Listing 6.1: RTSP DESCRIBE response

```
DESCRIBE rtsp://184.72.239.149/vod/mp4:BigBuckBunny_175k.mov RTSP/1.0
CSeq: 2
User-Agent: ProxyRTSPClient (LIVE555 Streaming Media v2018.10.17)
Accept: application/sdp
```

Listing 6.2: RTSP DESCRIBE request

6.1.2. SDP

Listing 6.1 shows an example of what SDP looks like in the response body. The SDP message format consists of a single character followed by an equal sign and the value for that character. An SDP message has to be read from top to bottom. SDP can describe multiple media sources. Each media source is identified with the character `m` with all lines coming after the line (up to the next character `m` or the end of the message) containing this letter describing the media source further. For example, the SDP message contained in the Listing mentioned above describes a video source with the string `m=video 0 RTP/AVP 97` with `video` being the media type, `0` being the port on the server, `RTP/AVP` the media protocol and `97` describing the media format. The number just stands for "dynamic", meaning that the correct media format is described in another line. The next line in this SDP message contains this information, encoded as `a=rtpmap:97 H264/90000`, mapping the dynamic type 97 to the media type `H264`.

In this message, the port for both the audio and the video source is `0`, which is not possible. The reason for that is that in this example it is being negotiated using an RTSP `SETUP` request with the ports sent back in the `Transport` header field of the RTSP response, an additional way of negotiating the RTP ports.

6.1.3. RTP / RTCP

RTP (Real Time protocol) is a protocol often used in combination with RTSP. While RTSP is used for the setup and controlling the stream, RTP is used for the actual data of the media stream. It typically uses User Datagram Protocol (UDP) as its transport layer protocol, but can work with other transport protocols as well. Figure 6.2 shows a Secure Real Time Protocol (SRTP) packet, which differs only slightly from an RTP packet. The header contains very few information, other than `timestamp` and `sequence number`, which are present for restoring packet order, the most notable fields are `synchronization source identifier` and `contributing source identifiers`. These identifiers can be used to identify the sources of the stream. The `synchronization source` is the source of the actual video stream, while the (optional) `contributing sources` can be used to signal the session partners that the `synchronization source` is proxying the data from another source.

RTCP is a sister protocol of RTP used for various features within the RTP data distribution. Its main feature is to provide information about the quality of the data distribution.

No.	Description
1	The RTSP `DESCRIBE` request. This request is shown in Listing 6.2. The client requests information for a specific RTSP URL.
2	The server responds with an RTSP `OK`. The RTSP response contains the media information encoded with SDP. Listing 6.1 shows this response in detail.
3	The client sends an RTSP `SETUP` request for the first media source. This request contains the URL for the media source the client wants to set up. This URL was given to the client with the response from point 2 in the line beginning with `a=control:`. This value is a URL relative to the URL given in the `Content-Base` response header.
4	The server responds with RTSP `OK`. The `Transport` response header contains information about the protocol to use as well as the client and server ports.
5	The client sends an RTSP `SETUP` request for the second media source. See point 3 for further information.
6	The server responds with RTSP `OK`. See point 4 for further information.

Table 6.1.: Table explaining Figure 6.1

There are four different package types, with "Sender report" and "Receiver report" being the two packages that are responsible for informing all stream participants about the number of messages they have send to or received from everyone, alongside other information. While the original RFC assumed that RTP and RTCP packets are send over different channels, RFC 5761 describes a way of multiplexing both packet types over a single port [46], in order to reduce the number of ports allocated on both client and server. When multiplexing is not used, the RTP conntection typically uses an even numbered port and the RTCP connection the next higher one [54]. The example featured in Listing 6.1 does not use this multiplexing feature, as its use must be signalled by the phrase `a=rtcp-mux` in the SDP message.

6.1.4. RTSP setup example using SDP

Figure 6.1 shows an example of the requests and responses that are needed in order to set up a media streaming session. The steps 1 to 6 that are numbered in this image are explained further in table 6.1.

6.2. Setup of a testing environment

In order to inspect and eventually encrypt the traffic from the RTSP server to a client, it was necessary to have an RTSP server to work with first. This section contains every software tried during the search for one that either supports every secure protocol needed or could be expanded to do so.

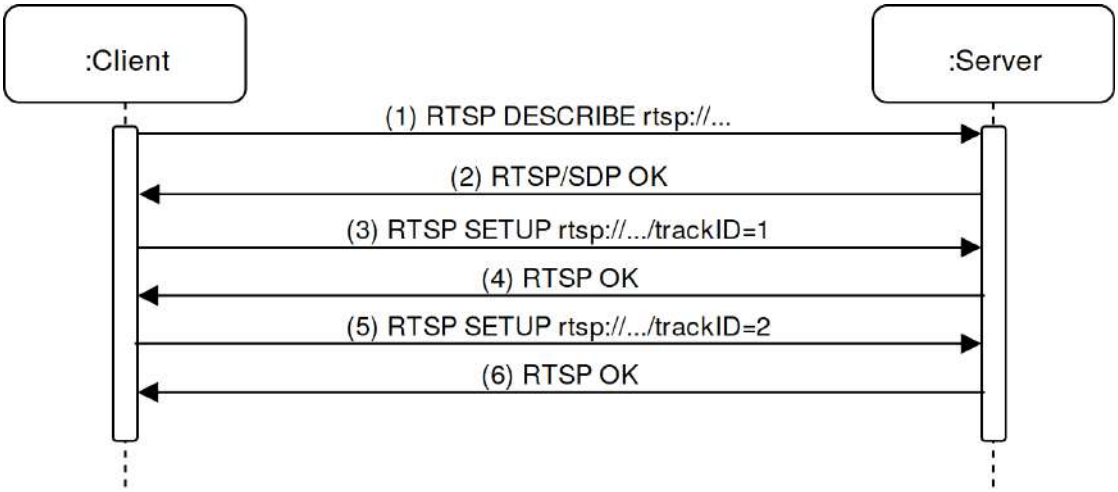

Figure 6.1.: RTSP setup.

6.2.1. ffserver

`ffserver` is an RTSP server build on top of `ffmpeg`, which have been tried out first. The results of that can be found in section 6.2.2. Because of the RTSP server being unreliable even for unencrypted streams and the support for `ffserver` being dropped with the next major release of `ffmpeg` (4.0), it was decided not to use this program.

6.2.2. ffmpeg

`ffmpeg` is a popular library used for primarily for video and audio transcoding. It also offers an RTSP client. While it can encode and decode RTP streams, it is not a fully featured RTSP server. As it is not easy to implement all features of an RTSP server, it was decided to not use this on its own, but to use an additional library for the RTSP server component.

6.2.3. live555

`live555` was eventually used as the library for the RTSP server. Its RTSP client is widely used within well known projects such as VLC Player [60]. Out of the box, it already comes with an RTSP proxy server, which can be extended to support the protocols necessary for an encrypted streaming session. Right now, it does not support neither encrypted RTP nor RTSP, so both must be implemented in order to fully encrypt the streaming session. While the project has these tasks on its roadmap, they have not been implemented, yet [38].

This library was extended to intercept the RTP and RTCP packages that are proxied over this server. The next step of the implementation on the server side would be to encrypt these packages using `ffmpeg`, for example. However, since there is much more

to do in order to encrypt the packages, this implementation has not been extended further. Section 6.3 describes the steps needed in order to achieve this goal. The current implementation can be found on the CD that is handed in with this thesis.

6.3. How to encrypt an RTSP stream

When encrypting an entire RTSP stream, one must encrypt all communication channels that make up that stream. While RTSP does not define the actual protocol to use for data transport, RTP is often used. As a result, this section will focus on how to encrypt both RTSP and RTP/RTCP. All protocols are explained in section 6.1. Because there are many different ways for achieving this resulting in a big complexity, this section will describe only one way.

6.3.1. RTSP

Even with extensive research, there does not seem to exist a standard solution for encrypting RTSP messages between clients and server for the version of RTSP that is used in this thesis (RTSP/1.0). There seem to exist some custom support for it within some popular projects such as Wowza [61], but they require an extensive setup. The current version of RTSP, RTSP/2.0, does feature an "RTSP over TLS" feature, which follows the same guidelines as HTTP does with regard to TLS [53].

Without an encrypted RTSP communication, it is very hard to prevent anyone from accessing the protected stream. Authentication within RTSP is done via the `WWW-Authenticate` header [55]. Here, the authentication credentials are send as a specific type, with `Basic` being a very common one, which just encodes username and password using base64 [49], resulting in the users credentials being send basically as clear text.

So in order to fully protect the stream, the first step must be to implement RTSP/2.0 in order to utilize the "RTSP over TLS" feature, which is mandatory to implement when implementing RTSP/2.0.

6.3.2. RTP/RTCP

RTP and its sister protocol RTCP both have a encrypted version which is defined in RFC 3711 [4]. Figure 6.2 shows how an SRTP packet is specified. While SRTP defines both encryption and authentication, it is possible to use just one of them. The RFC states that, when using both, encryption should be applied before authentication. When using just the encryption and not sending the optional master key identifier (labeled SRTP MKI in Figure 6.2), the SRTP package does not look any different than a regular RTP package with only the payload being encrypted.

While SRTP just has optional and recommended fields added to its packet layout, there are three mandatory and one optional field added to Secure Real Time Control Protocol

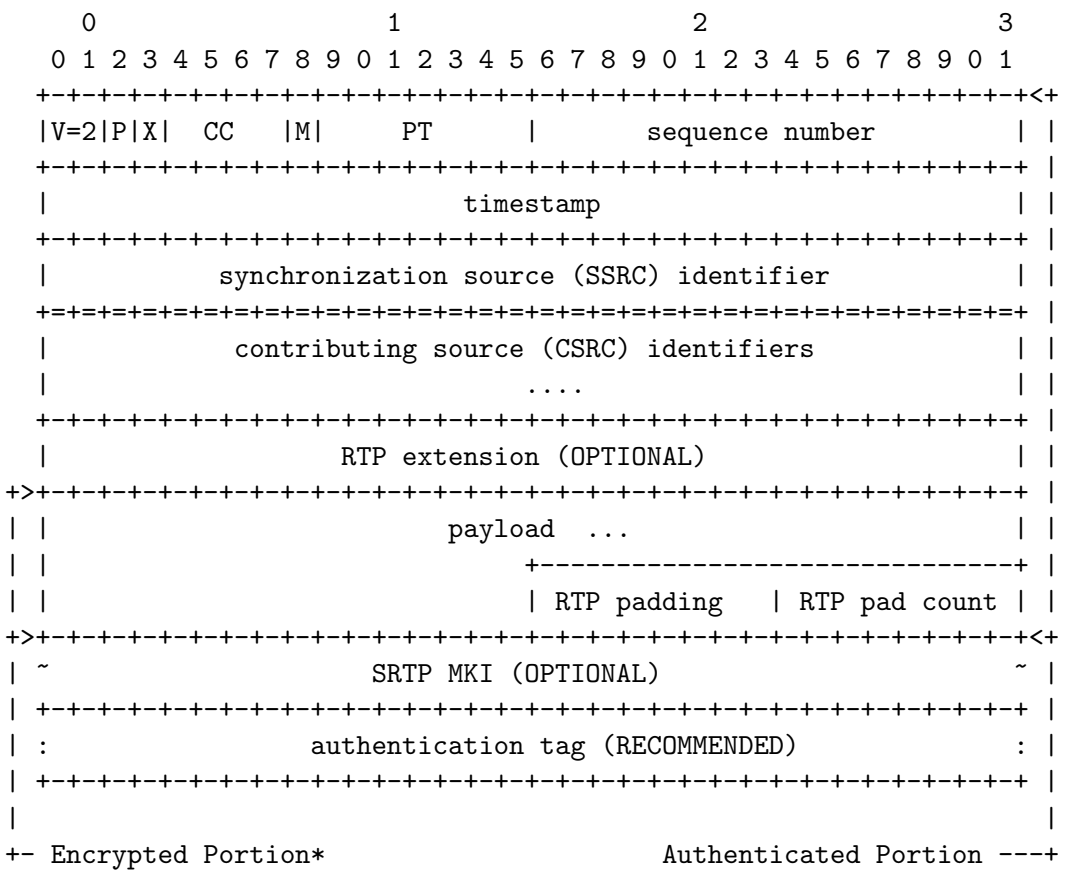

Figure 6.2.: SRTP packet illustration [4].

(SRTCP). Those three mandatory fields are a one bit long "encryption flag", the SRTCP index and the authentication tag. While authentication is recommended for SRTP, it is mandatory for SRTCP. The optional field is the master key identifier, which is analog to SRTP. Figure 6.3 shows how an SRTCP packet is specified.

RFC 3711 predefines three different ciphers, with the `NULL` cipher meaning no encryption, so it is ignored in this thesis. The two remaining ciphers are AES in f8 or Counter Mode. When implementing SRTP, the german Federal Office for Information Security does not recommend a specific one but instead states that both are okay to use [35]. For the message integrity, it recommends using SHA-2 instead of SHA-1 with the hash-based message authentication code (HMAC) being reduced to 80 bits in size. The example in section 6.3.3 uses SHA-1, as `ffmpeg` does not support SHA-2 at the time of writing.

6.3.3. Encrypt an RTP stream using ffmpeg

`ffmpeg` can be used to encrypt an RTP stream. While this library does not feature an RTSP server, the RTP stream can be directly opened using `ffmpeg`'s command line

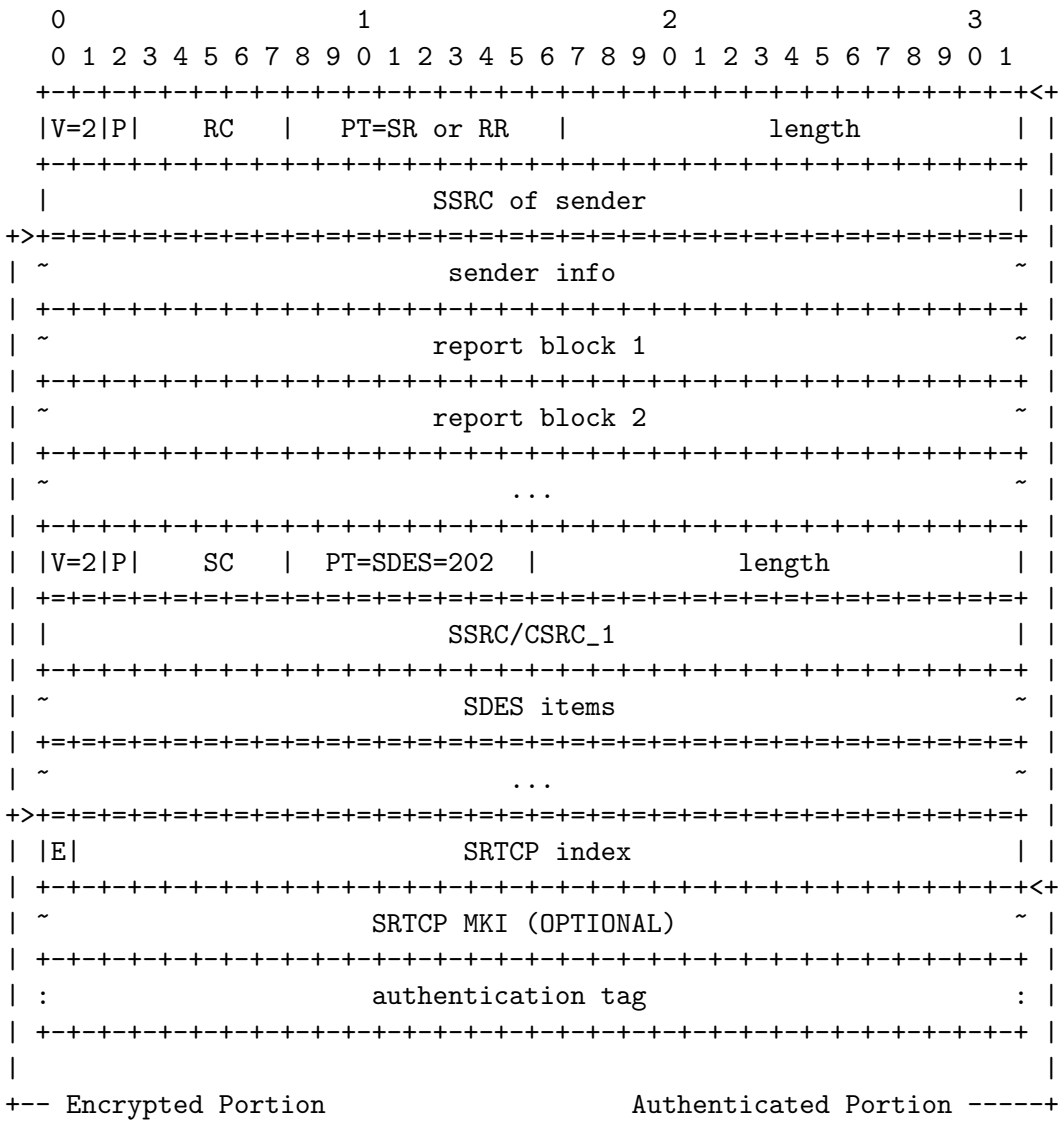

Figure 6.3.: SRTCP Sender Report packet illustration [4].

utilities. Listing 6.3 shows how to encrypt any input accepted by `ffmpeg` and write it to the output given as last parameter. In this Listing, `$INPUT` specifies the input. While it can be any video or audio file on the file system, one can also specify an RTSP URL here. By doing so, this command acts as a partial proxy encrypting the RTP stream. In this example, `ffmpeg` writes onto port 1337 of the local interface. So in order to read the SRTP stream that is written, one must start a separate `ffmpeg` utility program called `ffplay`, which will open a separate window in which the streamed video will be shown. Listing 6.4 shows how to achieve that.

```
ffmpeg -i $INPUT -f rtp_mpegts -srtp_out_suite
    SRTP_AES128_CM_HMAC_SHA1_80 -srtp_out_params
    012345678901234567890123456789012345678901234567890123456789 srtp://127.0.0.1:1337
```

Listing 6.3: Write an SRTP stream

```
ffplay -srtp_in_suite SRTP_AES128_CM_HMAC_SHA1_80 -srtp_in_params
    012345678901234567890123456789012345678901234567890123456789 srtp://127.0.0.1:1337
```

Listing 6.4: Read an SRTP stream

The two mandatory parameters `ffmpeg` needs in order to encrypt the data using SRTP are the cipher suite and the "encoding parameters", as `ffmpeg` calls them. The encoding parameters are a base64 encoded binary block with a length of 30 bytes. The first 16 bytes are used as master key with the remaining 14 bytes being used as master salt [57]. Because of the base64 encoding, an input with a size of 40 bytes is needed which then results in a binary block with a size of 30 bytes after decoding. The "encoding parameters" are the command line arguments `-srtp_in_params` for the encryption part and `-srtp_out_params` for the decryption part. They must match in order for the encryption/decryption and authentication to work properly. However, when deploying into a production environment, these parameters should not be hard coded, for obvious reasons. The original RFC for SRTP does not define any key exchange mechanism, but the protocol has since been extended to support different ones, which is described in section 6.3.4.

The second mandatory parameter is the cipher suite to use for encryption and decryption named `-srtp_in_suite` or `-srtp_out_suite`, respectively. In order for the encryption and decryption to work, they have to match, too. The examples are using `SRTP_AES128_CM_HMAC_SHA1_80`, which instructs `ffmpeg` to use AES in Counter Mode with a 128 bit long key and to use SHA-1 for message authentication. The key to use for encryption and decryption is generated from the master key using a deterministic key derivation function [4].

6.3.4. Key exchange for the RTP streams

While there are different ways of implementing the key exchange for the RTP streams (e.g. the use of Datagram Transport Layer Security (DTLS) [41]), the BSI recommends using MIKEY [35]. MIKEY is short for Multimedia Internet Keyring, which aims to provide a key management solution for multicast scenarios [3]. It does not reinvent the wheel, but instead aims to integrate existing key exchange solutions within the protocols, such as SDP, that are already in use in the RTSP ecosystem.

When it comes to a specific key transport and exchange method, the BSI does not prefer one over the others [35]. RFC 3830 defines the use of a pre-shared key, public key encryption, or Diffie-Hellman (DH) for the key exchange, but mentions that one advantage of DH over the others is the ability to provide perfect forward secrecy (PFS) [3]. With PFS, past communications remain relatively secure even when a certificate that is used for the initial handshake is compromised, which is why this section will focus on this for the exchange method. One disadvantage of comparison to the other exchange method is the fact that it cannot be used in a multicast scenario, because DH is strictly designed for the key exchange between two parties. This is accepted for this scenario.

One big disadvantage of all three key exchange methods is the missing scalability. This holds especially true for the pre-shared key, as new devices cannot easily be integrated into the network without having to add the pre-shared key to every existing device in the network the new device is supposed to communicate with. Because of that, the BSI additionally recommends using a public-key infrastructure (PKI) for DH and the public key encryption. This way, those two key exchange methods will scale [35]. This chapter will assume that there is a working PKI and that there are private and public key available ready to use. There exists one caveat with this approach. Other than HTTPS, where usually just the server needs a certificate [50], here both the client and the server need one, as they are both singing their DH key [3].

MIKEY packets are made up of separate payloads. The first payload in each MIKEY packet is the Common Header payload, whose structure is described in Figure 6.4. Each payload references the next one by utilizing the `next payload` header field, where it describes the next payloads type. This header field is located at different positions for different payload types. The most notable payload types are listed below. Their explicit structure is not listed, as it is not relevant for understanding how MIKEY works.

timestamp (T) This payload contains a timestamp in order to prevent replay attacks.

DH data payload (DH) This payload includes the data needed for the DH key exchange.

Signature payload (SIGN) This payload always is the last one. It contains the signature for the whole MIKEY message.

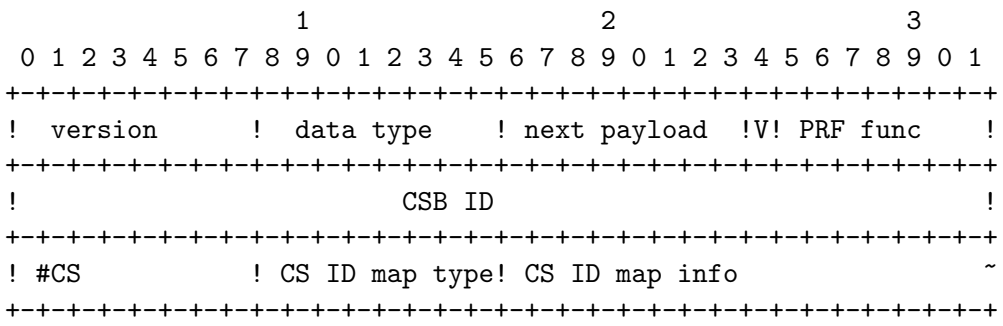

Figure 6.4.: MIKEY HDR payload [3].

Certificate Payload (CERT) This payload contains certificate data. If one wants to transmit a certificate chain, those separate certificates should be transmitted in separate payloads.

If one wants to integrate MIKEY into SDP so the key exchange can happen during the RTSP setup, an additional standard is needed, which describes a key management extension for SDP and RTSP, called RFC 4567 [2]. The MIKEY message for itself is just a binary stream. In order to send it inside of SDP, it needs to be text based. This is achieved by encoding the binary data using base64. The resulting string is then appended to the SDP message with the prefix `a=key-mgmt:mikey` to signal the use of MIKEY for key management.

6.3.5. Build an Android client that can decode the encrypted media stream

The stock Android VideoView is currently unable to decode encrypted RTP/RTCP [1]. As a consequence of this, a developer would have to implement a media player that is able to do that.

To ease the implementation, some libraries that already provide partial support could be used here, such as `ffmpeg`. Since this library is written in C, the developer might utilize the Android Native Development Kit (NDK) here [12]. Alternatively, he could implement a new codec and add it to Android through the OS update process [23].

6.4. Security considerations

This section lists security relevant information that should be taken into account when implementing an encrypted RTSP stream. This chapter lists what should be done in order to protect the stream. The previous chapters at this point listed what was done in order to improve security or protect the user, but since this chapter does not have a working implementation, it lists important aspects that need the attention of a potential developer implementing these.

6.4.1. Side channels

Especially when transmitting secure video or audio data, particular attention should be paid to minimizing the development of side channels [35]. For example, computer scientists from the John Hopkins University in Baltimore have found a vulnerability when using a variable bit rate for audio encoding [62]. The scientists measured the bandwidth consumed by an RTP stream over time and were able to guess certain keywords based on the bandwidth usage pattern. This is possible, because with a variable bit rate, lower tones can be transmitting using a lower bit rate, resulting in a lower bandwidth consumption.

A possible workaround for this would be to use a constant bit rate for audio encoding, choosing one that is preferred for the application. This results in an overall higher, but constant bandwidth usage.

6.4.2. Choose a random sequence number

The sequence number within the RTP header is not encrypted when using SRTP, because just the RTP payload is being encrypted, while both header and payload are authenticated. According to RFC 3550, the sequence number must be incremented for each new RTP packet, but should be random in order to make known-plaintext attacks on the encryption more difficult [54].

6.4.3. Sign the MIKEY message when using DH

While DH is a secure protocol for establishing keys over an insecure connection, it is vulnerable to Man-in-the-Middle attacks, as it contains no authentication. So, in order to prevent this attack, a MIKEY SIGN payload should be added with the public key of the sender signing the MIKEY message. The receiver should then verify the signature in order to detect a possible attack.

An alternativ to using the MIKEY SIGN payload is described in RFC 4650. The key management scheme described in this standard has the advantage of providing authentication for the DH keys without relying on a working PKI [29]. However, since section 6.3 already established that pre-shared keys do not scale as well as a working PKI, this idea has not been pursued any further.

7. Conclusions

Off of three security patterns investigated in this thesis, two of them could be implemented as prototypes. For them to be used in production environments, they would have to be subjected to more manual tests and automated unit tests to ensure code quality and stability. While the solution of chapter 4 can probably be used without any customizing, the one of chapter 5 might need more subclasses of the `OAuthClient` class in order to support more backends. Subclasses for GitLab and Slack have been implemented, being called `SlackOAuthClient` and `GitLabOAuthClient`, respectively.

While they keystore could eventually be used to store the generated key pair, there have been some difficulties at the beginning. There are many tutorials and blog posts to find on the internet, most of them are using the `KeyPairGeneratorSpec` (section 4.1.3) to create a key pair. This class is deprecated and should therefore not be used. Because the available algorithms, paddings and other features vary from device to device, most of the examples that used the newer `KeyGenParameterSpec` did not work, too. The main task when configuring the `KeyGenParameterSpec` was to Figure out a combination for those parameters and the provider that was eventually used for encryption and decryption that worked. The result of that can be found on the CD that is handed in with this thesis.

The solution that was planned for chapter 5 could be realized without many issues. The main issue was the fact that many OAuth providers would support the old version, OAuth 1.0, without clarifying that they do not support the newer version and calling it just "OAuth". A prominent example for this would be twitter [58]. This caused some confusion in the beginning, since the docs would not follow the flow described by the RFC, but after this confusion could be sorted out, everything went according to plan. The final solution for this chapter can be found on the CD that is handed in with this thesis, too.

RTSP has presented the biggest challenges. While there are many RTSP client implementations, there are just a few server implementations freely available. All the existing implementations that have been examined during this thesis lack the full support for encrypted RTSP. While `live555` could be modified so that the RTP and RTCP packets that were proxied could be looked at, their encryption has not been added, since a key management solution in order to exchange the encryption keys is necessary, too. As a result of this, no working prototype could be implemented for this chapter. The only result is a manual of what needs to be done to support encrypted RTSP.

7.1. Outlook

While the solutions that were developed for this thesis have been build to be as securely as possible, there are several ways of improving and extending them. This section will take a look on some of them.

7.1.1. Enhanced protection for the key pair

Right now, the only mechanism that prevents an unauthorized party from accessing the key in the keystore is the keystore itself. It does not allow an app to access a key that has not stored it there [26]. An additional way of protecting the key would be a password. The most secure way would be a password that the user of the app has to manually type in every time the app wants to access the key when it has not been already loaded. A less intrusive way would be to generate a fixed password from the static information available on the device, such as International Mobile Equipment Identity (IMEI), phone number, Medium access control (MAC) adresses and other. While this would provide a better user experience, it would not be really secure, as the key could then be accessed by decompiling the source code and take a look at how the password is generated. A third way would be to require some biometric data from the user, such as a fingerprint, but since not every phone has the required sensors build in, this idea has not been developed any further. The easiest way of enhancing the protection for the key pair would be to use a password chosen by the user which he must enter in order for the application to get access to the key. This solution has the danger that the user forgets the password he has chosen. Since there would be no way of recovering it, the key pair would remain locked and the secrets unable to be encrypted.

7.1.2. Signing and verification

The data to store securely is encrypted using the public key of the key pair. This approach works fine when the goal is just to prevent the secrets from theft. However, it cannot detect tampering with the stored secrets. Since the public key does not need to be kept secret, an attacker could gain access to it and replace the encrypted secret on disk without the application noticing it. An easy way to prevent this would be to sign the data using RSA mechanisms and verify the signature every time the secrets are accessed.

7.1.3. Deal with changes to the encryption and decryption algorithms used

The algorithm, block mode and padding are state of the art and secure as of the time of writing, but this fact will most certainly change in the future. At the moment, this implementation provides no mechanism to change these parameters while being able to access older secrets already stored using the old algorithm. To mitigate this problem, the implementation could be extended with a way of detecting which encryption parameters

were used. When these parameters are not the most recent ones, the implementation should decrypt the secrets using the old parameters, encrypt it using the most recent ones and write the secret with the updated encryption back.

7.1.4. A more stable way of generating a unique alias

For the sake of simplicity, the alias used to store the key pair and the shared preferences is the canonical class name of the activity given as a parameter to the `initialize` function. This way, it is not possible to rename or move this activity without losing access to the stored secrets. A more reliable way of generating a unique alias would be using the application id of the Android app, but since this is implemented as an Android library, there is no easy way of getting the application id from the app from within this library, so this problem has not been adressed any further, as it is not a part of the actual problem to solve.

7.1.5. Check the certificate validity

As the certificate that is stored inside of the Keystore is a X.509 certificate, it can have a validity period. To set the validity period, it is possible to set start and end date when building the `KeyGenParameterSpec`, which is explained in section 4.1.3. However, these parameters are optional and currently not used. The certificates validity is not checked during encryption or decryption. Because there could be a long time between encryption and decryption, the certificate may become invalid. The `KeyStoreCredentialSaver` would have to store all certificates ever created in order to ensure that an encrypted secret can always be decrypted. After that, he would have to ensure that it is encrypted using the latest valid certificate. To do so, he could use a mechanism similar to the one outlined in section 7.1.3.

Because even the public key is not publicly accessible, this measure to further improve the security has not been taken, because when an attacker can access the public key to start guessing the private one, he most certainly has access to the private key as well.

7.1.6. Deal with expired access tokens

An access token may have a limited lifespan. This this can be detected by the `"expires_in "` key in the JSON response of the token request. The value for this key is an integer, which describes the number of seconds the access token is valid from the time it has been issued. When that time is over and the client has been issued a refresh token as well, this refresh token could be used at the token endpoint to request a new access token (and an optionally new refresh token).

7.1.7. Implement other token types

The solution implemented for chapter 5 only provides support for the bearer token type. When using a different token type than this, any request for a protected resource will fail with an `IllegalArgumentException`. The only other mechanism that is recommended by the OAuth RFC is `"mac"`. However, the draft that is in development is currently in version five, which has expired on July 19, 2014 [51]. Because of that, this token type has not been implemented.

7.1.8. Further protection against CSRF

The `"state"` parameter used for CSRF protection in section 5.5.1 is vulnerable to an attack outlined in section 5.5.2. Here, the attacker is able to intercept the authorization grant as well as accessing the URL the client redirected the user to to authenticate him. The attacker can now try to trick the user into clicking on a link that redirects him back to the OAuth client with the correct state, but another authorization grant that will result in the client getting an access token for a account under the attackers control. This attack is another form of CSRF.

To mitigate the attack, it is possible to use a mechanism similar to the one described in section 5.5.2. When the authorization server returns the authorization grant, he could send a `"code_challenge"` along with it. When the autorization grant is exchanged for an access token, the authorization server would then send the `"code_verifier"` along with the access token. The client could then compare the code challenge with the verifier to verify that the access token he got is the one he actually wanted.

This protective measure has not been implemented, because it is not a part of standard OAuth. Additionally, the probability for such an attack is low. The attacker would have to gain access to the state in the URL that is called to get the authorization grant and make the user click on a malicious link before he got the correct authorization grant back from the authorization server. So, a protective measure at this point might just not be necessary.

7.1.9. Use Android App links to prevent Authentication grant interception

Android App Links are links with the `HTTP` scheme. An app developer can configure a specific domain to be opened with the app developed by him. To do so, he must claim ownership of this domain through one of the website association methods provided by Google. Those Android App Links offer the advantage that they cannot be used by another app if the developer has claimed ownership. This way, those links prevent the authorization grant interception attack outlined in section 5.5.2 [20].

This protective measurement has not been implemented, but could be with a few steps. Right now, the callback URL is hardcoded to be `"oauthschemecallback://path"`. This could be set via parameter when instantiating the OAuth client. To set the Activity that is

going to be launched when the user is redirected to the callback URL, the developer must implement this activity inside of the apps package, as he has to configure it in the Android manifest file. The implementation of this activity is no big deal, though, as it can be a simple subclass of the activity that is dealing with it in the provided solution of this chapter, `AuthorizationGrantConsumerActivity`.

7.1.10. Use HLS for live streaming

HTTP Live Streaming (HLS) is currently in the process of being standardized [45], but already supported by Android [1]. As the name suggests, it offers live streaming of video and audio data over the HTTP protocol. Since HTTP is used, its way of encrypting data, called "HTTP over TLS", specified in RFC 2818 can be used [50]. This protocol is used by millions of servers to encrypt their HTTP traffic and is much easier to deploy than secure RTSP, which uses several different protocols and channels for the actual data transport, where HTTP just uses one. Additionally, there exist tools such as SSLLab's SSL Server Test[1], which can test any server for weaknesses in its HTTPS deployment and LetsEncrypt[2], which offers free certificates and an automated installer.

While there are some major advantages when using HLS instead of RTSP, HLS has some flaws, too. At first, it is a pure TCP protocol, whereas RTSP can use UDP for the actual data transport. UDP as a protocol is more suited for the transport of live data, because it does fewer error checking and retransmissions in an error case. Since it does not offer reliable transmission, it does not check, if a packet has arrived at the recipient. This makes the data transport much faster, albeit missing some packets sometimes. Additionally to this, one mayor feature of RTSP is missing in HLS, the ability to control the data source (playback, pause, seek). While this can be implemented using HTTP, too, there is currently no standardized way to do so.

[1] https://www.ssllabs.com/ssltest/
[2] https://letsencrypt.org/

A. Appendix

A.1. DVD with Source Code

Glossary

AES Advanced Encryption Standard.

API Application programming interface.

ART Android Runtime.

BSI German Federal Office for Information Security.

CSRF Cross-site request forgery.

DH Diffie-Hellman.

DNS Domain Name System.

DTLS Datagram Transport Layer Security.

HAL Hardware Abstraction Layer.

HLS HTTP Live Streaming.

HMAC hash-based message authentication code.

HTTP Hypertext Transport Protocol.

IMEI International Mobile Equipment Identity.

IoT Internet of Things.

MAC Medium access control.

NDK Android Native Development Kit.

PFS perfect forward secrecy.

PKI public-key infrastructure.

RFC Request for Comments.

RSA Rivest–Shamir–Adleman, a public-key cryptosystem.

RTCP Real Time Control Protocol.

RTP Real Time Protocol.

RTSP Real Time Streaming Protocol.

SDK Software development kit.

SDP Session Description Protocol.

SRTCP Secure Real Time Control Protocol.

SRTP Secure Real Time Protocol.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TZI Center for Computing Technologies.

UDP User Datagram Protocol.

UI user interface.

XML Extensible Markup Language.

List of Figures

List of Tables

Listings

Bibliography

[1] Android Developers. *Supported media formats*. 2019. URL: https://developer.android.com/guide/topics/media/media-formats (visited on 01/08/2019).

[2] J. Arkko et al. *Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)*. RFC 4567. Network Working Group, July 2006. URL: https://tools.ietf.org/pdf/rfc4567.pdf.

[3] J. Arkko et al. *MIKEY: Multimedia Internet KEYing*. RFC 3830. Network Working Group, Aug. 2004. URL: https://tools.ietf.org/pdf/rfc3830.pdf.

[4] M. Baugher et al. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. Network Working Group, Mar. 2004. URL: https://tools.ietf.org/pdf/rfc3711.pdf.

[5] Dieter Bohn. *Amazon says 100 million Alexa devices have been sold — what's next?* Jan. 2019. URL: https://www.theverge.com/2019/1/4/18168565/amazon-alexa-devices-how-many-sold-number-100-million-dave-limp (visited on 01/10/2019).

[6] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. *Analysis of Secure Key Storage Solutions on Android*. Tech. rep. Institute for Computing and Information Sciences; Radboud University Nijmegen, Dec. 2014.

[7] W. Denniss and J. Bradley. *OAuth 2.0 for Native Apps*. BCP 212. Internet Engineering Task Force (IETF), Oct. 2017. URL: https://tools.ietf.org/pdf/rfc8252.pdf.

[8] Android Developers. *<manifest>*. 2018. URL: https://developer.android.com/guide/topics/manifest/manifest-element (visited on 11/04/2018).

[9] Android Developers. *AccountManager*. 2018. URL: https://developer.android.com/reference/android/accounts/AccountManager (visited on 03/12/2018).

[10] Android Developers. *Activity*. 2018. URL: https://developer.android.com/reference/android/app/Activity (visited on 11/01/2018).

[11] Android Developers. *Android keystore system*. 2018. URL: https://developer.android.com/training/articles/keystore (visited on 08/05/2018).

[12] Android Developers. *Android NDK*. 2019. URL: https://developer.android.com/ndk/ (visited on 01/15/2019).

[13] Android Developers. *AsyncTask*. 2018. URL: https://developer.android.com/reference/android/os/AsyncTask (visited on 06/30/2018).

[14] Android Developers. *Bundle*. 2018. URL: https://developer.android.com/reference/android/os/Bundle (visited on 03/12/2018).

[15] Android Developers. *Context*. 2018. URL: https://developer.android.com/reference/android/content/Context (visited on 11/01/2018).

[16] Android Developers. *Context#MODE_PRIVATE*. 2018. URL: https://developer.android.com/reference/android/content/Context#MODE_PRIVATE (visited on 08/10/2018).

[17] Android Developers. *Create a custom account type*. 2018. URL: https://developer.android.com/training/id-auth/custom_auth (visited on 12/14/2018).

[18] Android Developers. *Distribution dashboard*. 2018. URL: https://developer.android.com/about/dashboards/ (visited on 11/22/2018).

[19] Android Developers. *Extraction prevention*. 2018. URL: https://developer.android.com/training/articles/keystore#ExtractionPrevention (visited on 08/05/2018).

[20] Android Developers. *Handling Android App Links*. 2018. URL: https://developer.android.com/training/app-links/ (visited on 11/23/2018).

[21] Android Developers. *Hardware security module*. 2018. URL: https://developer.android.com/training/articles/keystore#HardwareSecurityModule (visited on 08/05/2018).

[22] Android Developers. *KeyGenParameterSpec.Builder*. 2019. URL: https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder (visited on 01/03/2019).

[23] Android Developers. *Media*. 2019. URL: https://source.android.com/devices/media.html (visited on 01/15/2019).

[24] Android Developers. *Permissions overview*. 2018. URL: https://developer.android.com/guide/topics/permissions/overview (visited on 08/15/2018).

[25] Android Developers. *Platform Architecture*. 2018. URL: https://developer.android.com/guide/platform/ (visited on 07/20/2018).

[26] Android Developers. *Security features*. 2018. URL: https://developer.android.com/training/articles/keystore#SecurityFeatures (visited on 08/05/2018).

[27] Android Developers. *SharedPreferences*. 2018. URL: https://developer.android.com/reference/android/content/SharedPreferences (visited on 06/03/2018).

[28] Chrome Developers. *Chrome Custom Tabs*. 2018. URL: https://developer.chrome.com/multidevice/android/customtabs (visited on 06/10/2018).

[29] M. Euchner. *HMAC-Authenticated Diffie-Hellman for Multimedia Internet KEYing (MIKEY)*. RFC 4650. Network Working Group, Sept. 2006. URL: https://tools.ietf.org/pdf/rfc4650.pdf.

[30] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[31] Gartner, Inc. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. Feb. 2019. URL: https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016 (visited on 01/03/2109).

[32] Gartner, Inc. *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020*. Dec. 2013. URL: https://www.gartner.com/newsroom/id/2636073 (visited on 01/03/2109).

[33] GStreamer. *GStreamer on Twitter*. 2019. URL: https://twitter.com/gstreamer/status/915998393989779456 (visited on 01/03/2019).

[34] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Internet Engineering Task Force (IETF), Oct. 2012. URL: https://tools.ietf.org/pdf/rfc6749.pdf.

[35] Federal Office for Information Security. *BSI TR-02102-1; Cryptographic Mechanisms: Recommendations and Key Lengths*. Tech. rep. Federal Office for Information Security, 2018.

[36] *IoT - Explore - Google Trends*. 2019. URL: https://trends.google.com/trends/explore?date=all&q=iot (visited on 01/05/2019).

[37] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. Internet Engineering Task Force (IETF), Oct. 2012. URL: https://tools.ietf.org/pdf/rfc6750.pdf.

[38] Live Networks, Inc. *LIVE555 Streaming Media*. 2019. URL: http://www.live555.com/ (visited on 01/02/2019).

[39] Tongbo Luo et al. *Attacks on WebView in the Android System*. Tech. rep. Dept. of Electrical Engineering & Computer Science, Syracuse University Syracuse, New York, USA, 2011.

[40] Cara McGoogan. *Unprecedented cyber attack takes Liberia's entire internet down*. 2016. URL: https://www.telegraph.co.uk/technology/2016/11/04/unprecedented-cyber-attack-takes-liberias-entire-internet-down/ (visited on 01/13/2019).

[41] D. McGrew and E. Rescorla. *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*. RFC 5764. Internet Engineering Task Force (IETF), May 2010. URL: https://tools.ietf.org/pdf/rfc5764.pdf.

[42] Nest Labs. *Nest — Create a Connected Home*. 2019. URL: https://nest.com/ (visited on 01/15/2019).

[43] New Jersey Cybersecurity & Communications Integration Cell. *Mirai*. 2016. URL: https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet (visited on 01/13/2019).

[44] Tobias Osmers. "Sicherheitsanalyse der TLS-Client-Implementierung von Android-Anwendungen bezüglich ihrer Kommunikation mit einem Gerät im lokalen Netz." Bachelor's Thesis. University of Bremen, Oct. 2018.

[45] R. Pantos and W. May. *HTTP Live Streaming*. RFC 8216. Independent Submission, Aug. 2017. URL: https://tools.ietf.org/html/rfc8216.pdf.

[46] C. Perkins and M. Westerlund. *Multiplexing RTP Data and Control Packets on a Single Port*. RFC 5761. Internet Engineering Task Force (IETF), Apr. 2010. URL: https://tools.ietf.org/pdf/rfc5761.pdf.

[47] The Open Web Application Security Project. *Cross-Site Request Forgery (CSRF)*. 2018. URL: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF) (visited on 11/15/2018).

[48] The Open Web Application Security Project. *OWASP Mobile Security Project*. 2017. URL: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#Top_10_Mobile_Risks (visited on 05/07/2018).

[49] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Internet Engineering Task Force (IETF), Sept. 2015. URL: https://tools.ietf.org/pdf/rfc7617.pdf.

[50] E. Rescorla. *HTTP Over TLS*. RFC 2818. Network Working Group, May 2000. URL: https://tools.ietf.org/pdf/rfc2818.pdf.

[51] J. Richer et al. *OAuth 2.0 Message Authentication Code (MAC) Tokens*. Tech. rep. Internet Engineering Task Force, 2014.

[52] N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. Internet Engineering Task Force (IETF), Sept. 2015. URL: https://tools.ietf.org/pdf/rfc7636.pdf.

[53] H. Schulzrinne et al. *Real-Time Streaming Protocol Version 2.0*. RFC 7826. Internet Engineering Task Force (IETF), Dec. 2016. URL: https://tools.ietf.org/html/rfc7826.pdf.

[54] H. Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. STD 64. Network Working Group, July 2003. URL: https://tools.ietf.org/pdf/rfc3550.pdf.

[55] Henning Schulzrinne, Anup Rao, and Robert Lanphier. *Real Time Streaming Protocol (RTSP)*. RFC 2326. Network Working Group, Apr. 1998. URL: https://tools.ietf.org/pdf/rfc2326.pdf.

[56] *smart home - Explore - Google Trends*. 2019. URL: https://trends.google.com/trends/explore?date=all&q=smart%20home (visited on 01/05/2019).

[57] The FFmpeg developers. *24.32 srtp*. 2019. URL: https://www.ffmpeg.org/ffmpeg-all.html#srtp (visited on 01/04/2019).

[58] Twitter, Inc. *Authentication*. 2018. URL: https://developer.twitter.com/en/docs/basics/authentication/overview/oauth.html (visited on 06/06/2018).

[59] VideoLAN Project. *#441 (RTSP/2.0 support) - VLC*. 2019. URL: https://trac.videolan.org/vlc/ticket/441 (visited on 01/03/2019).

[60] VideoLAN Project. *Live555 - VideoLAN Wiki*. 2019. URL: https://wiki.videolan.org/Live555/ (visited on 01/02/2019).

[61] Wowza Media Systems, LLC. *Get SSL certificates from the Wowza Streaming Engine StreamLock service*. 2019. URL: https://www.wowza.com/docs/how-to-get-ssl-certificates-from-the-streamlock-service (visited on 01/03/2019).

[62] Charles V. Wright et al. *Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations*. Tech. rep. Johns Hopkins University, Department of Computer Science; Baltimore, 2018.