Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Design of an Android App2App Redirect Flow for the FAPI 2.0 Standard

Miles Stötzner

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Ralf Küsters
Betreuer/in:	Dr. Guido Schmitz
Beginn am:	June 1, 2020
Beendet am:	December 1, 2020

Abstract

OAuth 2.0 Authorization Framework (OAuth 2.0) is an authorization framework used to grant third parties access to resources. OpenID Financial-grade API 2.0 (FAPI 2.0) is a profile for OAuth 2.0 with the goal to reach the security requirements of the financial sector. These requirements contain for example the assumption that an access token might leak to an attacker and that some endpoints are misconfigured due to social engineering attacks.

We present a design proposal for a redirect flow for FAPI 2.0 between two Android applications, the client and the auth app. A typical case of usage would be a financial wallet application, the client, that redirects the user to a banking app, the auth app, in order to authorize a financial transaction.

Our main goal is to securely redirect the user between the client and the auth app using today's technologies. We require integrity, confidentiality, source authentication and target authentication when redirecting the user. Roughly speaking, this means that the user is redirected from the correct source app to the correct target app and that no attacker is able to read or write the sent data. The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback. Authentication is based on comparing package signing certificates. The motivation for a secured redirect is to mitigate attacks as soon as possible as a defense-in-depth. The secured redirect can not only be applied to OAuth 2.0 but can be used to secure other scenarios.

Our proposal further defines the registration process for clients and auth apps. Considering this, we present the OAuth Integrity Attestation which ensures that only the correct applications running on an untampered device can register and that generated keys are hardware-backed. The OAuth Integrity Attestation contains e.g. a SafetyNet attestation and key attestations. Furthermore, we define the communication between the auth app and the corresponding backend, the authorization server, for interoperability, and security reasons.

To show the feasibility of our proposal we implemented the advanced profile in the context of a digital wallet app and a banking app. A user is able to link his bank account and to authorize financial transactions. Additionally, we implemented a malicious app that attacks the user redirect.

We discuss the security of our proposal with respect to our attacker model and list identified vulnerabilities. Our attacker model is based on the attacker model defined by FAPI 2.0 and extended by multiple assumptions and attacker capabilities. The additional attacker capabilities include e.g. that the client uses a misconfigured auth app and that the auth app might have some misconfigured endpoints. The motivation for these attacker capabilities are social engineering attacks. We also mitigate known problems with FAPI 1.0 that also apply to FAPI 2.0. One of the identified vulnerabilities is that a physical attacker with knowledge of the device credentials can access the same resources which a client has access to.

Kurzfassung

OAuth 2.0 Authorization Framework (OAuth 2.0) ist ein Authorizierungs-Framework, das dazu verwendet wird, um Dritten Zugriff auf Resourcen zu geben. OpenID Financial-grade API 2.0 (FAPI 2.0) ist ein Profil für OAuth 2.0 mit dem Ziel die Sicherheit zu gewährleisten, die im Finanzsektor gefordert ist. Zu den Anforderungen gehören zum Beispiel, dass Zugriffstoken in den Besitz eines Angreifers gelangen können und dass, aufgrund von Social Engineering Angriffen, falsche URLs verwendet werden.

Wir päsentieren in dieser Arbeit unseren Vorschlag für FAPI 2.0, bei dem der Nutzer von einer Android Applikation, dem Klient, zu einer weiteren Android Applikation, der Auth App, weitergeleitet wird und wieder zurück.

Unser Hauptziel ist es den Nutzer sicher weiterzuleiten. Das beinhaltet, dass der Nutzer von der korrekten Ursprungsapplikation zur korrekten Zielapplikation weitergeleitet wird und dass kein Angreifer den Inhalt ausgetauschter Information lesen oder ändern kann. Wir erreichen dieses Ziel, indem die Applikationen sich gegenseitig authentifizieren und ein Antwortchannel registriert wird. Eine Applikation kann authentifiziert werden, indem die Zertifikate überprüft werden, mit der die Applikation signiert wurde. Wir merken an, dass diese abgesicherte Weiterleitung auch in anderen Szenarien angewendet werden kann.

Wir spezifizieren den Registrierungsprozess des Kleints und der Auth App und sichern diesen mithilfe der sogenannten OAuth Integrity Attestation ab. Diese stellt sicher, dass nur korrekte Applikationen auf einem nicht kompromitierten Gerät sich registrieren können. Darüber hinaus spezifizieren wir die Kommunikation zwischen der Auth App und dem zugehörigen Server.

Wir implementieren unseren Vorschlag im Kontext einer Finanz-App und einer Bank-App. Ein Nutzer kann seinen Bankaccount verlinken und Überweisungen tätigen.

Wir diskutieren die Sicherheit unseres Vorschlags im Hinblick auf unser Angreifer-Modell, das auf FAPI 2.0 basiert, und führen verschiedene Schwachstellen auf. Wir gehen zum Beispiel zusätzlich davon aus, dass der Client die falsche Auth App oder dass die Auth App falsche URLs verwendet. Eine der identifizierten Schwachstellen ist, dass ein Angreifer, der Kenntniss der Zugangsdaten zum Gerät hat, auf die gleichen Resourcen zugreifen kann wie auch der Client.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADB Android Debug Bridge. 29

APK Android Package. 23

ART Android Runtime. 19

AS Authorization Server. 54

CDD Android Compatibility Definition Document. 20

CIBA Client Initiated Backchannel Authentication. 45

CSRF Cross-Site Request Forgery. 36

CTS Android Compatibility Test Suite. 20

DNSSEC Domain Name System Security Extensions. 48

FAPI 2.0 OpenID Financial-grade API 2.0. 17

FBE File-Based Encryption. 21

HAL Hardware Abstraction Layer. 19

HSTS HTTP Strict Transport Security. 48

IPC Inter-Process Communication. 18

JAR JWT Secured Authorization Request. 41

JARM JWT Secured Authorization Response Mode. 41

JWE JSON Web Encryption. 39

JWKS JSON Web Key Set. 40

JWS JSON Web Signature. 30

JWT JSON Web Token. 37

MASVS Mobile App Security Verification Standard. 32

MSTG Mobile Security Testing Guide. 32

mTLS Mutual TLS Authentication. 44

OAuth 2.0 OAuth 2.0 Authorization Framework. 17

OIDC OpenID Connect. 35

OS Mobile Operating System. 19

PAR	OAuth 2.0 Pushed Authorization Requests. 42
PKCE	Proof Key for Code Exchange. 43
RAR	OAuth 2.0 Rich Authorization Requests. 42
RS	Resource Server. 54
SE	Secure Environment. 19
SELinux	Security-Enhanced Linux. 19
TEE	Trusted Execution Environment. 19
TLS	Transport Layer Security. 48
TPM	Trusted Platform Module. 19
UID	User ID. 19
WebAuthn	Web Authentication. 113

1 Introduction

OAuth 2.0 Authorization Framework (OAuth 2.0) is an authorization framework used to give third parties access to resources. OpenID Financial-grade API 2.0 (FAPI 2.0) is a profile for OAuth 2.0 with the goal to reach the security requirements of the financial sector. These requirements contain for example the assumption that an access token might leak to an attacker and that some endpoints are misconfigured due to social engineering attacks.

OAuth 2.0 is usually used inside the browser. The user is redirected between two websites which exchange information using the URL. In this thesis we present a design proposal for a redirect flow for FAPI 2.0 between two Android applications, the client and the auth app. A typical case of usage would be a financial wallet application that redirects the user to a banking app in order to authorize a financial transaction. In this example the financial wallet application would be the client and the banking app would be the auth app.

Our main goal is to securely redirect the user between the client and the auth app using today's technologies. We require integrity, confidentiality, source authentication, and target authentication when redirecting the user. Roughly speaking, this means that the user is redirected from the correct source app to the correct target app and that no attacker is able to read or write the sent data. The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback. Authentication is based on comparing package signing certificates.

Additionally to the previous assumptions, we assume that the client uses a misconfigured auth app and that the auth app might have some misconfigured endpoints. These additional assumptions are based on social engineering attacks.

Our proposal further defines the registration process for clients and auth apps. The client registers dynamically as confidential OAuth 2.0 client at the authorization server. The auth app prompts the user for his credentials and registers at the authorization server. We present the oauth integrity attestation which ensures that only the correct applications running on an untampered device can register and that generated keys are hardware-backed. The oauth integrity attestation contains e.g. a SafetyNet attestation and key attestations. Furthermore, we define the communication between the auth app and the corresponding backend, the authorization server, for interoperability and security reasons.

To show the feasibility of our proposal we implemented the advanced profile in the context of an digital wallet app and a banking app. A user is able to link his bank account with the digital wallet app and to authorize financial transactions. Additionally, we implemented a malicious banking app that attacks the user redirect.

At the end we discuss the security of our proposal and list identified problems. The security discussion includes our attacker model and a discussion concerning a secure redirect. Our attacker model is based on the attacker model defined by FAPI 2.0 and extended by multiple assumptions and attacker capabilities. The additional attacker capabilities include e.g. that the client uses a misconfigured auth app and that the auth app might have some misconfigured endpoints. The motivation for these attacker capabilities are social engineering attacks. We also mitigate known problems with FAPI 1.0 that also apply to FAPI 2.0. Our discussion concerning a secure redirect can be applied to any user redirect on Android and is not restricted to OAuth 2.0. We also discuss redirects concerning the web, e.g. when the user is redirected from an application to a browser and back. One of the identified vulnerabilities is that a physical attacker with knowledge of the device credentials can access the same resources which a client has access to.

Related Work

OAuth 2.0 for Native Apps [DB17] defines best practices for the scenario where the client is a native application and the user is redirected to a web browser. In comparison, our proposal redirects the user to another application, the auth app.

The Android Platform Security Model [MSBK19] defines a security model for Android and briefly analyses the security. This paper is not related to OAuth 2.0 or redirecting the user. We exclusively make use of their security model when defining our attacker model.

“Analyzing Inter-Application Communication in Android” [CFGW11] analyses common security vulnerabilities when using Inter-Process Communication (IPC) between Android applications. These include e.g. that a malicious application might intercept a user redirect. The paper includes secure communication guidelines which are not sufficient concerning our requirements.

“OAuth Demystified for Mobile Application Developers” [CPC+14] presents a study of OAuth 2.0 implementations on mobile environments. The study lists existing redirection possibilities and analyses their security. The presented secure way of redirecting a user includes application authentication and a result callback. Application authentication is based on verifying the signature of the application package and is used to ensure that the user is redirected back to the client. The result callback can be used to securely redirect the user back to the client. We utilize both concepts. In comparison, we require mutual application authentication during the redirect and present a complete flow including registration and backend communication.

2 Android

In this section, we take a look at fundamental concepts of Android 10 resp. sdkVersion 29. We give an overview of the Android platform and native applications including the application runtime, IPC, and reverse engineering. Furthermore, we describe available security mechanisms including root detection, application sandbox, and hardware-backed security.

2.1 Platform

In this section, we describe the Android platform. We first provide an overview of the software stack and then explain several security mechanisms, e.g. user data encryption, Security-Enhanced Linux (SELinux), and hardware-backed security.

2.1.1 Overview

In the following, we give an overview of the Android platform by describing the software stack. The stack is as follows [Andag] [Ande]:

- System and user applications
- Java API framework
- Android Runtime (ART) and native C/C++ libraries
- Hardware Abstraction Layer (HAL)
- Linux kernel and security elements

Android is a Mobile Operating System (OS) based on a Linux kernel. The Linux kernel provides multiple functionalities and security features. For example, each application is assigned a unique User ID (UID) for isolating applications (see subsection 2.2.4). For improving the security some devices contain security elements like a Secure Environment (SE), Trusted Platform Module (TPM) and Trusted Execution Environment (TEE).

The HAL provides a generic interface to abstract from the underlying hardware. This enables to support various different types of hardware. For example, HAL provides access to the camera or audio.

An application that is e.g. written in Java is compiled into DEX files. The ART compiles the DEX files into native code. Each application runs as own process within an own ART instance [Andag] [Andh].

The platform contains native C/C++ libraries which are e.g. used by HAL. These can be accessed by applications using a Java API.

There are two types of applications: system and user applications. System applications are applications installed by the system. These include e.g. an email, SMS, or calendar application. User applications are applications installed by the user (see section 2.2). These include e.g. messaging or banking applications.

The Android Compatibility Definition Document (CDD) [Andm] sets requirements for a device to be compatible with Android. These include that the system installed on the device must pass Android Compatibility Test Suite (CTS).

The CTS [Andn] is used to test the functionality of the system. The system is required to pass CTS, otherwise the installed system is not considered to be Android [MSBK19].

2.1.2 Bootloader

The *bootloader* [Ele14] is the process that initializes hardware modules and starts the operating system.

The bootloader allows installing a custom operating system on the device. This is done by flashing the custom operating system. A user might install an Android version on a device, that does not officially support that particular Android version. One reason might be that the device is too old.

Flashing the operating system is only possible if the bootloader is unlocked. Unlocking the bootloader can be done by e.g. running the command *fastboot flashing unlock* which reboots the device in *fastboot* mode. The user is prompted to give consent by pressing a physical button. Android recommends a factory data reset when unlocking the bootloader [Andac]. This means especially that all user data and the memory are deleted to protect user data from a malicious flashed system.

The bootloader can be locked again. This can be done by e.g. running the command *fastboot flashing lock*. The user is prompted to give consent by pressing a physical button.

The system might be tampered e.g. by an attacker having physical access to the device disk. *Verified boot* [Ele14] ensures that an untampered operating system is loaded. The bootloader verifies the integrity and authenticity of each booting state and establishes a trust chain from the hardware up to the system. The root of the trust chain, the *root of trust*, is e.g. a public key burned into the device during manufacturing. If the verification of a booting stage fails then the device can not boot. It is possible to set a *user-settable root of trust* to use verified boot with a custom operating system.

2.1.3 Encryption

Android encrypts all user data. Whenever a file is read/ written the file is automatically decrypted/ encrypted.

Older Android versions use full-disk encryption [Andx]. Full-disk encryption uses a single key, that is available after the device is unlocked. One disadvantage of this concept is that the core functionality of the system is not available until the user enters his password.

With Android 10 File-Based Encryption (FBE) is required [Andv]. File-based encryption uses different keys for different files. Applications can be configured to be runnable before the device is unlocked by using storage that is available before the user unlocks his device. This enables e.g. that the alarm clock application can run directly after booting the device.

2.1.4 SELinux

Linux uses a discretionary access control system which is based on a concept of ownership. In such a system a user with access to a resource can grant other users access to the resource. This concept is in general coarse-grained and might lead to undesired resource accesses [Andal].

SELinux [Andal] is a mandatory access control system. A mandatory access control system grants a user access to a resource based on policies that can not be changed by the user. In SELinux all resources have a label. The label is used to determine if an action is allowed. This concept allows fine-grained policies and does not allow to pass the access to a resource to another user. SELinux is used to further improve the sandbox (see subsection 2.2.4).

2.1.5 User Authentication

The device can be protected by enabling a lockscreen. The user is locally authenticated before unlocking the device.

Android makes use of a three-tiered authentication model [CMCL] [Andm]. The primary tier is based on something the user knows, e.g. a PIN or pattern. This tier can always be used to unlock the device [Andm]. The secondary tier is based on something the user is, e.g. fingerprint or face recognition. The tertiary tier is based on something the user has, e.g. Bluetooth headphones. In subsection 7.6.2 we briefly discuss their security.

User authentication refers to Android authenticating the device user based on the mentioned tiers and is also called *local authentication*. We refer with *device credentials* to the primary tier and with *biometrics* to the secondary tier.

An application uses a *biometric prompt* to locally authenticate a user [Dama]. The biometric prompt displays a uniform UI provided by Android. As the name suggests the prompt is used to authenticate the user using biometrics but it is possible to configure the prompt to allow device credentials.

The local authentication should be enforced at a remote endpoint or be based on a cryptographic operation. Otherwise, the authentication can be bypass. For example, local authentication can be used to get access to a hardware-backed key that decrypts application data [Muea].

2.1.6 Android Keystore

The *Android Keystore* [Andg] can be used to store, generate, and use cryptographic keys. All operations are performed within an isolated container. The keystore prevents unauthorized key access including prevention of extraction. An application does not have direct access to keys. The application instructs the keystore to perform a specific cryptographic operation using a specific key and receives the result. In general, only the application that generated the key is authorized to use the key. Additionally, if the application is uninstalled the respective keys are also removed.

The isolated container is either implemented as software or hardware-backed [Andz]. For example, *StrongBox* [Andg] is a hardware-backed implementation. Note, each key is bound to the root of trust used for verified boot [Andf].

The key can require user authentication. In such a case the user is required to be locally authenticated. Access can be granted for a validity duration or a single cryptographic operation. If a validity duration has been set then the user can be authenticated using either device credentials or biometrics. Biometric authentication is required if the key access is only granted for a single cryptographic operation.

The key can be configured to be bound to currently registered biometrics. This means that the key is invalidated if a new biometric is registered, e.g. a new fingerprint.

The key can require user confirmation. For example, if the user is authenticated using face recognition, authentication only succeeds if the user confirms the authentication by clicking on a button and not by simply looking at the screen.

Biometric user authentication can be used to confirm critical operations, e.g. a financial transaction [Andam]. When the user logs into the application, the application generates a private key that is protected by biometrics and registers the public key at the backend. When a financial transaction should be done the application shows a biometric prompt. The transaction details can be displayed in the prompt description. After the user authenticates, the keystore signs transaction data using the private key. The backend can ensure that the user confirmed the request by verifying the signature.

Key Attestation [Andaq] is a feature to attest key properties. Key attestation is a certificate that holds information about the key, the keystore, the device, and a challenge. The challenge is passed to the keystore when the attested key is generated and can not be changed later. The certificate is signed by a key that is embedded in the keystore. The attestation might contain non-resettable device identifiers. These are not available for third-party applications [Andak]

2.1.7 Android Protected Confirmation

Android Protected Confirmation [Andah] is used by an application to display a confirmation request to the user using a hardware-backed trusted UI. The system does not have control over the UI but only over a text that is shown to the user. After the user confirms the request by pressing a physical button a signature over the displayed text is issued. This feature can be used e.g. for confirming a financial transaction. Note, that only user presence is tested and that the user is not authenticated.

In comparison to the biometric prompt of the Android Keystore, the user knows what is signed when using Android Protected Confirmation.

2.1.8 Web Capabilities

A user can not only visit a website by using a browser but also if the website is e.g. embedded inside of an application. In the following, we compare available capabilities concerning the following scenario: The user is currently inside of an application that wants to display a website to the user.

One approach is to redirect the user to a *browser application*. The advantage of using a browser is that common web security mechanisms are implemented [DB17]. The disadvantage is that the user leaves the context of the application.

A *web view* [SM14] embeds the website inside an application. The advantage is that the user does not leave the context of the application. The disadvantage is that the application has full control over the web view. This means that the application can e.g. read any passwords that the user enters on the website [DB17]. A web view should only be used if the website is under the control of the application developer [Andc].

A *custom tab* [Andp] is the combination of the browser and a web view: A custom tab starts a browser tab in the context of the application while the security of the browser is provided. This means especially that the application can not e.g. read passwords that the user enters on the website. Additionally, the custom tab has access to e.g. the cookies of the browser.

A *trusted web activity* [Andan] is basically a custom tab in fullscreen mode with ownership verification. Android verifies that the application is allowed to load the website inside of a trusted web activity based on digital asset links (see subsection 2.2.5). Trusted web activities are intended to be used in first-party scenarios.

2.2 Application

An application is distributed as Android Package (APK) which is installed on a device. Each application is assigned a unique UID, runs in its own process, and is isolated from other applications. In the following, we further describe an application concerning e.g. the package, the components, and the sandbox. An application is also called a *native application*.

2.2.1 Package

An application is distributed as APK [Andi] using an app store or other sources, e.g. the web. The APK contains the manifest, source code, libraries, configuration files, and additional resources. The *manifest* declares the components of the application and describes the package.

Each package has a *package name*, e.g. “de.milesstoetzner.pandawallet”, and a *version code*, e.g. “1”. The package name identifies the application and is locally unique on the device. If the package name changes then the application is treated as another application. Information about installed packages can be retrieved from the *package manager* [Andae].

Android requires that the APK is signed by one or multiple signing certificates [Andk], so-called *package signing certificates*. The signature is e.g. used during an update: Only if the new APK is signed by the same certificates then the application is updated [Ele14]. There are several APK signature schemes. The following concept is used for schemes version 2 and 3.

The APK is signed by each certificate by signing the hash of the content of the APK. The resulting signature is stored together with the certificate inside the APK. To verify a signature each signature and certificate is extracted from the APK and verified against the modified APK. In general, the signing certificates are self-signed and are trusted by default.

2.2.2 Components

An application consists of several components. A component is an entry point to the application and is declared in the manifest [Andi]. If a component shall be used by another application then the component must be exported.

An *activity* is a graphical component that displays the user interface. For example, the landing page of the application is an activity that displays a login screen and processes the received credentials.

A *service* is a process running in the background. For example, the process can play music or communicate with other applications.

A *broadcast receiver* can receive broadcasts. For example, the system can send a notification about a specific event that is handled by multiple applications.

A *content provider* provides an abstract data layer for managing data. For example, some data might be stored locally in an SQLite database, while other data is stored in the cloud.

2.2.3 Installation

When installing an application the signature of the APK is first verified. Then the complete APK is stored on the device. If the APK contains libraries then these libraries are extracted and separately stored on the disk. Depending on the compiler the source code contained in the APK is compiled and also stored on the disk. Each application is assigned a Linux UID that is used to isolate applications. For more information see subsection 2.2.4.

Each file is stored at the *userdata* partition in a separate folder. This partition holds all user data including installed applications or e.g. pictures.

Information about the installed application and package are registered at the system during the installation process. This includes e.g. exported components and package information.

System applications are already installed on the read-only *system* partition.

2.2.4 Application Sandbox

The *Application Sandbox* [Andj] is used to isolated applications from each other. It is based on UNIX-style user separation of processes and file permissions and has been enhanced by SELinux (see subsection 2.1.4). Each application is assigned a unique UID and runs in its own process. The sandbox applies not only to user-installed applications but includes every running process excluding the kernel.

Applications that are signed by the same signing certificates can share the same UID. This enables to access files or components of the other application.

An application is required to have the correct permission to perform an action that is outside of its sandbox [Andaf]. Required permissions can be requested in the manifest. For example, the “android.hardware.camera” permission is required to use the camera. Developers can define their own permissions in the manifest. Each permission holds a protection level which states under which conditions the permission is granted.

Normal permissions grant access to low-risk actions and are automatically granted. For example, connecting to a paired Bluetooth device.

Dangerous permissions grant access to high-risk actions and are only granted after the user gives consent. For example, accessing detailed information about the current location.

Signature permissions are only granted if the applications are signed by the same certificates. For example, using the system alert window.

SignatureOrSystem permissions are granted if the application is signed by the correct certificates or if the application is either a system application.

2.2.5 Digital Asset Links

Digital Asset Links [Goo] are statements about applications with respect to an URL. They are defined inside of an *association file* which is hosted at a well-known path of the domain of the URL. The association file contains e.g. information about which application is allowed to register an app link with respect to the URL (see subsubsection 2.3.2.3).

2.2.6 fs-verity Feature

Android can continuously verify the authenticity of the installed APK using *fs-verity* [Andw] if the application store supports this feature. The verification is based on a file signature issued by the application store and performed whenever the file is accessed. The corresponding public key is required to be stored by the manufacturer on the device.

2.3 Inter-Process Communication

IPC enables applications to communicate with each other. Android provides its own IPC mechanisms which should be used out of functionality, flexibility, and security reasons.

In the following, we briefly describe *Binder* which is the underlying concept of IPC on Android. Higher-level concepts are built on top of Binder. Such higher-level concepts are e.g. services, broadcasts, *Messenger*, or intents. A component that is supposed to be used from another application must be exported.

Messenger is a mechanism to implement message-based communication. The interface provides the functionality to send and receive a message from another application.

An *intent* is a command message that can be sent e.g. to an application to start an activity or a service. For example, if the user clicks on a link in the browser and one of the installed applications can handle the link then an intent is sent to this application which starts the respective activity. The user is therefore not redirected to the website but the application. In subsection 2.3.2 we further discuss intents.

2.3.1 Binder

Binder [Ele14] is the underlying concept of IPC on Android: When a process sends a message to another process then Binder copies the message to the memory of the receiver and notifies the receiver about the new message.

The sending process is wrapped into a transaction. Binder adds the UID of the sender to the transaction data. The receiver can trust these and use them to identify the sender. The UID can be resolved using the package manager to package information about the sender, including the package name and the signing certificates. If a shared UID is used then the UID is resolved into multiple packages. Since packages can only share a UID if they are signed by the same signing certificates they hold the same level of trust.

If the two communicating processes are not part of the same application then the receiver is required to be exported. Any process on the device can send a message to an exported process. Therefore, the receiver should enforce appropriate security checks including input validation, authentication, and authorization. Security checks can be e.g. implemented by using the UID of the sender or by relying on Androids permission system (see subsection 2.2.4).

2.3.2 Intents

An *intent* [Andaa] is a command message that e.g. can be sent to an application to start an activity or a service. The intent holds among others the following information:

- *action*: indicates the action to perform, e.g. “ACTION_VIEW” when starting an activity
- *data*: data encoded as URI
- *extras*: key-value pairs containing additional information, e.g. “EXTRA_TEXT” to pass a text message to a messaging service.

Listing 2.1 Intent encoded as URI

```
https://as.dinobank.milesstoetzner.de/oauth/authorization
?request_uri=urn%3Arequest-uri%3A0f37cd45-5391-461f-8ff6-58bea91213c7
#Intent;component=de.milesstoetzner.dinobank/.AuthorizationActivity;end
```

- *package name*: package name of the intent receiver
- *class name*: class name of the component of the intent receiver
- *referrer*: indicates the calling component. This information **can not** be trusted since it can be spoofed by simply setting the intent extra “EXTRA_REFERRER”.

The intent can be represented as URI. For example, the intent shown in 2.1 is used in our implementation to redirect the user to a banking app.

There are two types of intents: *implicit* and *explicit* intents. An implicit intent only specifies what should be done and does not specify a receiver. The receiver of the implicit intent is resolved by the system (see subsubsection 2.3.2.1). Therefore, a malicious application may receive the intent.

On the other hand, an explicit intent additionally specifies the receiver by setting the intent package name and/ or intent class name. The system ensures that the intent is delivered to the specified receiver [Andd].

An implicit intent can be transformed into an explicit intent by first resolving the implicit intent and secondly constructing an explicit intent using the information from the resolved intent.

To receive intents that are sent from other applications the receiver must export the respective component. A component is exported by either directly exporting the component via the “android:exported” [Anda] property or by registering an *intent filter* [Andab].

An *intent filter* is part of the manifest and registers at the system which kind of intents the application can handle. The intent filter can specify e.g. valid intent actions or intent data. For the intent data, the intent filter can specify the scheme, host, port, and/ or path of the to registered URI. An intent filter can e.g. register any URI that uses the “mailto” scheme.

Intent filters do not provide any kind of input validation. They are only used by the system to resolve an implicit intent. Any explicit intent can be sent to any exported component from any sender.

An intent filter can specify required permissions. The system ensures that the sender has the correct permissions.

2.3.2.1 Intent Resolution

When talking about *resolving an intent* we refer to the process of determining the intent receiver. If the intent is explicit then the intent is resolved to the specified receiver. Resolving an implicit intent is more complex. If the intent is e.g. used to start an activity then the system determines the intent receiver as the application that … [Ando]

1. … is registered as the default receiver for this kind of intents

Listing 2.2 Send Intent via href from the Browser

```
<a href="
intent://www.pandawallet.milesstoetzner.de/redirect_uri/dino_bank
#Intent;scheme=https;
package=de.milesstoetzner.pandawallet;
component=de.milesstoetzner.pandawallet.RedirectActivity;
S.browser\_fallback\_url=https://www.pandawallet.milesstoetzner.de/redirect_uri/dino_bank;end
">Redirect</a>
```

2. … that is the only application that has registered a matching intent filter
3. … the user chooses from an app selection dialog. The app selection dialog is shown to the user if the system can not resolve the intent on its own. A list of available applications is shown from which the user picks one.

2.3.2.2 Starting an Activity

An application can start the activity on another application by sending an intent to the exported activity. The activity starts in the foreground and, therefore, redirects the user.

If the sender expects a result the receiver can send an intent back using the same concept. This requires exporting an activity that results in an increased attack surface. Instead, the sender can register a result callback when starting the activity [Andy]. The receiver can return an intent by using the registered result callback. The system ensures that only the receiver can use the callback. When using a result callback the sender does not need to export any component to receive a result.

2.3.2.3 Deep and App Links

An application can claim an URL by registering a *deep link* [Ando]. When the user clicks in the browser on a link then the user is asked if he wants to open the link in the application or stay in the browser. If the user agrees an intent is sent to the application which starts the respective activity.

Multiple applications can register the same URL. For example, each email application might register the schema “mailto”. The user can then chose between the applications and might chose a malicious application.

To secure the registration of an http(s) URL an *app link* can be used [Andao]. An app link is a deep link where the ownership of the URL is verified based on digital asset links (see subsection 2.2.5). When the user clicks on an app link then the user is directly redirected to the application.

An app link is treated as a usual deep link if the ownership verification fails [Andao]. Additionally, app links protect the user only if the respective application is installed. If the application is not installed then a malicious application can register the URL. Therefore, app links are not a reliable security mechanism.

Besides using a deep link it is possible to directly send an intent from the browser [Chra]. 2.2 shows how such an intent can be send using html.

2.3.2.4 Pending Intent

A *pending intent* [GTH18] is an intent that can be handed over by the creator to a receiver who uses the intent at a later point to start e.g. an activity in the name of the creator. The pending intent is bound to the security context of the creator and delegates the creator's privileges to the receiver. A common use case is to pass a pending intent to a system service which later notifies the creator about an event.

The pending intent can be modified by the receiver if the intent is set to mutual by the creator. If not already specified by the creator then the receiver can e.g. specify the intent receiver. A wrong configured pending intent enables privilege escalation. When for example the pending intent is mutable and does not have any specified information then the receiver can send an arbitrary intent in the name of the creator [GTH18]. Groß et al. [GTH18] present a real-world attack to gain system privileges by tampering with a pending intent that has been created by Android's settings application which holds system privileges.

2.3.2.5 Sender and Receiver Identification

In the following, we discuss how an intent sender/ receiver can be identified. We use the identification later for authenticating the sender/ receiver in section 5.7.

The intent receiver can be identified by resolving the intent. Identifying the intent sender is more complex. In the following, we briefly explain the selected scenarios.

If the underlying Binder transaction is accessible then the UID of the sender can be retrieved. This is e.g. the case when the intent is used to bind to a service. As described in subsection 2.3.1 the UID can be resolved to the package information of the sender.

If the intent is used to start an activity and the intent sender registers a result callback then the intent sender can be identified by retrieving the calling package of the activity. The package name can be resolved for the sender's package information using the package manager. Retrieving the calling package of the activity when no result callback is registered would return *null*.

The intent referrer can not be trusted. It can be easily be spoofed by setting the intent extra "EXTRA_REFERRER".

2.4 Rooting and Reverse Engineering

Rooting refers to the process of obtaining root access on a device [Ele14]. Roughly speaking, root access enables full control over the device. This includes changing system configuration or reading and writing any data. Rooting might also disable SELinux which limits root privileges.

One way to root a device is to unlock the bootloader and to flash a custom operating system [Ele14]. The custom operating system has e.g. the *su binary* installed, a binary which enables root access, or is configured to run Android Debug Bridge (ADB) commands as root. ADB [Andb] is a command-line tool, that can be e.g. used to debug applications by sending intents.

device status	cts profile match	basic integrity
certified, genuine device that passes CTS	true	true
certified device with unlocked bootloader	false	true
genuine but uncertified device, such as when the manufacturer doesn't apply for certification	false	true
device with custom ROM (not rooted)	false	true
emulator	false	false
no device (such as a protocol emulating script)	false	false
signs of system integrity compromise, one of which may be rooting	false	false
signs of other active attacks, such as API hooking	false	false

Table 2.1: Examples of basic integrity and cts profile match [Andaj]

Another way to root a device is by exploiting a bug in the device or already installed operating system or to downgrade the operating system using an exploitable older version [Ele14].

Since rooting is done in various ways there are various ways trying to detect rooting [Mueb]. One way is to search for the *su binary* or to check if the *su daemon* is running. A rooted device can always hide from root detection that is enforced by the application on the device since the checks can be e.g. simply disabled. SafetyNet is e.g. a hardware-backed root detection that is evaluated on a server (see section 2.5).

Reverse engineering refers to the process of examining an application to understand how it works, to extract secrets, or to bypass security checks [Mueb]. This is e.g. done by decompiling source code, removing the security checks, and recompiling the source code. Advanced reverse engineering tools, like hooking frameworks, have a component installed on the device and usually require root access. Without root access they would be restricted to their sandbox and e.g. can not access the memory space of another application.

A hooking framework can hook into the execution of the application during runtime and intercept calls to the system or overwrite the logic of a called function. Such a framework can be used to read out secrets or to bypass local authentication that does not require a cryptographic operation. The application can try to protect against such a framework by e.g. checking if a corresponding hooking agent is running the same memory space as the application [Mueb]

Frida [Fri] is for example such a framework. Frida does not always require root access. To use Frida on a non-rooted device the application must be patched with a Frida gadget. Patching the application requires to repackage the application and, therefore, to resign the APK.

2.5 SafetyNet Attestation

SafetyNet Attestation [Andaj] is a JSON Web Signature (JWS) signed by Google that attests the current integrity of a device. Roughly speaking, this means that the device and the system are untampered.

The attestation is based on gathered information about the software and the hardware of the device which are compared against reference data. The attestation should be used whenever a client makes a critical request, e.g. during login or when requesting a bank transaction. Google uses the attestation e.g. to protect Google Pay [Rah].

When an application wants to send a resource request to a server the application starts the flow in step 1 of Figure 2.1 by requesting a fresh nonce at the server. The application uses the nonce to generate the SafetyNet attestation in step 3. It is recommended that the nonce also includes request data. This can be achieved by e.g. concatenating the hash of the resource request body with the nonce received from the server. SafetyNet evaluates device integrity by gathering information about the device and requests an attestation from Google's servers in step 4. After the application has received the attestation in step 6 the application requests the resource at the server in step 7. Part of the request is the attestation. The server verifies the attestation and returns the requested resource.

The payload of the attestation contains the following claims:

- *nonce*: the nonce that has been passed by the calling application during generation.
- *apk package name*: the package name of the calling application.
- *apk certificate sha256*: is the list of signing certificates that have been used to sign the package of the calling application.
- *apk digest sha256*: the hash of the package of the calling application. This claim is not part of the documentation but is mentioned in a blog post [Rod]. Note, that e.g. Google Play might add additional metadata [Rod].
- *basic integrity*: states if the device seems to be not tampered with.
- *cts profile match*: states if the system passed the CTS.
- *evaluation type*: states how the attestation has been evaluated. Possible values are "BASIC" and "BASIC,HARDWARE". The first is based on software evaluation while the last also makes use of hardware-backed security, e.g. hardware-backed key attestation. This claim is not part of the normal documentation but is mentioned in a mailing list [Andu].
- *timestamp ms*: the date when the attestation has been generated on Google's servers.

The package information contained in the attestation can only be trusted if "cts profile match" is true [Andaj]. One reason for this is that if "cts profile match" is false the device might use a custom ROM. In Table 2.1 some scenarios further illustrate the correlation of "basic integrity" and "cts profile match". For example, an emulator neither provides "cts profile match" nor "basic integrity" and a device with an unlocked bootloader does not provide "cts profile match" but "basic integrity".

For accessing the SafetyNet API an API key is required. The key is used for rate limiting, quota restrictions, and monitoring and is e.g stored in the source code. The key can bound to a specific package name and signing certificates and can be restricted to the SafetyNet API only.

The SafetyNet attestation attests device integrity. Bechtsoudis [Bec] analyses if *application integrity* can be attested. Application integrity is split into three goals that are roughly speaking defined as follows:

- *pre-installation integrity*: untampered application assets have been installed, e.g. the APK has not been repackaged.
- *post-installation integrity*: untampered application assets are used to launch the application, e.g. complied source code or extracted libraries have not been tampered with.
- *runtime integrity*: untampered application assets are loaded during launch or while runtime of the application, e.g no hooking frameworks are active.

Since the package signature is checked before installing the application the pre-installation integrity is given. The remaining integrity goals can not be reached [Bec].

Mulliner [Mul] presents a successful attack that makes use of a kernel bug to tamper with compiled source code. Mitigation for that specific attack would be to use the option *run embedded code* [Andai]. This option states that the APK contains DEX code that should be directly run. There are no guarantees though that the installed APK itself or extracted native libraries are untampered. If supported by the application store *fs-verity* can be used to verify the authenticity of the APK on each access (see subsection 2.2.6).

Bechtsoudis [Bec] mentions *The Requester Verification Problem*: How does the server know that the client who requests the nonce, runs the attestation process and finally sends the attestation is the same client. An attacker who is in possession of a leaked attestation could send the attestation from a tampered device misleading the server into believing that device integrity is given. The attack surface can be reduced by authenticating the client and binding the nonce to the client. This still does not fix the problem that the server does not know which client ran the attestation process.

Another mentioned problem is that a systemless root can tamper with the gathered information and successfully bypass the SafetyNet protection. Such a systemless root is e.g. Magisk [Wu]. But since hardware-backed evaluation is supported this is no longer possible [Rah].

2.6 Mobile AppSec Verification Standard

The Mobile App Security Verification Standard (MASVS) [Muea] defines mobile security and reverse engineering resilience requirements. MASVS defines two security levels. The first level holds security best practices for common mobile applications, the second one provides additional defense-in-depth for high-risk applications. The resilience requirements are meant to extend the security levels and aim to make reverse engineering more difficult. The Mobile Security Testing Guide (MSTG) [Mueb] can be used to evaluate the implementation of the security requirements.

MASVS-L1 is the first level and defines common mobile application security best practices. Those requirements contain e.g. input validation, prevent screenshots of sensitive information, certificate pinning, encrypted storage, and the use of hardware-backed security.

MASVS-L2 is the second level. It extends the first level by defining additional security requirements for providing defense-in-depth. This level is meant for high-risk applications like banking apps. For example, it is required to define a threat model, to use biometric authentication for unlocking keys, and to not hold sensitive data in memory longer as required.

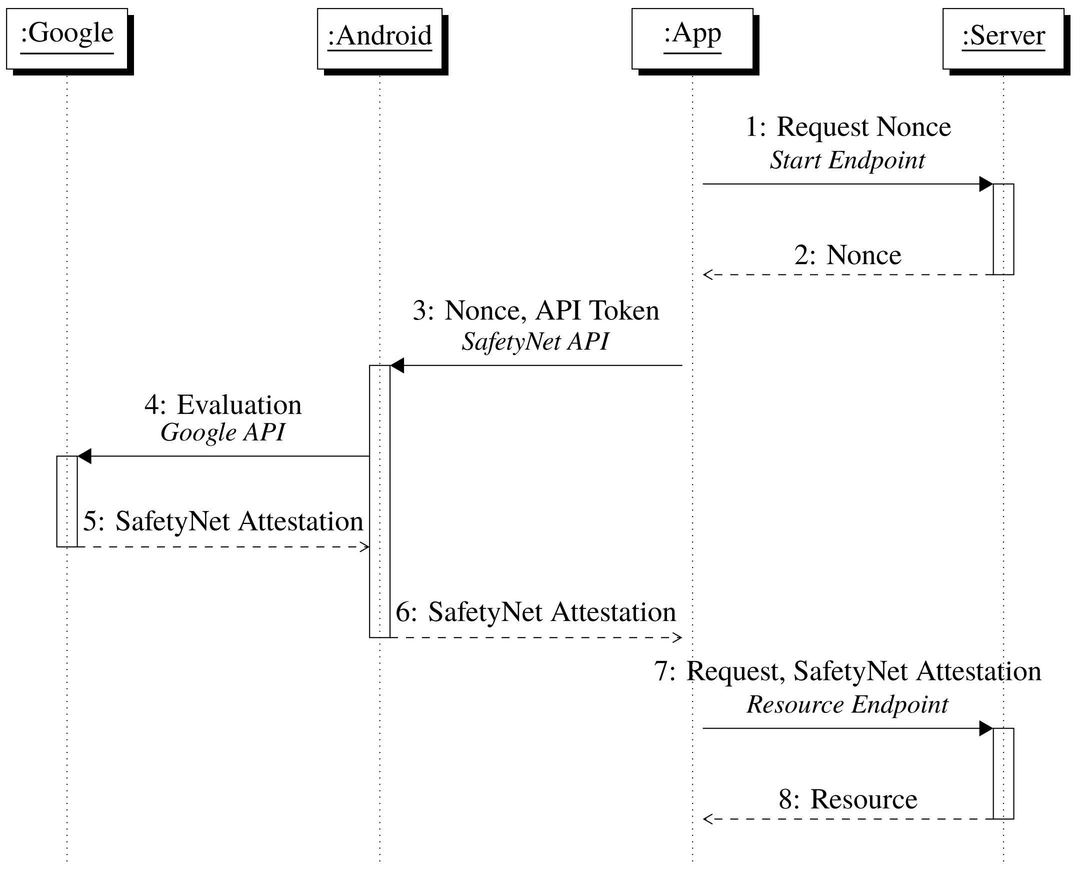


Figure 2.1: SafetyNet Attestation

The resilience requirements are meant to make reverse engineering and tampering harder. In general, we assume that the reverse engineer is always stronger than the reverse engineering defenses enforced in the application. Important verifications, like root detection, should be evaluated on a server and not inside the application. The requirements define e.g. root detection, debugging prevention, hooking detection, code tampering detection, code obfuscation, device binding, and application-level encryption.

3 OAuth2.0 Authorization Framework

OAuth 2.0 is an authorization framework that provides third party access to protected resources belonging to a user. In the following, we introduce OAuth 2.0 and additional extensions. An important extension is e.g. OpenID Connect (OIDC) which is an identity layer for user authentication.

3.1 Overview

A third party, called *client*, wants to access a resource belonging to a *resource owner*, also called *user*. The resource is available at a *resource server*. The resource server authorizes requests based on an *access token* which the client sends along with the request. The client receives the access token from an *authorization server* by providing an *authorization grant*. To obtain such a grant the client redirects the user via the browser to the authorization server. After authorizing the request the authorization server redirects the user back to the client while passing along the required grant.

The third party could be e.g. an image-processing application that stores images in the cloud storage of the user or a digital wallet application that requires access to the user's bank account.

3.2 Client

A client is a third-party application which is registered at the authorization server and which wants to access the resources of a user. The resources are protected by the authorization server and available at a resource server.

The client holds several metadata which are e.g. assigned during registration. Some of these metadata are the following:

- *client name*: human-readable and recognizable name for the client
- *client id*: locally unique client identifier issued by the authorization server
- *redirect URIs*: list of allowed redirect URIs. Each redirect URI identifies a *redirection endpoint*, that is used by the authorization server as the target for the authorization response.

There are two types of clients: *public* and *confidential* clients. A *public client* is not able to maintain secrets that can be e.g. used for authenticating the client. On the other hand, a *confidential client* is able to maintain secrets. There are several ways to authenticate a client, which is called *client authentication*. One way is a shared symmetric secret that is known by the client and the authorization server. This secret is generated by the authorization server during registration.

There are several client profiles: *web applications*, *user-agent-based applications*, and *native applications*. A *web application* is a confidential client that runs on a web server. The web server is capable of storing secrets. Such an application is e.g. a website whose HTML is rendered on the server.

A *user-agent-based application* is a public client that runs in a user-agent. Since the source code and configurations are available in the browser this client can not hold any secrets. Such an application is e.g. a Javascript web application.

A *native application* is a public client that is installed on the user-agent. Such an application is e.g. an Android app on the user's phone. The difference to a user-agent-based application is that such a native application can make use of the security features of the mobile environment to e.g. securely store dynamically registered secrets. This client can e.g. not hold any shared or preconfigured secrets across client instances without exposing them since the source code and all configurations are downloaded on the user-agent and can be easily accessed.

3.3 Authorization Code Mode

OAuth 2.0 has several modes and settings. In the following, we take a look at the protocol of the *authorization code mode*.

The client starts the flow by constructing and sending an *authorization request* (see step 1). The browser redirects the user to the *authorization endpoint* of the authorization server. The following parameters are encoded in the URL:

- *client id*: the client id received during client registration
- *response type*: defines what kind of grant should be returned which is in this case "code"
- *redirect URI*: defines the redirection endpoint
- *scope*: defines the resource the client wants to access. This value is business-specific and out of scope of the specification. In case of a banking scenario, this value could be "transactions:read", which might represent the read access to all bank transactions of a bank account.
- *state*: fresh nonce chosen by the client for Cross-Site Request Forgery (CSRF) protection

The authorization server displays details about the authorization request (see step 4). The user authorizes the request by authenticating at the authorization server and giving consent, e.g. by clicking on a button (see step 4).

The authorization server verifies among others that the user is allowed to authorize the request, issues an *authorization code*, and sends the *authorization response* to the redirect URI (see step 5). To prevent open redirect attacks (see subsection 3.3.2) the authorization server verifies that the redirect URI is allowed. The browser redirects the user to the redirection endpoint of the client (see step 6). The following parameters are encoded in the URL:

- *code*: contains the authorization code for the client opaque value
- *state*: taken from the authorization request

To prevent CSRF attacks (see subsection 3.3.1) the client verifies that the state parameter of the authorization response matches the state parameter of the authorization request. After that, the client exchanges the code for an access token, which is called *token exchange*: The client sends a *token request* to the *token endpoint* of the authorization server (see step 7). The request contains the following parameters:

- *grant type*: defines what kind of grant is provided which is in this case “authorization_code”
- *code*: taken from the authorization response
- *redirect URI*: taken from the authorization request
- *client id*: taken from the authorization request

If the client is a confidential client, then the authorization server authenticates the client. The authorization server verifies that the code has been issued for the client id and that the code is still valid. After verifying that the redirect URI matches the redirect URI from the authorization request, the authorization server issues an for the client opaque access token and returns the access token in the *token response* (see step 8).

The client uses the access token from the token response during a *resource request* sent to the *resource endpoint* of a *resource server* (see step 9). The resource server authorizes the request based on the access token and returns the resource (see step 10). How the resource server extracts the required information from the access token is out of scope of the specification. The access token might be a JSON Web Token (JWT) which contains all required information or the resource server inspects the token at the *introspection endpoint*. The introspection endpoint [Ric15] is an endpoint at the authorization server which can be used by the resource server to request information about an access token, e.g. if the token is still valid or which user authorized the token.

3.3.1 State Parameter

The client verifies that the value of the state parameter of the authorization response matches the value of the state parameter from the authorization request. This value should be a fresh nonce for each flow. Without this verification, an attacker can perform the following attack.

The attacker starts a flow using the honest client, authorizes the request but stops the flow before the browser redirects the attacker back to the client in step 6. When an honest user starts a flow using the honest client the attacker can send the authorization response of his flow to the redirection endpoint. The client uses the authorization code associated with the attacker in the flow of the honest user. The honest user could be e.g. forced to use the resources of the attacker.

3.3.2 Open Redirect

The authorization server verifies in step 4 that the redirect URI is allowed. Without this verification an attacker can perform the following attack.

The attacker can start a flow in the name of the honest client and set the redirect URI to an endpoint under his control. Information about the honest client is displayed to the honest user who authorized the request. The authorization response is sent to the attacker who gains access to a valid

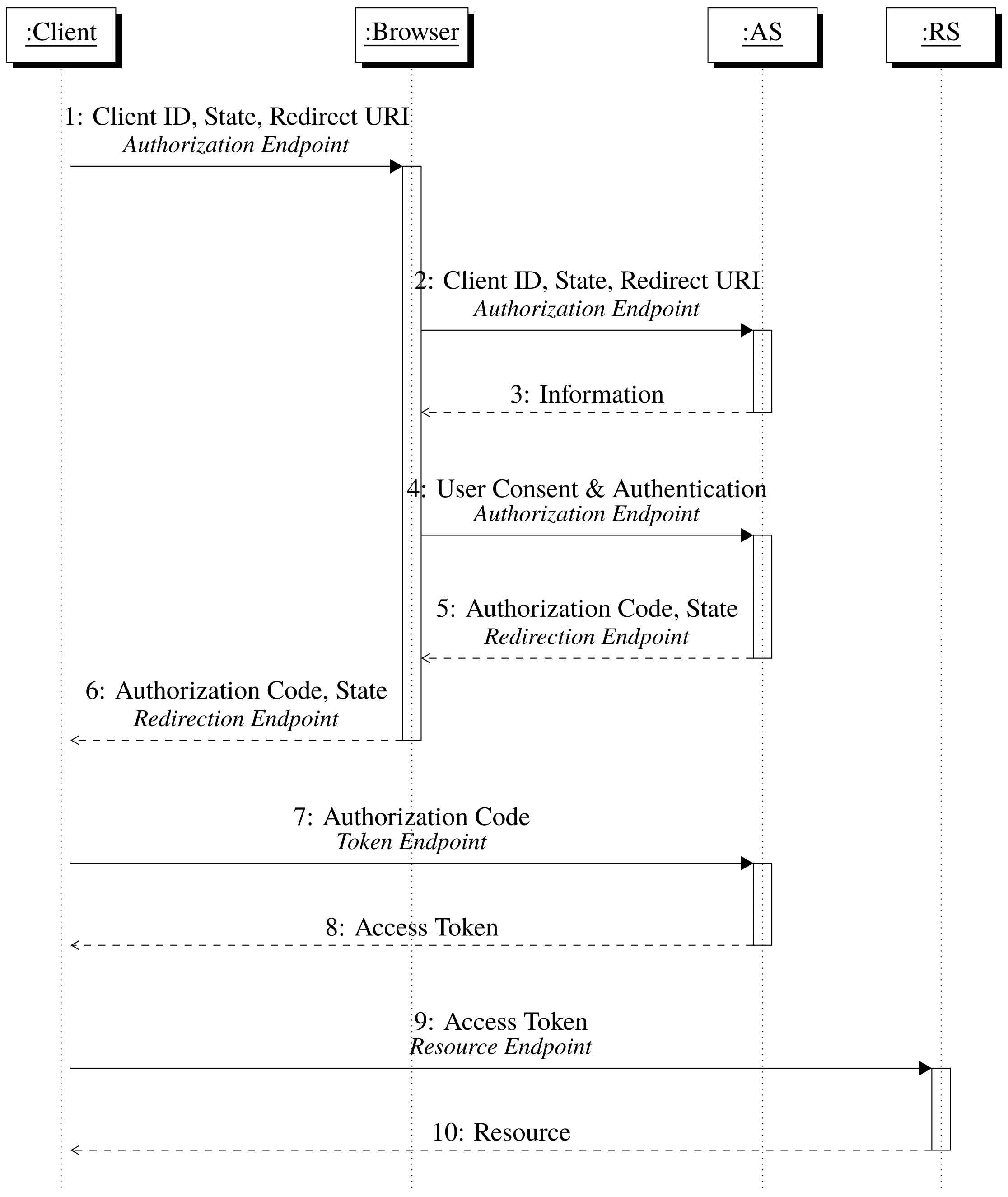


Figure 3.1: OAuth2.0 Authorization Code Flow

authorization code associated with an honest user. If the honest client is a public client, then the attacker can exchange the authorization code for an access token and use the access token to access the resources belonging to the honest user at the resource server.

3.4 JSON Web Token

A JWT [JBS15b] is a signed, encrypted, or signed and encrypted JSON object whose fields are called claims. JWTs can be used to safely pass information between parties.

In OAuth 2.0 a JWT can be used as an access token: The claims of the JWT hold all information that the resource server requires for authorization. This includes e.g. the "scope" parameter of the authorization request. The resource server can trust these claims after verifying that the JWT has been signed by the authorization server.

A signed JWT is a JWS [JBS15a]. The compact serialization of a JWS is a string that consists of three parts:

- *header*: contains metadata, e.g. *typ* which contains the media type of the payload or *alg* which contains the used signature algorithm
- *payload*: is the JSON object which holds the claims
- *signature*: is a signature over the header and payload

The claims are business-specific and can be extended as required. Common claims are the following:

- *iss*: identifier of the issuer of the JWT, e.g. the issuer of an authorization server
- *sub*: subject of the JWT, e.g. a username or email
- *aud*: audience for which the JWT has been issued, e.g. the identifier of a resource server
- *iat*: date when the JWT has been issued
- *exp*: date when the JWT is not valid anymore

An encrypted JWT is a JSON Web Encryption (JWE) [JH15] with the JSON object as plaintext. A signed and encrypted JWT is an encrypted JWT whose plaintext is a signed JWT. More details about JWTs can be found in [JBS15b].

3.5 OpenID Connect

OAuth 2.0 provides an authorization and not an authentication framework. OIDC is an identity layer on top of OAuth 2.0 to enable user authentication. In the following, we explain authentication based on the authorization code flow.

The flow does only differ from the authorization code flow in additional request parameters and that the client exchanges the authorization code at the token endpoint not only for an access token but also for an *id token* that holds the identity of the user. The authorization request contains the following additional parameters:

- *nonce*: fresh nonce for mitigating replay attacks
- *scope*: contains next to optional other scopes the value "openid"

The authorization server additionally returns an *id token* at the token endpoint which holds the identity of the user. The id token is a JWT that is signed by the authorization server and that holds among others the following claims:

- *iss*: identifier of the authorization server
- *sub*: locally unique identifier of the user
- *aud*: contains at least the client id
- *nonce*: taken from the authorization request
- *at hash*: hash of the access token

The identity contained in the id token is in scope of the authorization server.

Considering OIDC, the authorization request is called *authentication* request. This applies to other namings too.

3.6 Dynamic Client Registration Protocol

OAuth 2.0 does not define the mechanism for client registration at an authorization server and assumes that client registration is e.g. previously done manually. *OAuth 2.0 Dynamic Client Registration Protocol* [RJB+15] defines a protocol for dynamically registration of clients.

A client sends a *registration request* to the *registration endpoint* at the authorization server. The request contains the client's metadata, e.g. the client name, a set of valid redirect URIs, or a JSON Web Key Set (JWKS). After successfully verifying the registration request the authorization server registers the client and returns the client metadata including a client id and if required, client secrets.

The specification defines new client metadata called *software id* and introduces a concept to assert client metadata during registration, the *software attestation*. Both can be part of the registration request.

The *software id* is an identifier of the client software. It can be e.g. used to assign metadata to all instances of client software, e.g. client name or redirect URIs. In comparison to the client id each client instance of the same software shares the same software id but has a different client id.

The *software statement* is a signed JWT which holds client metadata as claims. The statement can be issued by the client or by a third party. The authorization server can use the statement to allow or reject registration requests. The statement should at least contain the software id. As stated by the specification the software statement is treated as self-asserted and is "not sufficient in most cases to fully identify a piece of client software" [RJB+15].

To protect the registration endpoint from potential malicious registrations, the endpoint can e.g. require an access token, the so-called *initial access token*. How the client obtains such an initial access token is out of scope. Such a malicious registration might contain e.g. a misleading logo.

3.7 JWT Secured Authorization Request

The authorization request parameters are encoded in the URL and send e.g. through the web browser. This concept does not provide integrity protection or source authentication. An attacker can tamper with these parameters; this enables e.g. mix-up attacks [FKS16]. To mitigate these problems JWT Secured Authorization Request (JAR) [SBJ20] introduces the *request object*.

A *request object* is a signed and optional encrypted JWT that holds the authorization request parameters as claims. The signature provides integrity and source authentication while the encryption provides confidentiality. The authorization server verifies the JWT before trusting the claims. The distribution of the required keys is out of scope of JAR. They might be registered during client registration.

The request object is either passed by value or by reference. When passed by value then the request object is encoded in the URL as "request" parameter. When passed by reference then the request object is stored at some URI which the authorization server can access, so-called *request URI*. The request URI is encoded in the URL as "request_uri" parameter. The request URI can point to any location, including third parties or the authorization server itself. See section 3.9 for more details on storing the request object at the authorization server.

3.8 JWT Secured Authorization Response Mode

JWT Secured Authorization Response Mode (JARM) [LC] defines a way to send the authorization response parameters not as URL parameters but as claims in a JWT. Since the JWT is signed and issued by the authorization server JARM provides integrity of the response parameters and mitigates e.g. mix-up attacks [FKS16]. The client verifies the JWT before trusting the claims. The distribution of the required keys is out of scope of JARM. They might be published as part of the authorization server's metadata.

The JWT holds typically the following claims:

- *iss*: identifier of the authorization server
- *aud*: client id
- *exp*: expiration of the JWT
- *code*: authorization code

- *state*: taken from the authorization request

There are different response modes. We make only use of “query.jwt” which encodes the JWT in the *request* URL parameter.

3.9 Pushed Authorization Requests

OAuth 2.0 Pushed Authorization Requests (PAR) [LCS+20] introduces the *pushed authorization request endpoint* which addresses, among others, the same problems as discussed by JAR: integrity protection and source authentication.

The pushed authorization request endpoint requires client authentication and is used by the client to push the authorization request parameters to the authorization server in exchange for a request URI before redirecting the user. The authorization server validates the parameters the same way as it would have validated the parameters at the authorization endpoint. Since client authentication is required and the request parameters are verified the authorization server can stop the flow even before the user is redirected. Note, that the request URI is one-time use and bound to the client.

PAR complements JAR and provides a mechanism to pass a request object by reference: The client can push a request object by value to the pushed authorization request endpoint. The returned request URI is then used to pass the request object by reference at the authorization endpoint.

3.10 Rich Authorization Requests

OAuth 2.0 Rich Authorization Requests (RAR) [LRC19] defines a new authorization request parameter called *authorization details*. Authorization details can be used to request fine-grained access, e.g. “transfer 100€ from Marvin to Fiona”. In comparison, the “scope” parameter is more coarse-grained and would usually request more access than required, e.g. “require access to transfer money using the user’s bank account”.

Authorization details is an array of JSON objects, so-called *authorization data*. Each authorization data may hold the following fields:

- *type*: defines the type of authorization data, e.g. “payment_initiation”
- *locations*: defines the resource endpoints
- *actions*: defines the actions to perform at the resource endpoints

The authorization data can hold any additional required fields, e.g. debtor and creditor account for money transactions. Exemplary authorization details for requesting a payment is shown in Listing 3.1.

Authorization details can replace or complement existing scopes. For example, the “openid” scope can be also expressed as authorization details (see Listing 3.1).

Listing 3.1 Authorization Details Example [LRC19]

```
[
   {
      "type": "payment_initiation",
      "actions": ["initiate", "status", "cancel"],
      "locations": ["https://example.com/payments"],
      "instructedAmount": {
         "currency": "EUR",
         "amount": "123.50"
      },
      "creditorName": "Merchant123",
      "creditorAccount": {
         "iban": "DE02100100109307118603"
      },
      "remittanceInformationUnstructured": "Ref Number Merchant"
   },
   {
      "type": "openid",
      "locations": ["https://op.example.com/userinfo"],
      "claim_sets": ["email", "profile"]
   }
]
```

3.11 Proof Key for Code Exchange

Proof Key for Code Exchange (PKCE) [SBA15] presents a mitigation for authorization code interception attack for public clients. The rough idea is to dynamically generate credentials that are used during the flow to authenticate the public client at the token endpoint.

PKCE assumes that the authorization request and response can leak to an attacker who obtains among others a valid authorization code associated with an honest user. The leakage of the authorization request is motivated by leaked HTTP log messages, the leakage of the authorization response is motivated by a malicious application which registers the redirect URI on the mobile device and intercepts the user redirect (see section 3.13). The attack has the same impact as the open redirect attack (see subsection 3.3.2). If the client is a public client, then the attacker can exchange the authorization code at the token endpoint for a valid access token and use this access token to access the resources of an honest user. Note, that the token endpoint does not require client authentication if the client is a public client. Such attacks have been observed in the reality [SBA15]. The presented mitigation works as follows.

The public client generates a fresh *code verifier*, which is a cryptographically random key, and a *code challenge*, which is the sha256 hash of the code verifier. The client sends the code challenge as authorization request parameters and the code verifier later during the token exchange. The authorization server verifies that the caller of the token request provides the correct code verifier. The idea is that only the creator of the code challenge has knowledge of the correct code verifier. Note, that PKCE does not authenticate the caller as the client but only as the creator of the code challenge (see subsection 5.15.1).

Furthermore, PKCE can be used to protect the client state and prevent injections of authorization responses. PKCE should be used by all kinds of clients [LBLF20]. This makes the use of the “state” parameter for state protection obsolete.

3.12 Mutual TLS for Client Authentication and Certificate Bound Access Tokens

OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens [CBSL20] defines how Mutual TLS Authentication (mTLS) can be used for client authentication and how access tokens can be bound to mTLS client certificates.

A client can authenticate at the authorization server by using an mTLS client certificate. During the TLS handshake, the proof of possession of the corresponding private key is performed. Depending on the scenario, the client uses either a self-signed certificate or a certificate signed by an authority that the authorization server trusts. A self-signed certificate could be e.g. dynamically registered together with the client at the registration endpoint. Note, that the client id is not contained in the certificate. To identify the client, the client is supposed to send his client id, e.g. in the body of the request.

In general, the possession of a valid access token is enough to use the access token at the resource endpoint. This enables an attacker to use a leaked access token for a resource request. To mitigate this problem the authorization server can issue sender-constrained access tokens by binding the access token to the client certificate. The binding can be done e.g. by associating the access token with the thumbprint of the mTLS client certificate. The resource server additionally verifies this binding at the resource endpoint. An attacker who is not in possession of the corresponding private key can therefore not use the leaked access token on his own.

Note, that the resource server is not required to verify the client certificate. The proof of possession of the corresponding private key is sufficient. The certificate has been already verified by the authorization server when issuing and binding the access token.

This concept can be applied to any kind of tokens that the authorization server issues.

3.13 Native Apps

OAuth 2.0 for Native Apps [DB17] defines current best practices for using OAuth 2.0 with a native application as a client that redirect the user to the web.

The client must only redirect the user to an external user-agent and must not use an embedded user-agent. An external user-agent is a user-agent that is not controlled by the client, e.g. a browser. An embedded user-agent is a user-agent that is under the control of the client, e.g. a web view. When the client is using an embedded user-agent then he has access to e.g. entered credentials or cookies. Another reason to use an external user-agent is that the user might be e.g. already logged in at the external user-agent.

Since native applications are considered public they can register dynamically as a confidential client at the registration endpoint. Each application instance has its own client id and credentials assigned.

On Android, the user should be redirected to the web by sending an intent. The document does not further specify if the intent is implicit or explicit but we assume that an implicit intent is meant. The user-agent redirects the user back to the redirection endpoint using an app link.

The document falsely assumes that the identity of the app that receives the authorization response is guaranteed by the app link. As discussed in subsubsection 2.3.2.3 this is not the case. This enables the interception of authorization responses as assumed (and mitigated) by PKCE.

3.14 Client Initiated Backchannel Authentication Flow

The authorization code flow is a redirect based flow. The user is redirected from the client to the authorization server and back. There are scenarios where a user redirect is not possible or desired. For example, the user authorizes a payment at the checkout of a supermarket using his mobile phone. Client Initiated Backchannel Authentication (CIBA) [Fer] presents a way to authenticate the user in the background on another device.

The client is running on the so-called *consumption device*. To start the flow the client sends an authentication request to the newly introduced *backchannel authentication endpoint* at the authorization server in exchange for an *auth req id* (see step 1). The auth req id identifies the current flow and is later used as a grant for the token exchange, a so-called *ciba* grant.

Depending on the client type the backchannel authentication endpoint requires client authentication. The authorization server authenticates the user in the background using an *authentication device*, e.g. the user's mobile phone (see step 5). How the authorization server identifies the authentication device or how the user is notified and authenticated is out of scope. The user might have entered his email at the client who passes the email as part of the authentication request to the authorization server.

There are three modes. In the *poll* mode, the client sends periodically the token request (see step 3 and step 7). In the *ping* mode, the client sends the token request after receiving a notification at the so-called *client notification endpoint* from the authorization server. The client notification endpoint is protected by a *client notification token* that the client sends during the authentication request. In the *push* mode, the authorization server sends the token response directly to the client notification endpoint. Note, that the push mode is not allowed by *CIBA FAPI* [Sakc].

One problem with this concept is that malicious clients can start a flow and might display notifications on the authentication device. The proposed mitigation is to use a *user code*. The user enters a *user code* at the client which the client sends as part of the authentication request to the authorization server. The authorization server verifies that the user code is as expected. Registration and verification of the user code is out of scope. The user code might be registered during account registration at the authorization server. Note, that this code is not a valid credential to authenticate as the user at the authorization server.

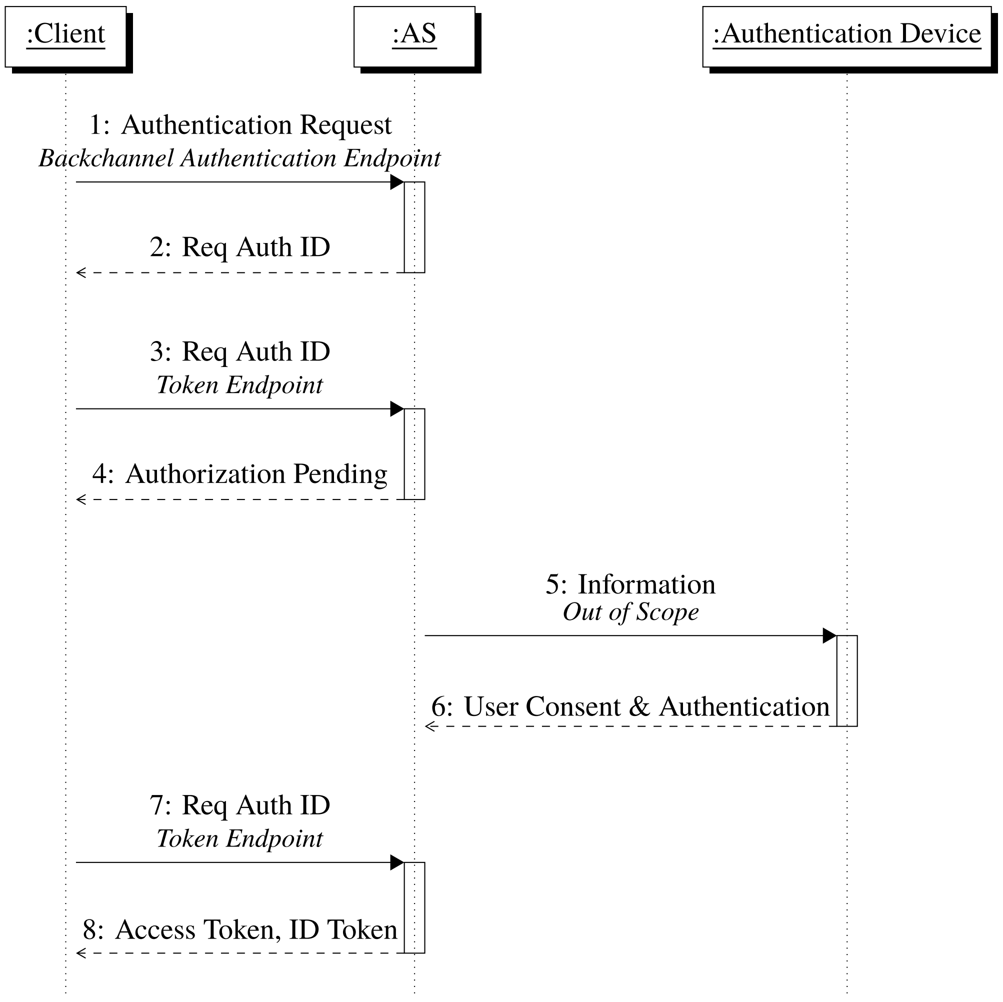


Figure 3.2: CIBA Poll Mode

Another problem is that the user does not know if the currently displayed request on the authentication device corresponds to the request that the user intends to finish. The client can send a *binding message* as part of the authentication request. The binding message is e.g. a six-digit long number and is displayed at the consumption and authentication device. The user is required to verify that the binding messages match.

4 OpenID Financial-grade API 2.0

FAPI 2.0 [Fetc] is a security profile for OAuth 2.0. This means that FAPI 2.0 specifies which settings and extensions of OAuth 2.0 are required. FAPI 2.0 aims to reach the security level for high-risk environments like the financial sector. For example, Open Banking [Ope] defines a uniform API for banks and makes use of FAPI 1.0 [Saka] [Sakb] to protect the endpoints. Note, that FAPI 2.0 is currently a draft.

In this section, we take a look at the attacker model and the two security profiles defined by FAPI 2.0. The baseline profile aims to reach the required security level while the advanced profile extends the baseline profile by non-repudiation requirements.

4.1 Attacker Model

In the following, we give an overview of the attacker model under which FAPI 2.0 aims to be secure. For more information see *FAPI 2.0 Attacker Model* [Fetb].

FAPI 2.0 defines three security goals. We explain these in more detail during our security discussion.

- *Authorization* states that “no attacker can access resources belonging to a user” [Fetb].
- *Authentication* states that “no attacker is able to log in at a client under the identity of a user” [Fetb].
- *Session Integrity* states that “no attacker is able to force a user to be logged in under the identity of the attacker and that no attacker is able to force a user to use the resources of the attacker” [Fetb].

The following attacker capabilities are considered. We explain these in more detail during our security discussion.

The attacker can have control over the network. Such an attacker can e.g. block messages.

The attacker can be a standard web attacker. Such an attacker can e.g. send and receive messages and act as an authorization server.

The attacker can read the authorization request. This assumption is based on the fact that the user is redirected through a complex stack with a large attack surface.

The attacker can read the authorization response. This assumption is e.g. based on the fact that a malicious application can register the redirection endpoint of an honest client and can then intercept the authorization response (see section 3.11).

The client might use a misconfigured token endpoint. This assumption is based on social engineering attacks. The attacker has therefore access to an authorization code of an honest user and can try to inject an access code.

The client can read the resource request and has therefore access to an access token.

4.2 Baseline Profile

The *FAPI 2.0 Baseline Profile* [Fetc] is a security profile for OAuth 2.0 and aims to be secure with respect to the attacker model.

The baseline profile makes use of the following mechanisms:

- authorization code mode
- PAR for integrity protection and source authentication of the request parameters
- PKCE with sha256 for state protection
- mTLS for client authentication
- sender-constrained tokens for mitigating leaked tokens
- RAR for fine-grained authorization requests
- Transport Layer Security (TLS) to encrypt messages that are sent over the network
- HTTP Strict Transport Security (HSTS) to protect against TLS stripping attacks
- Domain Name System Security Extensions (DNSSEC) to protect against DNS spoofing attacks

Furthermore, clients are required to protect against mix-up attacks. This can be done by using a distinct redirect URI per issuer [FKS16].

In the following, we take a look at the flow when using the baseline profile.

The client starts the flow by pushing the authorization request to the pushed authorization request endpoint in exchange for a request URI (see step 1). Part of the authorization request is the client id, redirect URI, authorization details and a fresh generated code challenge using sha256. The authorization server authenticates the client at the pushed authorization request endpoint and verifies the request parameters.

The client redirects the user via the browser to the authorization endpoint (see step 3). Only the request URI encoded in the URL is passed to the authorization server. After the user authenticates and gives consent (see step 7) the authorization server issues an authorization code that is bound to the client certificate and redirects the user back to the redirect URI (see step 8).

The client sends the authorization code and code verifier to the token endpoint in exchange for an access token (see step 9). If user authentication has been requested by the client then an id token is additionally returned. The authorization server verifies the binding of the authorization code and coder verifier. The issued access token is bound to the client certificate.

The client sends the access token to the resource endpoint (see step 11). The resource server verifies the binding of the access token and returns the resource.

4.3 Advanced Profile

The *FAPI 2.0 Advanced Profile* [Feta] is a security profile for OAuth 2.0 and aims to be secure with respect to the attacker model and non-repudiation requirements.

The advanced profile extends the baseline profile by non-repudiation requirements. These non-repudiation requirements include e.g. the non-repudiation of authorization requests and responses. The additional requirements are achieved by using application-level signatures, e.g. JAR and JARM.

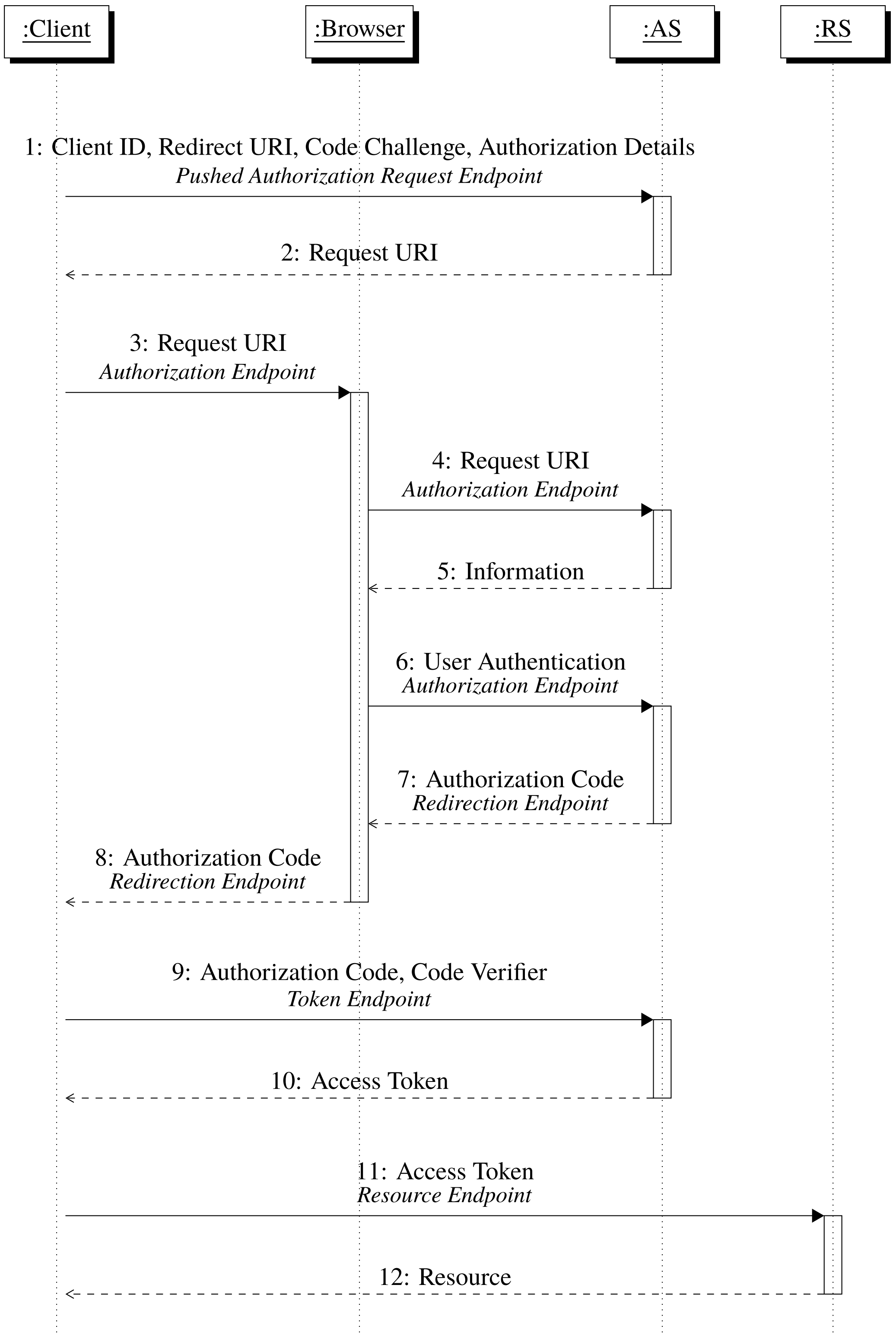


Figure 4.1: FAPI 2.0 Baseline Profile

5 Android App2App Redirect Flow

In the following, we present our proposal for a *FAPI 2.0 Android App2App Redirect Flow*. Instead of being redirected between two websites in the browser the user is redirected on his device between two native applications, the client and the auth app.

5.1 Overview

The client is an OAuth 2.0 client implemented as a native application that dynamically registers as confidential client at the authorization server. The auth app is a native application that implements the authorization endpoint. The auth app prompts the user for his credentials and registers at the authorization server.

During the flow, the client redirects the user to the auth app using an explicit intent and registers a result callback. The redirect is mutually authenticated. The client authenticates the intent receiver as the expected auth app and the auth app authenticates the intent sender as the expected client. Authentication is based on verifying that the package of the intent receiver is signed by expected signing certificates.

When authorizing the request the user is locally authenticated using biometrics. The authentication unlocks a hardware-backed private key for a single cryptographic operation which is used to sign the request information. This signature is called *user consent*. The corresponding public key has been registered at the authorization server during auth app registration. The auth app exchanges the user consent at the authorization server for an authorization code and redirects the user back to the client using an intent result callback.

During client and auth app registration the authorization server verifies that the correct application has sent the registration request using an untampered device. The verification is based on an oauth integrity attestation which contains device and app integrity attestations, e.g. the SafetyNet attestation.

Our main goal is to securely redirect the user on Android between the client and the auth app using today's technologies. This is achieved by authenticating the intent receiver and sender based on package signing certificates as well as by using a result callback.

We assume that the client uses a misconfigured auth app and that the auth app might have some misconfigured endpoints. The endpoint for introspecting the request URI and the endpoint for exchanging the user consent for an authorization code might be misconfigured. These assumptions are based on social engineering attacks. We also mitigate known attacks on FAPI 1.0 [Dam18]. In general, we mitigate our assumptions and the known attacks by informing the authorization server or resource server about which auth app or endpoint has been used. Note, that each FAPI 2.0 profile is required to be secure with respect to the FAPI 2.0 attacker model.

5.2 Parties

In this section, we present the parties that are part of the app2app redirect flow. We describe the function of each party during the flow, along with metadata and keys that are important for our proposal.

5.2.1 Client

A *client* is an OAuth 2.0 client implemented as native application. We assume that the native application does not have a server backend. After installation, the client registers as a confidential client at an authorization server using dynamic client registration (see section 5.10).

We differentiate between the data that belongs to the client, the *client data*, and additional data of the native application that is e.g. related to business logic or user experience. The native application might contain multiple clients for supporting e.g. multiple banks. The client data hold e.g. access tokens and is encrypted using a hardware-backed key.

The client holds the following hardware-backed keys. Only the client has access to these keys for a validity duration after the user has been locally authenticated using device credentials.

- *client registration certificate*: self-signed certificate used during registration used for binding an integrity nonce
- *client certificate*: self-signed certificate used for client authentication and certificate bindings
- *client verification key*: public key for application-level signatures
- *client signature key*: private key for application-level signatures
- *client encryption key*: public key for application-level encryption
- *client decryption key*: private key for application-level encryption
- *client data key*: key for encrypting and decrypting the client data

The client holds among others the following (newly introduced) metadata:

- *client id*: oauth client id, e.g “90634c81-2c1b-40d5-a0fb-ebd734f245a4”
- *software id*: inherits properties from the software identified by this software id, e.g “urn:android-package:de.milesstoetzner.pandawallet”
- *apk package name*: name of the package, e.g “de.milesstoetzner.pandawallet”
- *apk redirect activity*: canonical name of the java class that identifies the (app) redirection endpoint, e.g “de.milesstoetzner.pandawallet.RedirectActivity”
- *apk signing certificates*: list of signing certificates that have been used to sign the package
- *apk minimum version*: version code that defines the minimum required package version, e.g “1”
- *apk maximum version*: version code that defines the maximum allowed package version, e.g “1”

We define the term *client instance* as the instance of the client software that is installed on a device [RJB+15]. Each instance has e.g. a different client id and a different set of keys.

5.2.2 Authorization App

An *auth app*, or *authorization app*, is a native application that implements the authorization endpoint. The user enters his credentials into the auth app which registers at an authorization server (see section 5.11). The auth app can then be used to authorize requests.

We differentiate between the data that belongs to the auth app, the *auth app data*, and additional data of the native application that is e.g. related to business logic or user experience. The native application might contain multiple auth apps for supporting e.g. different accounts at different banks at the same time. Note, that an auth app is registered only at a single authorization server.

Each auth app must be isolated from each other. This means that e.g. developers must make sure that no auth app can access any data or credentials from another auth app.

We define the term *auth app authentication* as the authentication of an auth app instance.

The auth app holds the following hardware-backed keys. Only the auth app has access to these keys for a validity duration after the user has been locally authenticated using device credentials.

- *user verification key*: public key for verifying user consent
- *user signature key*: private key for signing user consent. This key requires biometric authentication for each operation.
- *auth app registration certificate*: self signed certificate used during registration used for binding an integrity nonce
- *auth app certificate*: self signed certificate used for auth app authentication and certificate binding
- *auth app verification key*: public key for application-level signatures
- *auth app signature key*: private key for application-level signatures
- *auth app encryption key*: public key for application-level encryption
- *auth app decryption key*: private key for application-level encryption
- *auth app data key*: key for encrypting and decrypting app data

The auth app holds among others the following metadata:

- *auth app id*: locally unique identifier of an auth app instance issued by the authorization server during registration, e.g. “4665f634-1a9b-467a-a9a1-2b256dc0de5b”
- *software id*: inherits properties from the software identified by this software id, e.g “urn:android-package:de.milesstoetzner.dinobank”
- *apk package name*: name of the package, e.g “de.milesstoetzner.dinobank”
- *apk authorization activity*: canonical name of the java class that identifies the (app) authorization endpoint, e.g. “de.milesstoetzner.dinobank.AuthorizationActivity”

- *apk signing certificates*: list of signing certificates that have been used to sign the package
- *apk minimum version*: version code that defines the minimum required version, e.g "1"
- *apk maximum version*: version code that defines the maximum allowed version, e.g "1"

The software id enables the authorization server to identify to which branding the auth app belongs. Analogously to the term *client instance*, we define *auth app instance* as the instance of the auth app software that is installed on a device.

Besides the auth app id these metadata could be distributed using the authorization server metadata. The authorization server should set appropriate caching policies since e.g. the apk minimum version might change quite frequently in comparison to the metadata *response types supported*.

Technically the client is not required to know the apk authorization activity when the auth app implements an intent filter. The use of an intent filter would be more flexible for future internal changes of the auth app. We explicitly use the canonical name of the activity for an easier discussion.

5.2.3 Authorization Server

This party is an authorization server as presented in chapter 3.

The authorization server holds among others the following keys:

- *as certificate*: TLS server certificate
- *as verification key*: public key for application-level signatures
- *as signature key*: private key for application-level signatures
- *as encryption key*: public key for application-level encryption
- *as decryption key*: private key for application-level encryption

The authorization server holds among others the following newly introduced metadata:

- *apk preference*: indicate that e.g. no auth app exists or that the use of an auth app is required:
  - *prohibited*: client is not allowed to use the auth app
  - *preferred*: client can decide if it wants to use the auth app
  - *required*: client must use the auth app
- *safety net attestation evaluation types supported*: list of allowed evaluation types of the SafetyNet attestation, e.g. "['HARDWARE_BACKED']"

5.2.4 Resource Server

This party is a resource server as presented in chapter 3.

5.2.5 User and Device

A user is a resource owner and the owner of an Android device. The device must support hardware-backed security, e.g. Samsung Galaxy S10.

5.3 OAuth Integrity Attestation

The *oauth integrity attestation* is meant to continuously provide integrity attestations for a device and/ or application and non-repudiation with respect to request data. The idea is to generate one or multiple *integrity attestations* and send them wrapped in a signed and optional encrypted JWT as header to the authorization server whenever a critical action should be performed by a client. An integrity attestation requires a nonce that is issued by the authorization server and extended by request data by the client. Such a client is e.g. an oauth client during token exchange or an auth app during user consent exchange. Additionally, the attestation is used to verify that the correct application has sent the request.

The concept is adapted from SafetyNet and extended to incorporate additional optional integrity attestations which for e.g complement each other. The oauth integrity attestation is defined in a way that it can not only be used for Android but also in other environments, e.g. IOS. The attestation is signed to provide integrity and optional encrypted to provide confidentiality. Private information could be part of integrity attestations that contain e.g. the current location of the user or a global device identifier.

The oauth integrity attestation is a JWT which has the type *oauth_integrity_attestation+jwt*. The payload of the JWT contains the following claims.

- *iss*: identifier of the issuer. This claim is missing during registration.
- *aud*: issuer of the authorization server
- *exp*: date when the JWT is not valid anymore, should be short living
- *iat*: date when the JWT has been issued
- *integrity nonce*: integrity nonce issued by the authorization server
- *source endpoint*: URL of the endpoint from where the integrity nonce has been received, e.g. “https://as.dinobank.milesstoetzner.de/integrity”
- *target endpoint*: URL of the endpoint where this attestation is sent to, e.g. “https://as.dinobank.milesstoetzner.de/registration”
- *client data hash*: hash of a stringified JSON containing the *client data* (see below)
- *safety net*: (optional) contains an attestation returned by the SafetyNet Attestation API

The payload might contain any additional claims for providing other types of integrity attestations, e.g. Approov [App] which claims to be able to attest app integrity.

We assume that the generation of an integrity attestation requires a challenge. The so-called *client data hash* must be used as challenge. It is the hash of the following stringified JSON object which contains top-level claims and the *request data* (see below). The client data binds the key attestation not only to the integrity nonce but also to the client, the authorizations server, the used endpoints, and the request data. Note, the JSON keys must be lexicographically ordered:

- *iss*: taken from the top level claims
- *aud*: taken from the top level claims
- *integrity nonce*: taken from the top level claims
- *source endpoint*: taken from the top level claims
- *target endpoint*: taken from the top level claims
- *request data hash*: hash of the request data

All hashes are base64url encoded hashes using a cryptographic secure hash function. At the given time we require the usage of sha256.

The integrity nonce has been generated by the authorization server and is bound to the client mTLS certificate. This nonce can be e.g. received from the integrity endpoint (see subsection 5.5.1). This endpoint is e.g. used during client and auth app registration. In another scenario, the nonce is additionally returned by the pushed authorization request endpoint.

We use in our proposal the request body as *request data*. This approach could be extended by e.g. including headers or other data.

The authorization server must verify that the JWT is valid, that the JWT holds the expected values, and that each contained integrity attestation is valid with respect to the expected client data hash. This includes the verification of the binding of the integrity nonce. Since the integrity nonce is sender-constrained the oauth integrity attestation is sender-constrained. Depending on the contained attestations the authorization server can e.g. assume that device integrity is given. The oauth integrity attestation is required at the token endpoint and the request authorization endpoint (see subsection 5.5.9).

There are several disadvantages to consider when applying the oauth integrity attestation:

- *Header Size Problem:* The header might become quite larger. For example, Nginx or Node.js has a default allowed maximum header size of 8KB. In our implementation, we reach this limit when using a signed and encrypted JWT which contains a SafetyNet attestation. Therefore, we extend the limit for Node.js to 10KB when starting the authorization server. A possible solution would be to send the attestation inside the body by wrapping the original body in a structure that holds the attestation and the original body.
- *User Experience Problem:* Generating an integrity attestation using hardware-backed security takes enough time to be noticeable by the user. Our implementation makes heavy use of attestations which resolves in various loading screens.

- *Rate Limiting and Quota Problem:* The SafetyNet API is quota-restricted and rate-limited. This might lead to not successful flows.

We decided to keep the concept simple but want to mention several interesting specifications or drafts that can be used to improve our concept. The main reason next to simplicity why we did not integrate them is that we would have made too many adjustments to the respective specification or draft:

- *A Method for Signing HTTP Requests for OAuth* [RBT16] presents a way to sign HTTP requests. The corresponding signature is passed as header. The draft defines a JSON object that could be used as request data.
- *OAuth 2.0 Dynamic Client Registration Protocol* [RJB+15] defines the software statement which is used to attest client metadata during registration. Adjusting and extending this concept resolves in our concept.
- *HTTP Integrity Header* [Hal] presents the *integrity header* to transport integrity attestations. This header might be used to transport the oauth integrity attestation. We do not make any assumptions about which header is used to send the oauth integrity attestation. In our implementation, we use the custom header "x-oauth-integrity-attestation".

5.4 Key Attestation Challenge

We use the hash of the following stringified JSON object as key attestation challenge. The challenge binds the key attestation not only to the integrity nonce but also to the authorization server and the used endpoints. Note, the JSON keys must be lexicographically ordered:

- *aud*: the issuer of the authorization server
- *integrity nonce*: integrity nonce received from the authorization server
- *source endpoint*: URL of the endpoint from where the integrity nonce has been received, e.g. "https://as.dinobank.milesstoetzner.de/integrity"
- *target endpoint*: URL of the endpoint where this attestation is sent to, e.g. "https://as.dinobank.milesstoetzner.de/registration"

The hash is the base64url encoded hash using a cryptographic secure hash function. At the given time we require the usage of sha256.

5.5 Endpoints

In this section, we describe new endpoints that generally define the communication between the auth app and the authorization server. The most important endpoints are the following:

- *Integrity Endpoint* for retrieving an integrity nonce (see subsection 5.5.1)
- *App Authorization Endpoint* for implementing the authorization endpoint on the auth app (see subsection 5.5.4)

- *Request Introspection Endpoint* to introspect the authorization request (see subsection 5.5.6)
- *Request Authorization Endpoint* to exchange user consent for an authorization response (see subsection 5.5.9)
- *App Redirection Endpoint* for implementing the redirection endpoint on the client (see subsection 5.5.5)

If not stated otherwise the following statements hold. The endpoint is an HTTP POST endpoint and requires TLS as defined by FAPI 2.0 [Fetc]. Depending on the profile the request/ respond is either a JSON object or a signed and optional encrypted JWT. Attributes defined in requests/ responses are either top-level keys in a JSON object or top-level claims in a JWT. Any additional attributes can be added to the request/ response.

Each JWT contains the sender as issuer, e.g the auth app id, and the receiver as audience, e.g. the issuer of the authorization server. JWTs should be short-living, e.g. 10 seconds, and should contain *iat*.

Some endpoints require auth app authentication. The auth app is identified based on the issuer that needs to be present in each JSON and JWT.

Requests and responses must not be cached. We do not further specify errors.

5.5.1 Integrity Endpoint

The *integrity endpoint* is an endpoint at the authorization server for receiving an *integrity nonce*. The integrity nonce is used to generate an oauth integrity attestation (see section 5.3). This endpoint is used during client and auth app registration (see section 5.10 and section 5.11) and requires mTLS.

The *integrity nonce* is a fresh nonce and only valid for a short period, e.g. 10 seconds. The nonce is bound to the certificate of the caller as described in section 3.12. The authority of the caller's certificate is not checked, therefore, self-signed certificates can be used. There exists only one integrity nonce for the same certificate at the same time when using this endpoint.

The *integrity request* does not contain any attributes.

The *integrity response* contains only the newly generated integrity nonce as attribute *integrity nonce*.

5.5.2 Auth App Registration Endpoint

The *auth app registration endpoint* is an endpoint at the authorization server for registering an auth app instance (see section 5.11). The endpoint requires mTLS to verify the binding of the integrity nonce that has been received previously from the integrity endpoint. The authority of the certificate is not checked. The required oauth integrity attestation is validated based on the received software id.

The *auth app registration request* contains the following attributes:

- *software id*: identifier of the auth app software

- *jwks*: a JWKS that contains the public keys as described in section 5.2
- *username*: (optional) the authorization server might require a username for authenticating the user
- *password*: (optional) the authorization server might require a password for authenticating the user
- any additional attributes, e.g. *otp* for a multi-factor authentication
- if the response is a JWT then the type is *auth_app_registration_request+jwt* and the *issuer* is empty

The *auth app registration response* contains the following attributes:

- *auth app id*: contains a freshly generated auth app id
- any additional attributes, e.g. *user* containing a display name or available IBANs
- if the response is a JWT then the type is *auth_app_registration_response+jwt*

5.5.3 Auth App Deregistration Endpoint

The *auth app deregistration endpoint* is an endpoint at the authorization server for deregistering an auth app instance. The endpoints require auth app authentication.

The *auth app deregistration request* does not contain any attributes. If the request is a JWT then the type is *auth_app_deregistration_request+jwt*.

The *auth app deregistration response* does not contain any attributes. If the request is a JWT then the type is *auth_app_deregistration_response+jwt*.

5.5.4 App Authorization Endpoint

The *app authorization endpoint* is the authorization endpoint implemented by the auth app using an Android activity that is identified by the auth app metadata “apk authorization activity”. The endpoint authenticates the intent sender as the expected client. We explicitly define this endpoint since there are fundamental differences between redirecting the user between websites and between native applications.

The auth app adds the used request authorization endpoint to the authorization response to mitigate a misconfigured request authorization endpoint (see subsection 5.15.3). This information is encoded as “request_authorization_endpoint” query parameter in the redirect URI.

5.5.5 App Redirection Endpoint

The *app redirection endpoint* is the redirection endpoint implemented by the client app implemented by an Android activity that is identified by the client metadata “apk redirection activity”. The endpoint authenticates the intent sender as the expected auth app. We explicitly define this endpoint since there are fundamental differences between redirecting the user between websites and between native applications.

5.5.6 Request Introspection Endpoint

The *request introspection endpoint* is an endpoint at the authorization server for introspecting an authorization request. This endpoint requires auth app authentication. After introspecting the request URI gets bound to the auth app id.

The *request introspection request* contains the following attributes:

- *request uri*: the request URI of the request that should be introspected
- if the response is a JWT then the type is *request_introspection_request+jwt*

The *request introspection response* contains the following attributes:

- *request uri*: taken from the request. The auth app must verify that this value is as expected.
- *integrity nonce*: a newly generated integrity nonce which is valid as long as the request URI is valid and bound to the auth app id and request URI.
- *client details*: details about the client, e.g. client name, image URI, redirect URI, apk minimum version, apk signing certificates, and response mode.
- if the response is a JWT then the type is *request_introspection_response+jwt*

5.5.7 Request Update Endpoint

The *request update endpoint* is an endpoint at the authorization server for updating an authorization request. This endpoint requires auth app authentication. The binding of request URI to auth app id must be verified.

One use case for this endpoint is e.g. when the user can select between multiple bank accounts.

The *request update request* contains the request URI as *request uri* attribute and any additional use-case specific attributes, e.g an IBAN. If the response is a JWT then the type is *request_update_request+jwt*.

The *request update response* is the same as the request introspection response. If the response is a JWT then the type is *request_update_response+jwt*.

5.5.8 Request Delete Endpoint

The *request delete endpoint* is an endpoint at the authorization server for deleting an authorization request. This endpoint requires auth app authentication. The binding of request URI to auth app id must be verified.

The *request delete request* contains the request URI as *request uri* attribute. If the response is a JWT then the type is *request_delete_request+jwt*.

The *request delete response* contains the following attributes:

- *request uri*: taken from the request. The auth app must verify that this value is as expected.
- *authorization response*: (optional) contains a JARM containing the error *access_denied*
- If the response is a JWT then the type is *request_delete_response+jwt*

5.5.9 Request Authorization Endpoint

The *request authorization endpoint* is an endpoint at the authorization server for authorizing an authorization request. The auth app can exchange a user consent for an authorization response. This process is called *user consent exchange*. The endpoint requires auth app authentication and a valid oauth integrity attestation. The binding of request URI to the auth app id and to the request URI must be verified. The authorization server validates the authorization request parameters the same way as it would have validated the parameters at the authorization endpoint. In addition, the authorization server must verify that the user consent is a valid signature over the request introspection response using the user verification key.

The *request authorization request* contains the following attributes:

- *request uri*: the request URI of the current request
- *user consent*: the user consent as defined in section 5.6
- If the response is a JWT then the type is *request_authorization_request+jwt*

The *request authorization response* contains the following attributes:

- *request uri*: taken from the request. The auth app must verify that this value is as expected.
- *authorization code*: (only baseline) contains the authorization code
- *authorization response*: (only in advanced) contains the JARM
- If the response is a JWT then the type is *request_authorization_response+jwt*

5.6 User Consent

User consent is the signature over the request introspection response using a hardware-backed key that can only be unlocked by biometrics for a single cryptographic operation. The auth app exchanges the user consent for an authorization code at the request authorization endpoint.

The auth app generates a user signature key and registers the corresponding verification key at the authorization server when the user logs in at the auth app. The user signature key is hardware-backed, can only be accessed by the auth app for a single cryptographic operation, and is protected by biometrics. The user gives consent to an authorization request by successfully passing the local biometric authentication. Note, that this authentication authenticates the device user.

The result for the authentication is a signature over the request introspection response, the so-called *user consent*. The auth app exchanges the user consent for an authorization code at the request authorization endpoint. We call this exchange the *user consent exchange*. A strong signature algorithm must be used, e.g. *Sha256withECDSA*.

User consent is among others bound to the information that has been displayed to the user, especially to the request URI. The request URI gets bound at the request introspection endpoint to the introspecting auth app. Furthermore, user consent provides non-repudiation and is sender-constrained.

If the request update endpoint has been used during the flow then the user consent signs the request update response.

We do not make use of Android Protected Confirmation since this mechanism does not support user authentication (see subsection 2.1.7).

5.7 Application Authentication

Application Authentication refers to first verifying that the package being authenticated holds the expected package name and secondly comparing the signing certificates of the package being authenticated against expected certificates. Simply checking if the installed package holds the expected package name is not sufficient since the package might be e.g. repackaged.

This concept is presented by Chen et al. [CPC+14] and is the “only reliable mechanism by which you can authenticate your peer” [Chrb]. Additionally, the version of the installed package should be checked. This enables e.g. to revoke insecure implementations. All required information can be retrieved from the package manager.

An intent receiver can be authenticated by first identifying the intent receiver and secondly by authenticating the identified receiver. The same applies when authenticating an intent sender. For more information about identifying the intent receiver/sender see our discussion in subsubsection 2.3.2.5.

Note, that application authentication only verifies that the expected package has been installed and does not e.g. authenticate a specific application instance (see subsection 7.3.4).

5.8 Secure Redirect

During the flow, the user is redirected from the client to the auth app and back. Redirecting the user means to start an activity in another application. We require integrity, confidentiality, source authentication, and target authentication when redirecting the user. Roughly speaking, this means that the user is redirected from the correct source app to the correct target app and that no attacker is able to read or write the sent data. The motivation for a secured redirect is to mitigate attacks as soon as possible and to preserve the integrity of the authorization request and response. An attacker should e.g. not be able to swap the request URI or tamper with the used request authorization endpoint which is returned to mitigate a misconfigured request authorization response (see subsection 5.15.3).

In this section, we present the result of our discussion in section 7.3. The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback (see subsection 7.3.2).

The client redirects the user to the auth app using an explicit intent and registers a result callback. The redirect is mutually authenticated. The client authenticates the intent receiver as the expected auth app and the auth app authenticates the intent sender as the expected client. The required information is e.g. distributed via the authorization server's metadata or returned by the request introspection response. The intent is called *auth intent* and is constructed as follows:

- "apk package name" of the auth app as the "intent package name"
- "apk authorization activity" of the auth app as the "intent class name"
- the authorization URI as "intent data"

The auth app uses the result callback to redirect the user back to the client. The intent that is passed to the result callback is called *result intent* and is constructed as defined by the response mode (see section 5.9). Generally, the result intent holds the redirect URI as intent data with the authorization code encoded.

Only the auth app is required to expose an activity, the *apk authorization activity*. Since a result callback is used the client does not need to expose any component.

5.9 Response Modes

In this section, we introduce new response modes that define how the result intent is constructed and sent. We explicitly define app2app specific response modes for a better discussion, future changes and to prevent misconfiguration and misunderstandings.

- "intent.result.query" uses a result callback and authenticates the sender before redirecting the user. The intent data holds the redirect URI with the authorization code encoded as defined by "query".
- "intent.result.extra" uses a result callback and authenticates the sender before redirecting the user. The intent data holds the redirect URI while the intent extra "EXTRA_OAUTH_CODE" contains the authorization code.

- “intent.query” uses an explicit intent and authenticates the receiver before redirecting the user. The authorization response is passed as defined by “intent.result.query”.
- “intent.extra” uses an explicit intent and authenticates the receiver before redirecting the user. The authorization response is passed as defined by “intent.result.extra”.
- “query” is an already existing response mode. An implicit intent will be used to redirect the user back. The result is encoded as a query parameter in the intent data.

For JARM we introduce the following response modes:

- “intent.result.query.jwt” is the same as “intent.result.query” but the used redirect URI is as defined by “query.jwt”.
- “intent.result.extra.jwt” is the same as “intent.result.query.jwt” but the result is passed in the intent extra “EXTRA_OAUTH_RESPONSE”.
- “intent.query.jwt” is the same as “intent.query” but the used redirect URI is as defined by “query.jwt”.
- “intent.extra.jwt” is the same as “intent.query.jwt” but the result is passed in the intent extra “EXTRA_OAUTH_RESPONSE”.
- “query.jwt” is an already existing response mode defined by JARM. An implicit intent will be used to redirect the user back. The result is encoded as a query parameter in the intent data.

Based on our discussion of a secure redirect and since intent extras introduce a new concept we require using “intent.result.query” or “intent.result.query.jwt” (see section 7.3).

The response mode generally does not make any assumption about how the user has been redirected to the auth app. For example, a result callback can be registered using an implicit or explicit intent.

5.10 Client Registration

When the client is started for the first time on the device the client generates the client data key, a hardware-backed key with time-limited access for managing the data that belongs to the client, e.g. access tokens. To get access to this key the user is locally authenticated in step 1 using the device credentials.

The client generates a client registration certificate and uses the certificate to request an integrity nonce from the integrity endpoint in step 3. The authorization server binds the integrity nonce to the client registration certificate. With the knowledge of the integrity nonce the client is able to generate the remaining keys along with their key attestations. The challenge in section 5.4 defined key attestations challenge is used for that. The challenge contains the integrity attestation, the issuer of the authorization server, and the used integrity and registration endpoint.

The client generates the oauth integrity attestation next. Therefore, any integrity attestation that the authorization server requires is generated. For example, in step 5 the client data hash is used as a challenge for a SafetyNet attestation. The client computes the required client data hash using the previously fetched integrity nonce and as request data the registration request body. After generating the oauth integrity attestation the client registers in step 7 as a confidential oauth client and receives

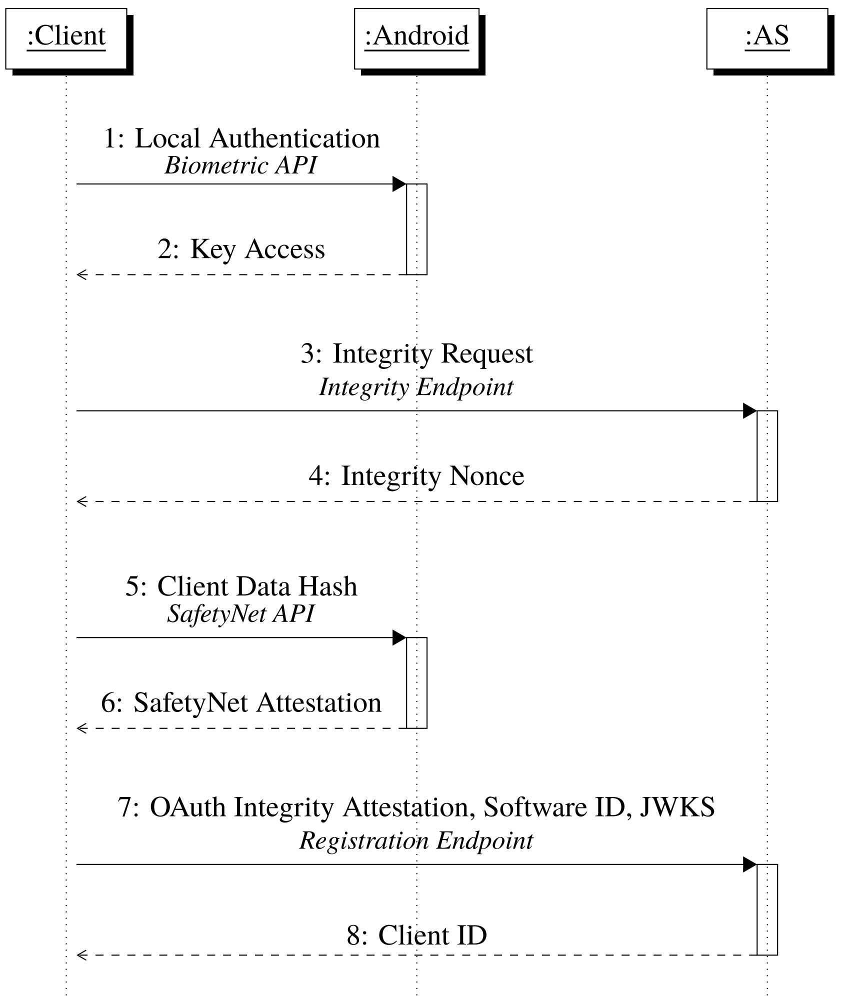


Figure 5.1: Client Registration

his client id. The registration request contains the oauth integrity attestation, a software id, and a JWKS containing the client's public keys along with their key attestation. The authorization server verifies the oauth integrity attestation based on the software id and the used mTLS certificate and verifies each key attestation. Based on the attestations the authorization server can assume that the request has been sent from an untampered client running on an untampered device.

The client uses for future communication the client certificate as mTLS client certificate. The client registration certificate is only required to bind the integrity nonce during the registration to the caller. For more information about this requirement see the *The Requester Verification Problem* in section 2.5. Note, that the client can not generate the client certificate and the corresponding key attestation without the knowledge of the integrity nonce.

5.11 Auth App Registration

The auth app registration (see Figure 5.2) is very similar to the client registration (see section 5.10). The registration differs only in the endpoint used and that the auth app prompts the user for his credentials.

The auth app registration request contains an oauth integrity attestation, a software id, a JWKS, and all required information to authenticate the user, e.g. username and a one-time password. We assume that the user is already registered at the authorization server. The authorization server verifies the oauth integrity attestation based on the software id and the used mTLS certificate and verifies each key attestation. The auth app registration response contains an auth app id and e.g. information about the user. In our implementation, we return a display name and an IBAN. Another example is to return a list of IBANs from which the user chooses during authorizing a transaction (see subsection 5.5.7).

Note, that the auth app is required to identify the authorization server if several authorization servers are supported. Identification is out of scope but might be done by the user entering a BIC or bank name. For example, the banking app of the Sparkasse [Spa] supports multiple authorization servers. Each authorization server belongs to a Sparkasse branch.

5.12 Baseline Profile

The *FAPI 2.0 Android App2App Baseline Profile* is based on the existing baseline profile (see section 4.2) and has been designed to reach the security goals as defined in section 7.1 with respect to our attacker model as defined in section 7.2. In the following, we describe the requirements that are additional to the baseline profile. These requirements include e.g. requirements for a secure redirect or hardware-backed keys.

We set the following general requirements:

- The user must be redirected as described in section 5.8.
- The response mode “intent.result.query” or “intent.result.query.jwt” must be used.
- An oauth integrity attestation must contain a SafetyNet attestation.
- Native applications must be MASVS-L2 conform.
- Native applications should implement appropriate resilience requirements.
- Packages must be signed using the APK signature scheme v2 or newer.
- A mitigation against the second and third attack in subsection 5.15.1 must be implemented. For example, the client must send the used token endpoint along with a resource request and the resource server must verify the used token endpoint.
- A mitigation against the attack in subsection 5.15.2 must be implemented. For example, the client must send the used auth app along with a token request and the authorization server must verify the used auth app.

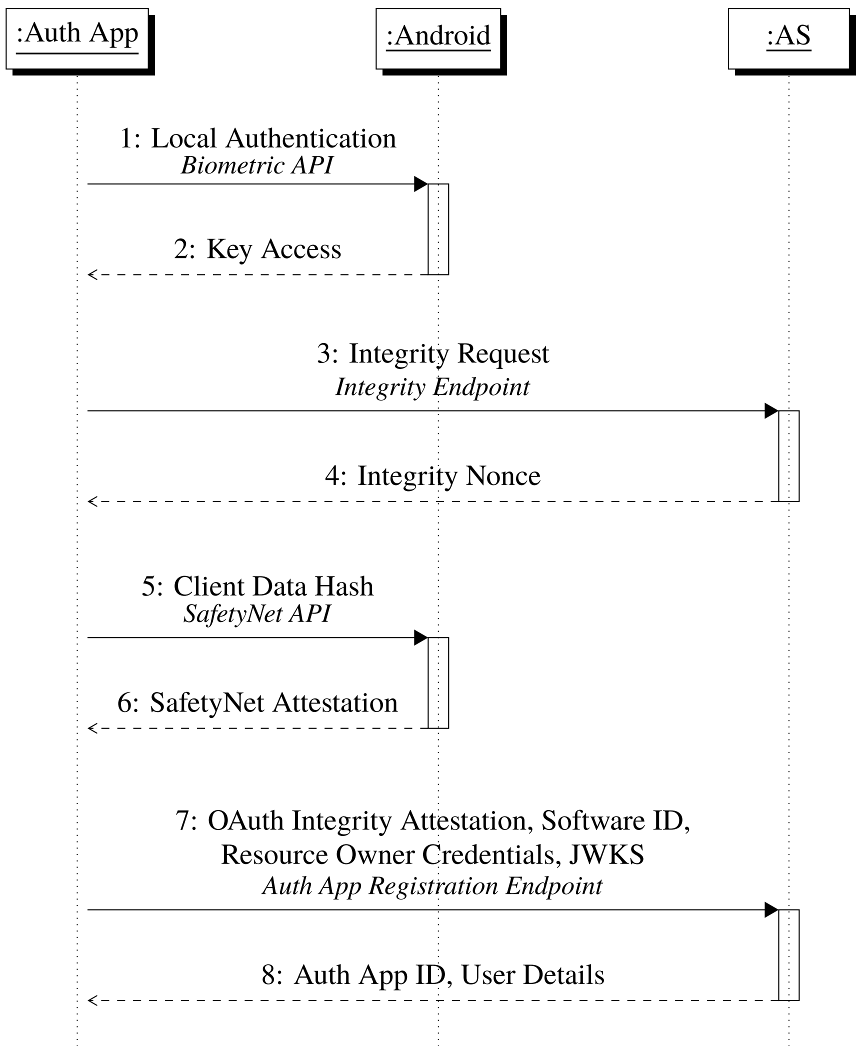


Figure 5.2: Authorization App Registration

- A mitigation against the attack in subsection 5.15.3 must be implemented. For example, the auth app must send the used request authorization endpoint which is verified by the authorization server during a token request.
- *JSON Web Token Best Current Practices* [SHJ20] must be followed, e.g. the *Cross-JWT Confusion* must be mitigated. An attacker might try to use the signed request introspection response as an access token at the resource server. The resource server can e.g. mitigate this by verifying the JWT type.

We set the following requirements for authorization servers:

- The registration endpoint must require an oauth integrity attestation. The corresponding integrity nonce is issued at the integrity endpoint.
- The pushed authorization request endpoint must additionally return an integrity nonce which is valid as long as the request URI.

- The token endpoint must require an oauth integrity attestation. The corresponding integrity nonce is issued at the pushed authorization request endpoint.
- The authorization server must verify the key attestation for each hardware-backed key. We describe the requirements for hardware-backed keys below.
- The authorization server should consider deregistration of clients and auth apps that have not been used for a while.

We set the following requirements for clients:

- The client must register as described in section 5.10.
- The client must verify that the result intent data matches the redirect URI.
- The client must follow the "apk preference" authorization server metadata.
- When the user logs out the respective app must reset the data and deregister at the authorization server. This especially means that all keys and tokens are deleted. Additionally, any valid token should be revoked (see [LDS13]).

We set the following requirements for auth apps:

- The auth app must register as described in section 5.11.
- The auth app must verify that the auth intent data matches the authorization URI.
- The auth app must use the request introspection endpoint to introspect the request URI.
- The auth app must use the request update endpoint to update the request.
- The auth app must use the request authorization endpoint to exchange the user consent for an authorization code.
- The auth app must delete the request at the request delete endpoint if the user aborts the flow.
- The auth app must warn the user during login when a new biometric has been registered (see subsubsection 7.4.2.2).

We set the following requirements for hardware-backed keys and state which fields of the key attestation must be verified (see [Andaq]). We discuss in subsection 7.6.3 why we do not require biometrics in general.

- The challenge is constructed as defined in section 5.4: The expected value must match "attestationChallenge".
- The key must be hardware-backed: "attestationSecurityLevel" must be either "1" or "2".
- Strongbox must be used if available.
- The key must be generated and not e.g. imported: "teeEnforced.origin" must be "0".

- The key must require user authentication when unlocking the key: “noAuthRequired” must not be present in “softwareEnforced” or “teeEnforced”.
- The key must require user confirmation when unlocking the key. If the keystore has e.g. “exclusive control of the scanner and performs the fingerprint matching process” [Andaq] then “teeEnforced.trustedUserPresenceRequired” must be present. Only available for key attestations version 3.
- The key must require an unlocked device when unlocking the key. “teeEnforced.unlockedDeviceRequired” must be present. Only available for key attestation version 3.
- The validity duration must be as low as applicable: “teeEnforced.authTimeout” must be set to e.g. “300”.
- The key must not be accessible for all applications: “allApplications” must not be present in “softwareEnforced” or “teeEnforced”.
- The key must be only accessible by the creator: “attestationApplicationId” must be present in “softwareEnforced” or “teeEnforced” and must only contain information about the creator and no other application.
- The key must be invalidated if a new biometric is enrolled. We are not aware of a way to verify this.
- General information about the key, e.g. the purpose, might be evaluated.
- Information about the root of trust might be evaluated as an additional defense-in-depth, e.g. “verified boot state” which indicates if the device has been booted in a verified state or “verified boot key” which is the used root of trust. Note, we do not make use of these since we verify the SafetyNet attestation. SafetyNet uses these values under the hood.
- System information might be evaluated as an additional defense-in-depth, e.g. “os version” which contains the Android version. Note, we do not make use of these since we verify the SafetyNet attestation.

Additionally, to the requirements metioned above, we set the following requirements for the user signature key:

- The user signature key must require user authentication for each operation: “authTimeout” must not be present in “softwareEnforced” or “teeEnforced”.
- The user signature key must be protected by biometrics: “teeEnforced.userAuthType” must be set to a biometric auth type, e.g. “2” for fingerprint.

We set the following requirements for generating and validating a SafetyNet attestation:

- “client data hash” of the oauth integrity attestation is used as challenge.
- “evaluation type” must be “BASIC,HARDWARE_BACKED”.
- “cts profile match” must be true.
- “apk digest sha256” might be evaluated.

5.13 Advanced Profile

The *FAPI 2.0 Android App2App Advanced Profile* is based on our app2app baseline profile (see section 5.12) and the advanced profile (see section 4.3). Our advanced profile has been designed to reach the security goals as defined in section 7.1 and the non-repudiation requirements as defined in section 7.5 with respect to our attacker model as defined in section 7.2.

Additionally, to already existing requirements, we require non-repudiation for the user consent and the communication between the auth app and the authorization server. For more information see section 7.5.

5.14 Redirect Flow

In the following, we describe the flow when applying our app2app baseline profile.

Before starting the flow the client locally authenticates the user to get access to the client data key (see step 1). The client pushes the authorization request to the pushed authorization request endpoint in exchange for a request URI and integrity nonce in step 3. Part of the authorization request is a freshly generated code challenge using sha256.

The client constructs the *auth intent*, an explicit intent based on the apk package name and apk authorization activity of the auth app. After authenticating the intent receiver as the auth app the client redirects the user to the app authorization endpoint by starting an activity using the auth intent and registers a result callback (see step 5). The intent is sent to the auth app via the OS.

The auth app first locally authenticates the user to get access to the auth app data key (see step 7). If the native application has multiple registered auth apps then the correct auth app must be identified. This can be either done by a selection dialog or the required information is encoded in the request URI.

The auth app verifies that the auth intent data matches the authorization URI and introspects the request URI at the request introspection endpoint in step 9. The request introspection endpoint returns authorization information including an integrity nonce. After authenticating the intent sender as the expected client the auth app displays request information and asks for user consent. The user verifies the information and authorizes the request by passing e.g. a fingerprint prompt (see step 11). Android issues the user consent which is a signature over the request introspection response using the user signature key (see step 12). Note, the user might change information like e.g. the bank account if his user account has access to several bank accounts at that bank. In such a case the auth app updates the authorization request using the request update endpoint and the request update response is used for the signature of the user consent.

The auth app exchanges the user consent and request URI for an authorization code at the request authorization endpoint (see step 15). The endpoint requires an oauth integrity attestation. The attestation is generated using a client data hash based on the integrity nonce returned by the request introspection endpoint and the request authorization request body. The attestation contains a SafetyNet attestation which uses the data hash as challenge (see step 13).

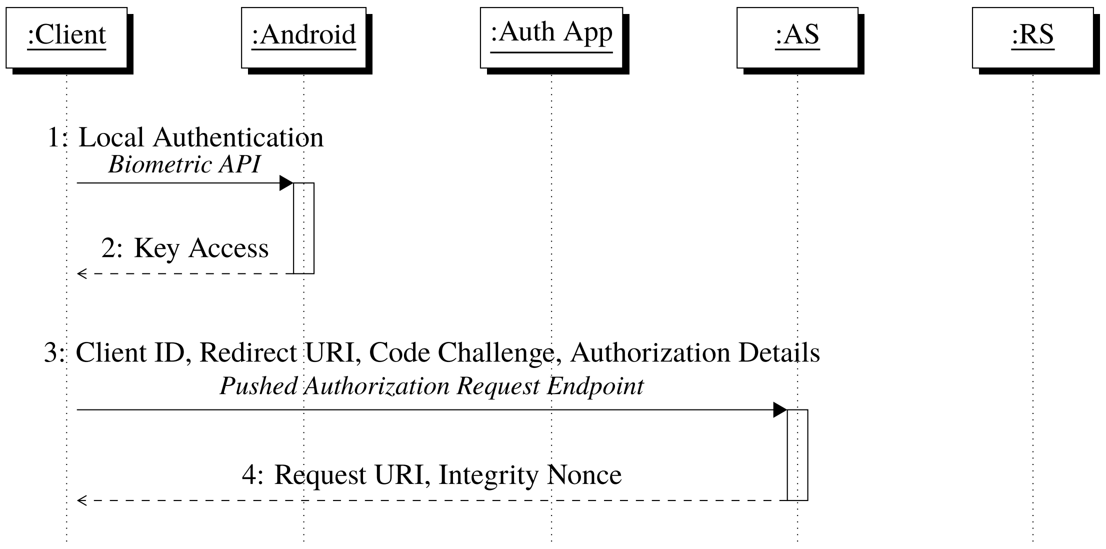
The authorization server validates the authorization request parameters at the request authorization endpoint the same way as it would have validated the parameters at the authorization endpoint. After verifying the oauth integrity attestation and user consent the authorization server issues and returns a sender-constrained authorization code.

The auth client redirects the user to the app redirection endpoint at the client using the result callback (see step 18). The result intent passed to the result callback contains the redirect URI as intent data. The result intent is sent to the client via the OS. The authorization code and the used request authorization endpoint (see subsection 5.15.3) are encoded in the redirect URI.

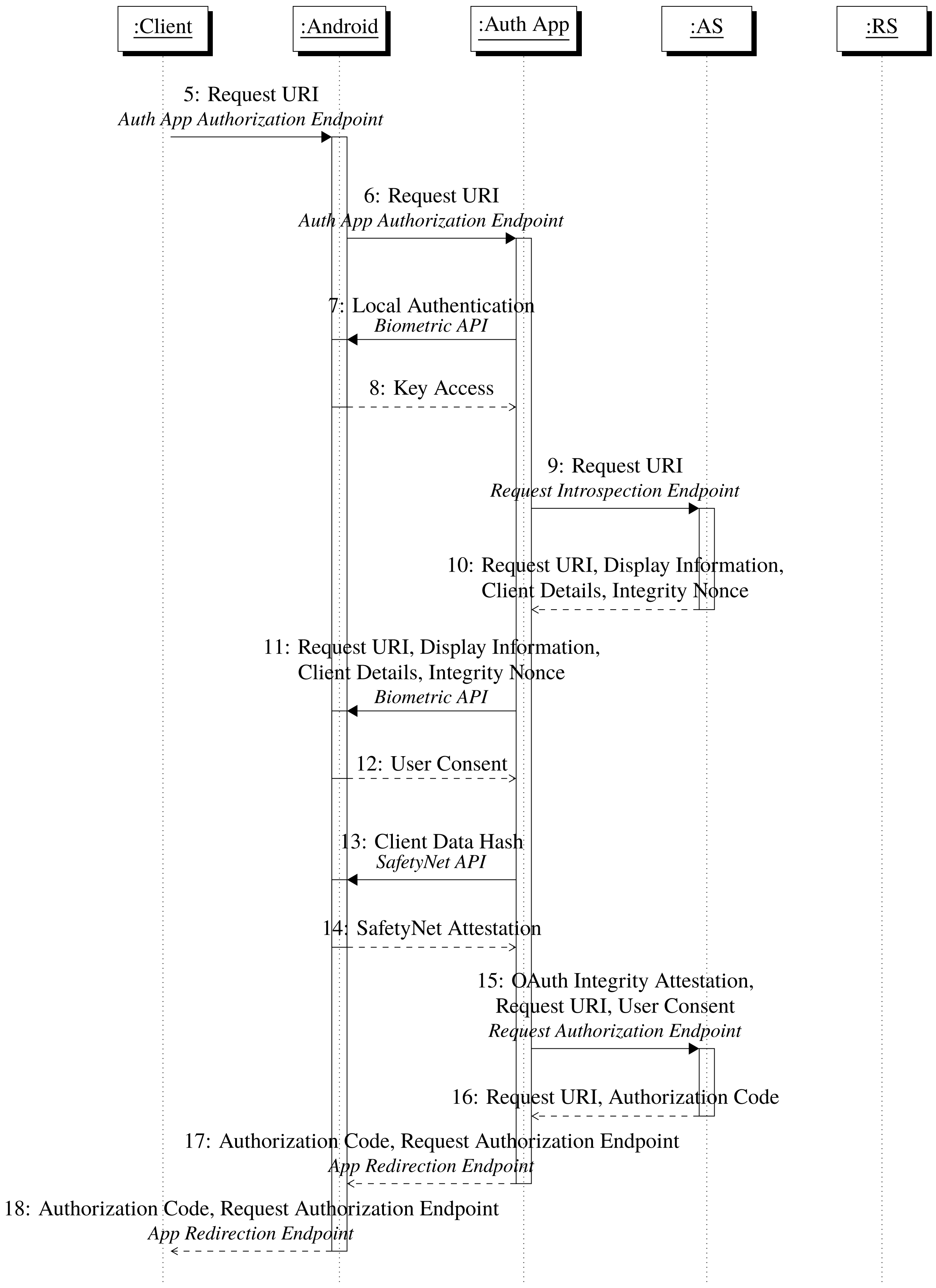
The client verifies that the result intent data matches the expected redirect URI. After that, the client exchanges the authorization code at the token endpoint for an access token (see step 21). The endpoint requires an oauth integrity attestation. The attestation is generated using a client data hash based on the integrity nonce returned by the pushed authorization request endpoint and the token request body. The attestation contains a SafetyNet attestation which uses the data hash as challenge (see step 19). The authorization server verifies the SafetyNet attestation and performs the usual checks.

Additionally, the client sends the used auth app and the used request authorization endpoint to the token endpoint. The authorization server verifies that the client used the correct auth app and that the auth app used the correct request authorization endpoint. This mitigates attacks based on misconfigurations. For more information see subsection 5.15.2 and subsection 5.15.3.

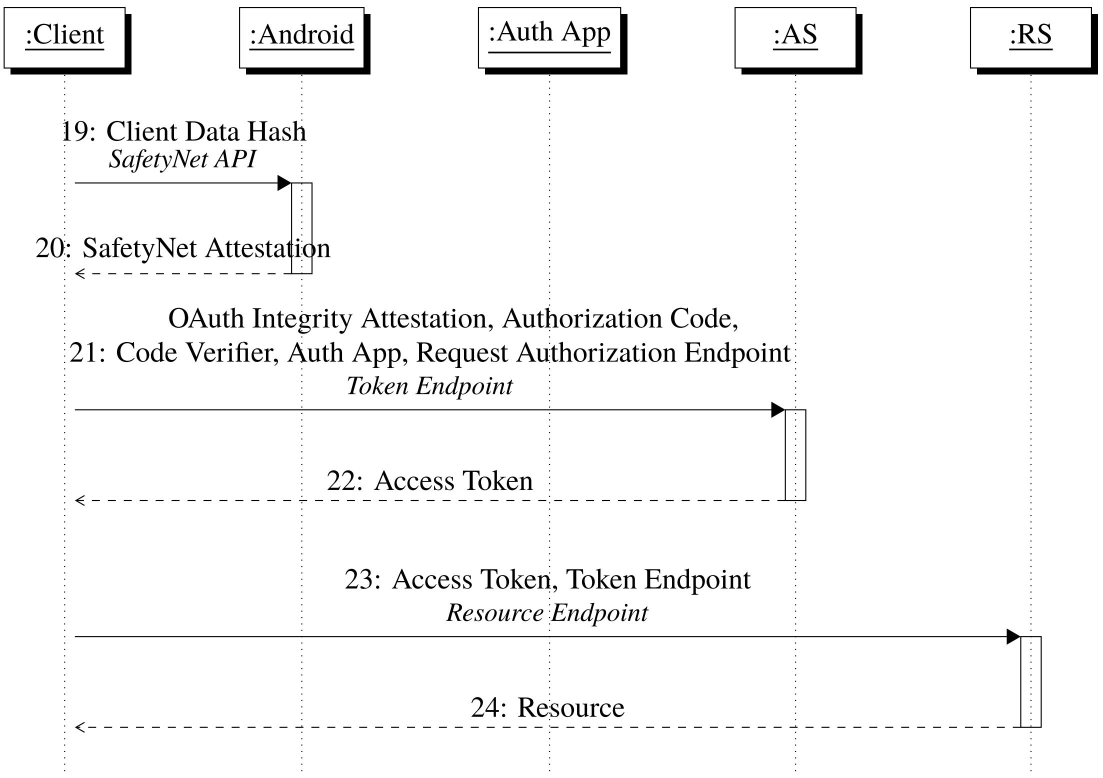
The client requests the resource at the request endpoint (see step 23) using the client certificate and the access token. Additionally, to the access token the client sends the used token endpoint to the resource server. After verifying the access token, the token binding and that the client used the correct token endpoint, the resource server returns the resource (see subsection 5.15.1).


:Client
:Android
:Auth App
:AS
:RS
1: Local Authentication
Biometric API
2: Key Access
3: Client ID, Redirect URI, Code Challenge, Authorization Details
Pushed Authorization Request Endpoint
4: Request URI, Integrity Nonce

(a) Pushed Authorization Request Part


:Client
:Android
:Auth App
:AS
:RS
5: Request URI
Auth App Authorization Endpoint
6: Request URI
Auth App Authorization Endpoint
7: Local Authentication
Biometric API
8: Key Access
9: Request URI
Request Introspection Endpoint
10: Request URI, Display Information, Client Details, Integrity Nonce
11: Request URI, Display Information, Client Details, Integrity Nonce
Biometric API
12: User Consent
13: Client Data Hash
SafetyNet API
14: SafetyNet Attestation
15: OAuth Integrity Attestation, Request URI, User Consent
Request Authorization Endpoint
16: Request URI, Authorization Code
17: Authorization Code, Request Authorization Endpoint
App Redirection Endpoint
18: Authorization Code, Request Authorization Endpoint
App Redirection Endpoint

(b) Authorization Part


:Client
:Android
:Auth App
:AS
:RS
19: Client Data Hash
SafetyNet API
20: SafetyNet Attestation
OAuth Integrity Attestation, Authorization Code,
21: Code Verifier, Auth App, Request Authorization Endpoint
Token Endpoint
22: Access Token
23: Access Token, Token Endpoint
Resource Endpoint
24: Resource

(c) Access Part

Figure 5.2: App2App Redirect Flow

5.15 Mitigations

In the following, we present mitigations for the assumptions that the client might use a misconfigured auth app and that the used auth app might use an misconfigured request authorization endpoint. We do not explicitly mitigate a misconfigured request introspection endpoint since this misconfiguration is implicitly mitigated by the user consent. Additionally, we mitigate known attacks on FAPI 1.0. The mitigations are required in our baseline profile.

5.15.1 FAPI 1.0 Attacks

There had been several attacks found on FAPI 1.0 [Dam18]. The first attack (see 6.1 [Dam18] makes use of the fact that PKCE does not make any assumption about who generated the code challenge but only that only the creator knows the corresponding code verifier. This enables a malicious client to generate an authorization request in the name of an honest public client by using the client id of the honest client and generating a code challenge and verifier on its own. The corresponding authorization response leaks to the attacker who is able to obtain and use an access token of an honest user. This attack can be mitigated by authenticating the client. Therefore, this attack is mitigated by FAPI 2.0.

In the second attack (see 6.2 of [Dam18]) the client chooses a wrong authorization server which is under the control of an attacker who successfully injects a leaked access token at the token endpoint. The proposed mitigation is to send the issuer of the authorization server along with the resource request. This information can be taken from the id token that is always returned from the token endpoint during the FAPI 1.0 read-write flow. The resource server additionally verifies that the used access token has been returned from the correct issuer.

In the third attack (see 6.3 of [Dam18]) the client is misconfigured and uses a token endpoint that is under control of the attacker which returns a leaked access token. The proposed mitigation is to add the hash of the access token to the id token. An attacker can not be in possession of such an id token which mitigates the attack [Dam18].

The proposed mitigations of the third attack can not be applied to FAPI 2.0 since an id token is only returned from the token endpoint if authentication of the user is requested. After a discussion with Damabi we use a mitigation that mitigates the second and the third attack. The client sends the used token endpoint along with the resource request. The resource server additionally verifies that the used access token has been returned from the correct token endpoint. The correct token endpoint might be part of the access token. How the token endpoint is passed during the resource request or how the resource server knows the correct token endpoint is out of scope.

In the fourth and last attack (see 6.4 [Dam18]) an attacker authorized a leaked authorization request of an honest client and forges the authorization response back into the client. This results in the honest user having access to the attackers resources or being logged in under the identity of the attacker. No proposed mitigation that can be applied to all scenarios. One mitigation is to authenticate the sender of the authorization request as the honest client. This is possible in our case since we authenticate the intent sender as the expected client app. Therefore, this attack is not possible in our case.

5.15.2 Misconfigured Auth App

The client might be misconfigured and is using an auth app under the control of an attacker. We assume that the attacker is in possession of a valid authorization code that belongs to the client and is related to the resources of the attacker. The attacker returns the authorization code in the intent result callback and the client successfully uses the code at the honest authorization server. This breaks session integrity for authorization.

We can apply the same mitigation concept as for the attacks in subsection 5.15.1. The client informs the authorization server about the used auth app: The client sends the apk package name and apk signing certificates used for the secure redirect to the token endpoint. The token endpoint verifies that the authorization code has been returned from the correct auth app.

5.15.3 Misconfigured Request Authorization Endpoint

The auth app might be misconfigured and is using a request authorization endpoint under the control of an attacker. This scenario enables the same consequences as described in subsection 5.15.2. The attacker can inject a leaked authorization code into the flow by returning the authorization code as part of the request authorization response.

We can mitigate this by informing the authorization server about which request authorization endpoint has been used. The auth app sends the used request authorization endpoint in the authorization response to the client and the client sends the used request authorization endpoint along with the token request. The authorization server verifies that the authorization code has been returned from the correct request authorization endpoint.

The advanced profile implicitly mitigates this attack since the request authorization endpoint returns a JWT that contains the request URI. Therefore, the integrity of the response is preserved and the response can not be replayed during another flow. The attacker can not be in possession of a JWT signed by an honest authorization server that contains the request URI and authorization code of different flows. Note, that the request URI is a nonce that is freshly generated for each flow.

Note, if the misconfigured endpoint is under control of an honest authorization server then auth app authentication is required. Since auth apps are isolated from each other no auth app has access to the credentials of another auth app. Therefore, the authorization server will reject the request since the auth app is not registered at this authorization server.

5.15.4 Misconfigured Request Introspection Endpoint

The auth app might be misconfigured and is using a request introspection endpoint under the control of an attacker. The same considerations concerning the endpoint being under the control of an honest authorization server as discussed in subsection 5.15.3 apply.

User consent is a signature over the request introspection response. The client verifies that the response contains the correct request URI. If the attacker tampers with the response then the resulting user consent is a signature over information that does not match the request URI. The user consent would be therefore invalid for every flow. Note, the request URI is a nonce generated for each flow.

Additionally, the response contains an integrity nonce. A corresponding valid oauth integrity attestation is required at the request authorization endpoint. The integrity nonce is generated at the honest authorization server and is bound to an auth app certificate and the request URI. An attacker can not generate or request such an integrity nonce.

Therefore, a misconfigured request introspection endpoint is implicitly mitigated.

5.16 Error Handling

When the user does not give consent or aborts by pressing the back button on the device, the request should be aborted using the request delete endpoint. During the advanced profile, the endpoint will return a JARM which can be returned to the client as usual.

If there had been a validation error during the communication with the authorization server then the error is returned as query parameter to the client. This applies also to the advanced profile since the auth app can not obtain a valid JARM from the authorization server.

The client will handle the received error as usual. Note, that automatic redirects on errors is not permitted [Har12].

6 Implementation

In this chapter, we present the proof of concept implementation of our advanced profile using signed and encrypted JWTs. We implemented a banking app for our fictional *Dino Bank* and a digital wallet app, called *Panda Wallet*. A user can display his bank account at Panda Wallet and transfer money. We make use of RAR for expressing fine-grained authorization requests. Therefore, each financial transaction requires a separate OAuth 2.0 flow.

The implementation supports most of the in section 5.9 mentioned response modes, e.g. "query.jwt" which uses an implicit intent that can be intercepted. Panda Wallet can be configured to use any supported scenario. In fact, we implemented a malicious application, the *Shark Bank*, and show how the malicious application can intercept an authorization response (see subsection 6.2.5).

During our design process, we developed two experimental concepts which we implemented to further study their implications. The *request code* is an authorization code that is returned by the pushed authorization request endpoint (see subsection 7.6.11). The *auth app id* request parameter contains the auth app id to bind the request URI to the auth app instance running on the same device as the client.

We additionally implemented an app2web flow to analyse the redirecting possibilities. The app2web flow is non-functional and will not return an authorization response. The redirect is secured by browser whitelisting (see subsubsection 7.3.6.1).

The implementation is not production-ready. There are insecure mechanisms and settings implemented to analyse them. For example, the authorization server accepts safety attestations that are not hardware-backed to allow Emulators.

Our implementation requires an oauth integrity attestation for each endpoint at the authorization server and resource server. This is not required in our profiles. Additionally, the resource endpoint uses signed and encrypted JWTs for requests/ responses.

6.1 Parties

In the following, we describe some implementation details of each party. Our scenario consists of *Panda Wallet* as the client and *Dino Bank* as the auth app. The auth app belongs to an authorization server and resource server. Additionally, our setup contains a malicious application, so-called *Shark Bank*.

6.1.1 Panda Wallet

The *Panda Wallet* is a digital wallet app implemented as native application written in Kotlin for the sdkVersion 29 resp. In our implementation Android 10. Panda Wallet is the client. Users can link their Dino Bank account to access the account information and to transfer money. The client is using RAR for fine-grained authorization requests and implements the "auth app id" parameter (see subsection 7.6.12).

The Panda Wallet software holds the following metadata that is preconfigured at the authorization server:

- *software id*: "urn:android-package:de.milesstoetzner.pandawallet"
- *client name*: "Panda Wallet"
- *redirect uris*: "https://www.pandawallet.milesstoetzner.de/redirect_uri/dino_bank"
- *apk package name*: "de.milesstoetzner.pandawallet"
- *apk redirection activity*: "de.milesstoetzner.pandawallet.RedirectActivity"
- *apk signing certificates*: "OgyjCBWFZqmk0uuhYvqKRCnXPmfufBRyjqFDbTMTHf0"
- *response types*: "ping" and "code"

For encrypting the client data we use *encrypted shared preferences* [Andt]. The hardcoded SafetyNet API key is restricted and bound to the apk package name and the apk signing certificates.

We implemented a settings page that can be used to configure the client. The client supports the authorization code grant, refresh token grant, and request code grant (see subsection 7.6.11).

An implicit or explicit intent can be used as auth intent. Implicit intents are secured by browser whitelisting.

The client can be configured to set the intent referrer of the auth intent to a trusted or untrusted browser. The auth app verifies the intent referrer using browser whitelisting.

Screenshots of the Panda Wallet are framed with a brown border.

6.1.2 Dino Bank

The *Dino Bank* is a banking app implemented as native application written in Kotlin for the sdkVersion 29 resp. Android 10. Dino Bank is the auth app for our authorization server. A user can log in and authorize requests.

The Dino Bank software holds the following metadata that are preconfigured at the authorization server:

- *software id*: "urn:android-package:de.milesstoetzner.dinobank"
- *apk package name*: "de.milesstoetzner.dinobank"
- *apk authorization activity*: "de.milesstoetzner.dinobank.AuthorizationActivity"
- *apk authorization service*: "de.milesstoetzner.dinobank.AuthorizationService"

- *apk signing certificates*: “OgyjCBWFZqmk0uuhYvqKRCnXPmfufBRyjqFDbTMTHf0”

Note, that Panda Wallet and Dino Bank are signed by the same debug certificate. For encrypting the client data we use *encrypted shared preferences* [Andt]. The hardcoded SafetyNet API key is restricted and bound to the apk package name and the apk signing certificates.

Screenshots of the Dino Bank are framed with a blue border.

6.1.3 Shark Bank

The *Shark Bank* is a malicious native application written in Kotlin for the sdkVersion 29 resp. Android 10. The intent filter of the Shark Bank registers the authorization endpoint of the Dino Bank authorization server and the redirection endpoint of the Panda Wallet. In subsection 6.2.5 we describe an attack where the Shark Bank intercepts an authorization response and obtains a valid authorization code of an honest user when the client uses “query.jwt” as response mode.

Screenshots of the Shark Bank are framed with a red border.

6.1.4 Authorization Server

Our authorization server is a Node.js Express server written in TypeScript. It uses a self-signed certificate for TLS which is pinned at the Panda Wallet and Dino Bank.

The authorization server holds the following metadata:

- *issuer*: “https://as.dinobank.milesstoetzner.de”
- *authorization endpoint*: “https://as.dinobank.milesstoetzner.de/oauth/authorization”
- *apk preference*: “preferred”
- *authorization data type supported*: “account” and “transaction”

We define two authorization detail types: “account” for read access of the balance and the transaction history (see 6.1) and “transaction” for write-access of a transaction (see 6.2).

The authorization server supports the authorization code, refresh token, and request code grant (see subsection 7.6.11). The server allows no hardware-backed SafetyNet attestation to enable the use of Emulators.

The oauth integrity attestation is sent via the custom header “x-oauth-integrity-attestation” and exceeds the default header size of Node.js. Therefore, we extended the limit from 8KB to 10KB.

Requirements concerning hardware-backed security are not enforced. For example, the authorization server allows a client or auth app running on an Emulator to register.

The authorization server has several seeded user accounts (see Table 6.1). All accounts have the same password which is prefilled in the Dino Bank login screen.

display name	username	password	IBAN
Alice Alison	alison	123456	DE02100100109307118603
Bob Bobson	bobson	123456	DE89500105178445712545
Eve Evson	evson	123456	DE27500105173332914374

Table 6.1: Seeded Users a the Dino Bank Authorization Server

6.1.5 Resource Server

Our resource server is a Node.js Express server written in TypeScript. It uses a self-signed certificate for TLS which is pinned at the Panda Wallet. The issuer of the resource server is “https://rs.dinobank.milesstoetzner.de”.

The server provides an API for fetching the balance and transaction history of an account and for creating a transaction. There are already seeded accounts and transactions. The resource endpoints use signed and encrypted JWTs for request/ response.

Each resource request requires an oauth integrity attestation. The corresponding integrity nonce is issued by an integrity endpoint implemented by the resource server.

6.1.6 Device

The implementation has been developed using the Android Emulator and tested on a Samsung Galaxy S10.

The Android Emulator is configured with a density of 560 and a resolution of 3040 x 1440. The Android Emulator is not blocked by the authorization server since the authorization server supports the SafetyNet attestation evaluation type “BASIC”. This support is not allowed by the baseline profile. Note, that at least one biometric and device credential must be registered.

Hardware-backed attestations are tested using the Samsung Galaxy S10.

6.2 Scenarios

In the following, we describe selected scenarios of our implementation from the view of the user. These include client and auth app registration, account linking, transaction authorization, an interception attack, and an app2web flow.

6.2.1 Client Registration

The client is locked until the user is locally authenticated (see Figure 6.1). The user is prompted to locally authenticate (see Figure 6.2). Note, that biometrics do not work. For more information see subsection 7.6.3.

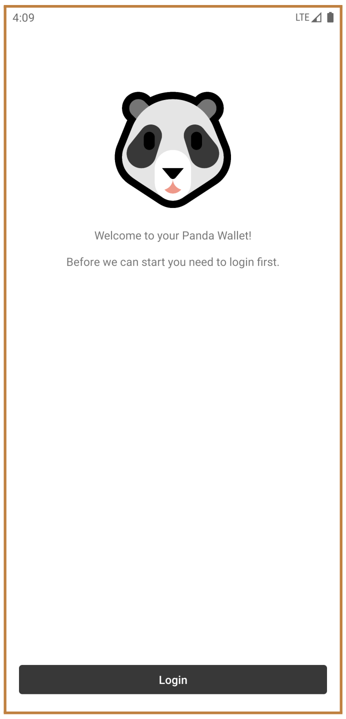

4:09
LTE
Welcome to your Panda Wallet!
Before we can start you need to login first.
Login

Figure 6.1: Locked Client

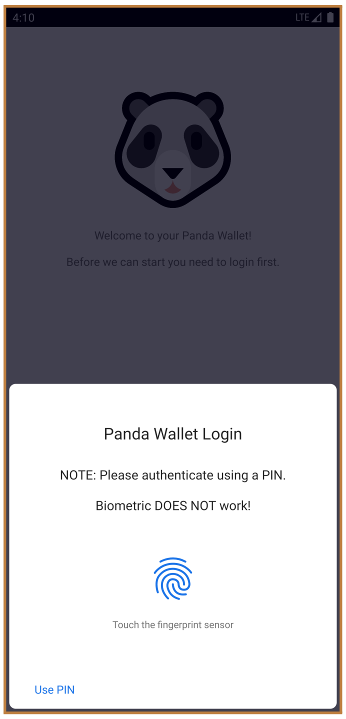

4:10
LTE
Welcome to your Panda Wallet!
Before we can start you need to login first.
Panda Wallet Login
NOTE: Please authenticate using a PIN.
Biometric DOES NOT work!
Touch the fingerprint sensor
Use PIN

Figure 6.2: Client Unlocking Prompt

The user enters his device PIN in Figure 6.3 and unlocks the client. After locally authenticating the user, the client registers at the authorization server in the background. In Figure 6.4 the client is registered and ready to use.

6.2.2 Auth App Registration

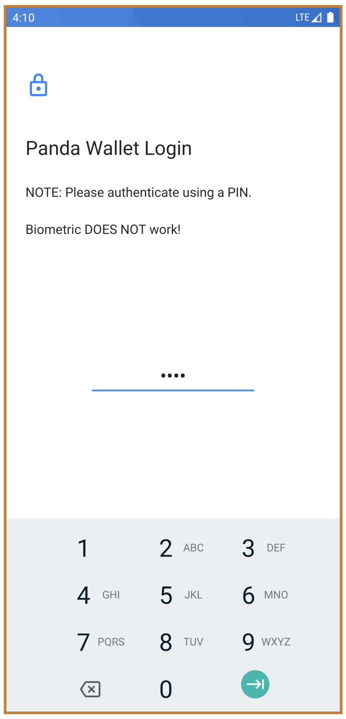
The auth app is locked until the user is locally authenticated. The process does not differ from the local authentication done by the client.

After unlocking the auth app the user is required to log in at the authorization server using a username and password (see Figure 6.5). In Figure 6.6 the auth app is registered at the authorization server and ready to use.

6.2.3 Account Read Access

The user can give the client read access to his bank account. The “account” authorization details are used (see Listing 6.1) for the authorization request. They request read access for account and transaction details.

The client redirects the user to the auth app which displays the request information (see Figure 6.7). If the user has not recently used the auth app then he is prompted to locally authenticate. The user authorizes the request using his fingerprint (see Figure 6.8). After that, the user is redirected back to the client (see Figure 6.9). The user can now e..g access his transactions (see Figure 6.10).



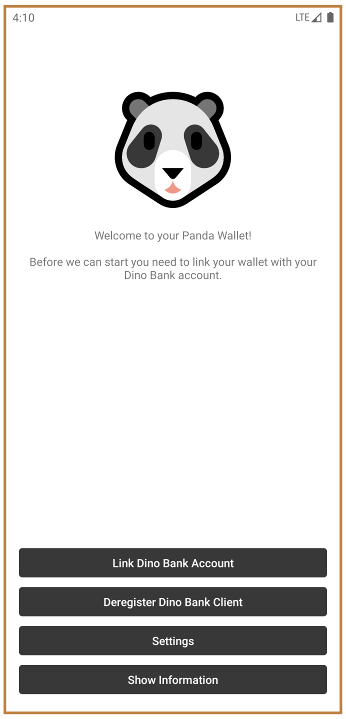
Panda Wallet Login

NOTE: Please authenticate using a PIN.

Biometric DOES NOT work!

••••

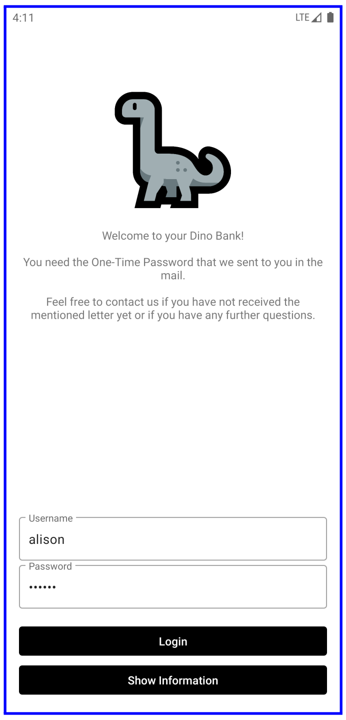
Figure 6.3: Client Local Authentication



Welcome to your Panda Wallet!

Before we can start you need to link your wallet with your Dino Bank account.

Figure 6.4: Client Home



Welcome to your Dino Bank!

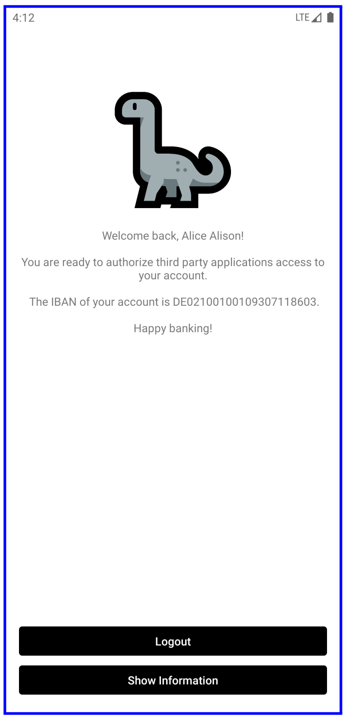
You need the One-Time Password that we sent to you in the mail.

Feel free to contact us if you have not received the mentioned letter yet or if you have any further questions.

Username: alison

Password: ••••••

Figure 6.5: Auth App Home



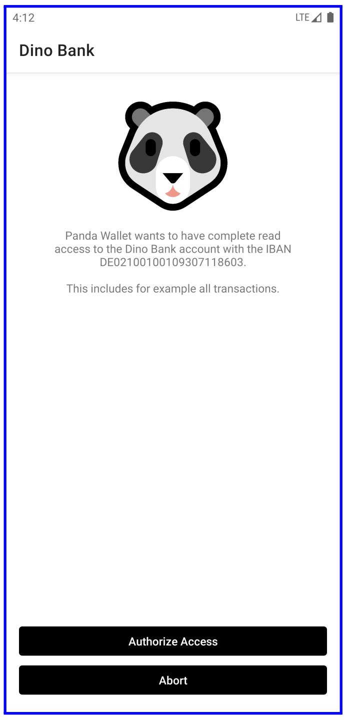
Welcome back, Alice Alison!

You are ready to authorize third party applications access to your account.

The IBAN of your account is DE02100100109307118603.

Happy banking!

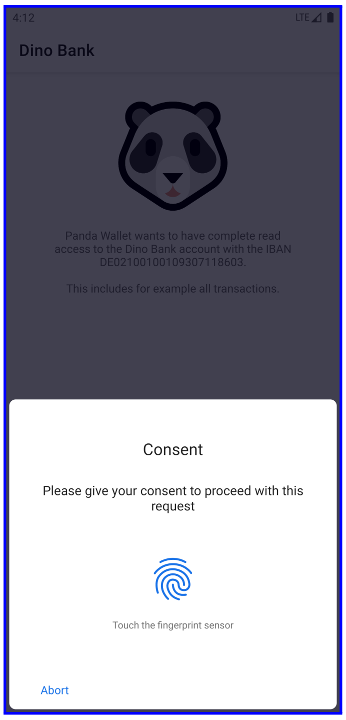
Figure 6.6: Logged In at Auth App



Panda Wallet wants to have complete read access to the Dino Bank account with the IBAN DE02100100109307118603.

This includes for example all transactions.

Figure 6.7: Account Access Request



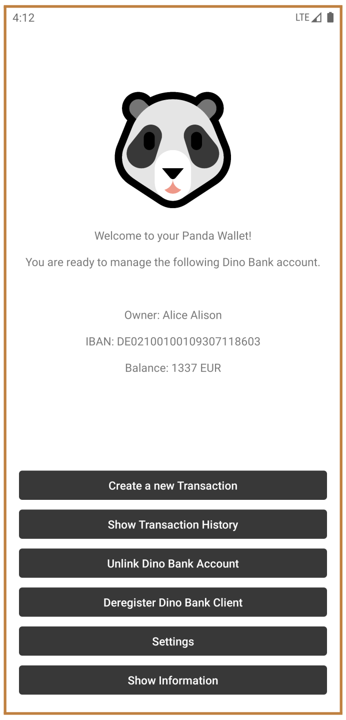
Panda Wallet wants to have complete read access to the Dino Bank account with the IBAN DE02100100109307118603.

This includes for example all transactions.

Consent

Please give your consent to proceed with this request

Figure 6.8: Account Access User Consent



Welcome to your Panda Wallet!

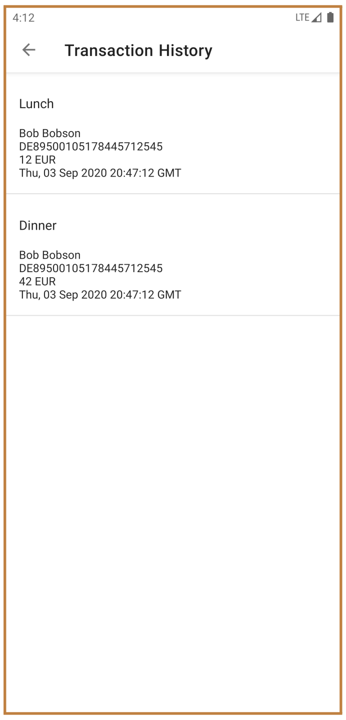
You are ready to manage the following Dino Bank account.

Owner: Alice Alison

IBAN: DE02100100109307118603

Balance: 1337 EUR

Figure 6.9: Linked Bank Account



Lunch

Bob Bobson
DE89500105178445712545
12 EUR
Thu, 03 Sep 2020 20:47:12 GMT

Dinner

Bob Bobson
DE89500105178445712545
42 EUR
Thu, 03 Sep 2020 20:47:12 GMT

Figure 6.10: Transaction History at Client

Listing 6.1 Account Authorization Details

```
{
  type: 'account',
  actions: [ 'read_account', 'read_transactions' ],
  locations: [
    'https://rs.dinobank.milesstoetzner.de/accounts',
    'https://rs.dinobank.milesstoetzner.de/transactions'
  ]
}
```

Listing 6.2 Transaction Authorization Details

```
{
  type: 'transaction',
  actions: [ 'create' ],
  locations: [ 'https://rs.dinobank.milesstoetzner.de/transactions' ],
  creditor_iban: 'DE02100100109307118603',
  creditor_name: 'Alice Alison',
  amount: 123.5,
  currency: 'EUR',
  recipient_iban: 'DE89500105178445712545',
  recipient_name: 'Bob Bobson',
  usage: 'Lunch last time'
}
```

6.2.4 Transaction Write Access

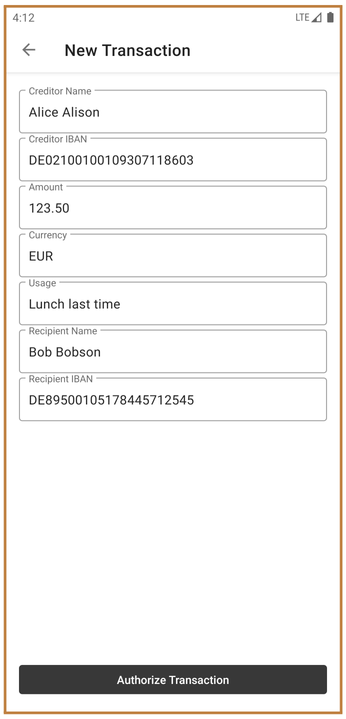
The user creates a new financial transaction at the client (see Figure 6.11). The "transaction" authorization details are used (see Listing 6.2) for the authorization request. They specify transaction details which include the creditor account, receiver account, and the amount of money.

The client redirects the user to the auth app which displays the request information (see Figure 6.12). If the user has not recently used the auth app then he is prompted to locally authenticate. The user authorizes the request using his fingerprint (see Figure 6.13). After that, the user is redirected back to the client which can now transfer the money (see Figure 6.14).

6.2.5 Intercepted Authorization Response

The attack requires that the client is using the "query.jwt" response mode and "code" response type and that the referrer header is not overwritten (see Figure 6.15).

The user wants to authorize the client access to his account details. The client uses a response mode that redirects the user using an implicit intent. After giving consent, the user is prompted with an app selection dialog (see Figure 6.16). The user chooses the malicious Shark Bank and gets redirected to the attacker. The attacker is now in possession of the authorization response (see Figure 6.17). The authorization response is in our case a signed and encrypted JARM. The authorization code is


4:12

New Transaction

Creditor Name: Alice Alison

Creditor IBAN: DE02100100109307118603

Amount: 123.50

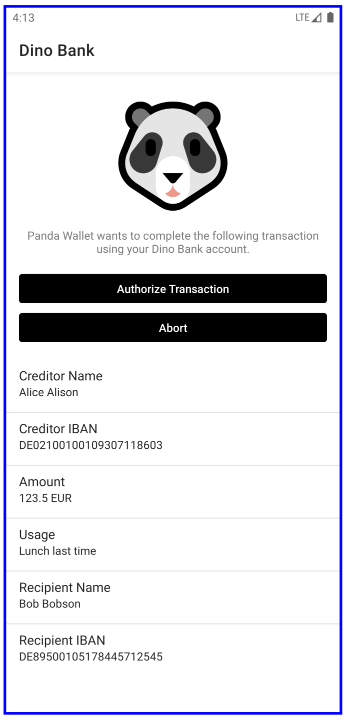
Currency: EUR

Usage: Lunch last time

Recipient Name: Bob Bobson

Recipient IBAN: DE89500105178445712545

Figure 6.11: Create Transaction


4:13

Dino Bank

Panda Wallet wants to complete the following transaction using your Dino Bank account.

Creditor Name
Alice Alison

Creditor IBAN
DE02100100109307118603

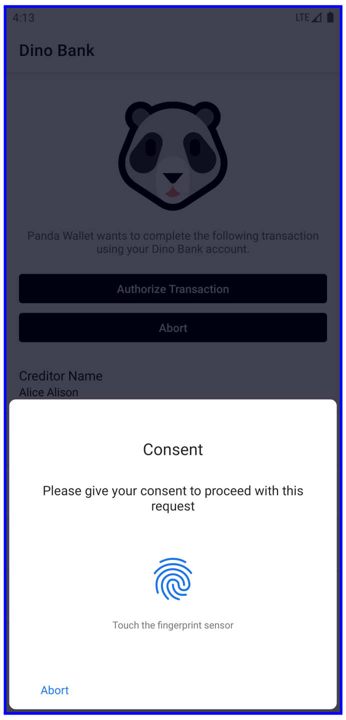
Amount
123.5 EUR

Usage
Lunch last time

Recipient Name
Bob Bobson

Recipient IBAN
DE89500105178445712545

Figure 6.12: Transaction Request


4:13

Dino Bank

Panda Wallet wants to complete the following transaction using your Dino Bank account.

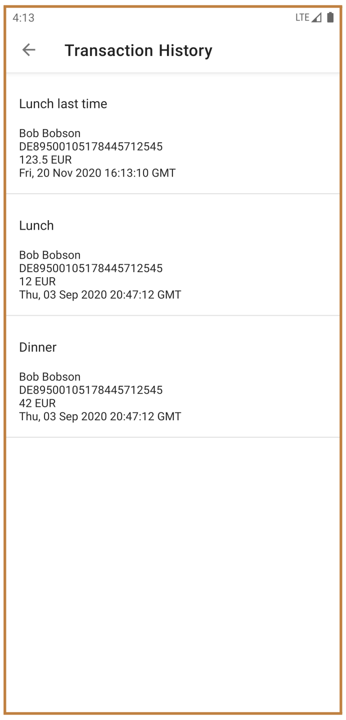
Creditor Name
Alice Alison

Consent

Please give your consent to proceed with this request

Touch the fingerprint sensor

Figure 6.13: Transaction User Consent


4:13

Transaction History

Lunch last time

Bob Bobson
DE89500105178445712545
123.5 EUR
Fri, 20 Nov 2020 16:13:10 GMT

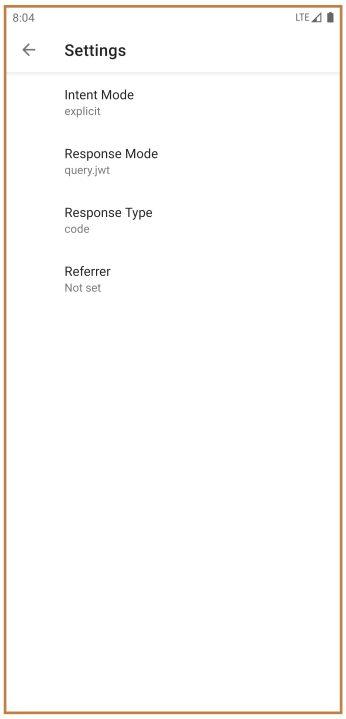
Lunch

Bob Bobson
DE89500105178445712545
12 EUR
Thu, 03 Sep 2020 20:47:12 GMT

Dinner

Bob Bobson
DE89500105178445712545
42 EUR
Thu, 03 Sep 2020 20:47:12 GMT

Figure 6.14: Updated Transactions History


8:04

Settings

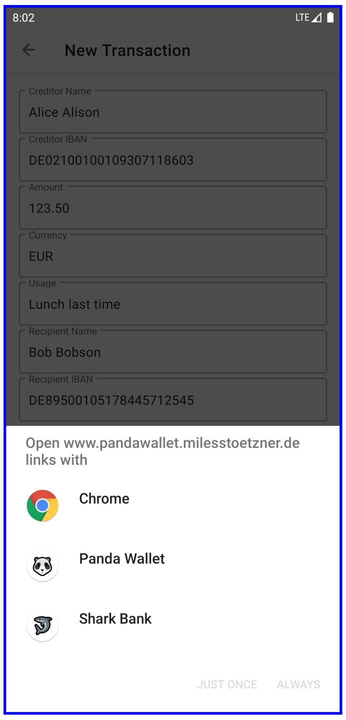
Intent Mode
explicit

Response Mode
query.jwt

Response Type
code

Referrer
Not set

Figure 6.15: Attack Settings


8:02

New Transaction

Creditor Name
Alice Alison

Creditor IBAN
DE02100100109307118603

Amount
123.50

Currency
EUR

Usage
Lunch last time

Recipient Name
Bob Bobson

Recipient IBAN
DE89500105178445712545

Open www.pandawallet.milesstoetzner.de links with

Chrome

Panda Wallet

Shark Bank

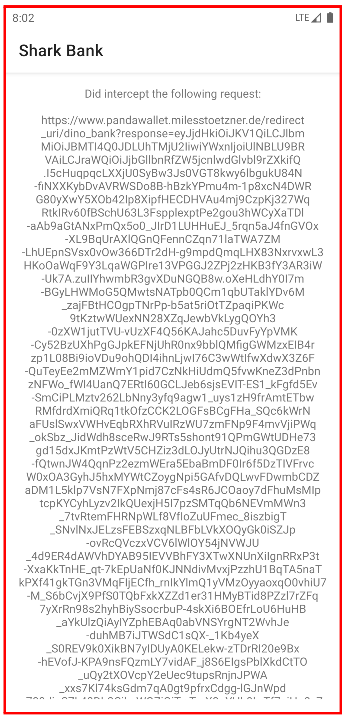
JUST ONCE ALWAYS

Figure 6.16: App Selection Dialog

not required to be encrypted, therefore, the attacker gets in general access to a valid authorization code. The attacker could even try to hide the interception attack by redirecting the user to the client.

6.2.6 App2Web

To redirect the user to the web the client must use an implicit intent. The implicit intent is resolved and verified to be a trusted browser. The client tries to start a custom tab and falls back to the browser (see Figure 6.18). The website is non-functional and can only be used to redirect the user back to the redirection endpoint using either an implicit intent or an explicit intent. Depending on the scenario both can be intercepted (see subsubsection 7.3.6.3).



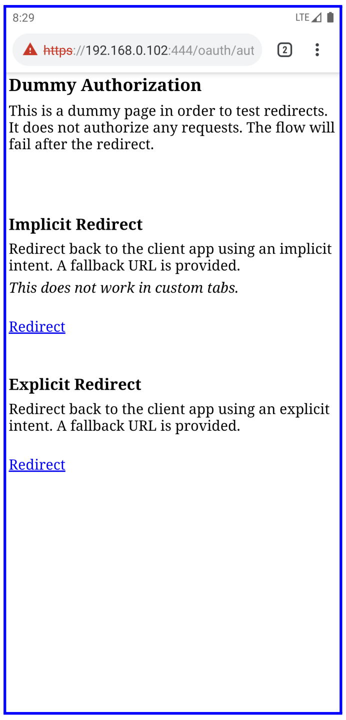
8:02

Shark Bank

Did intercept the following request:

```
https://www.pandawallet.milesstoetzner.de/redirect
_uri/dino_bank?response=eyJjdHkiOiJKV1QiLCJlbm
MiOiJBMTI4Q0JDLUhTMjU2IiwiYWxnIjoiUlNBLU9BR
VAiLCJraWQiOiJjbGllbnRfZW5jcnlwdGlvbl9rZXkifQ
.I5cHuqpqcLXXjU0SyBw3Js0VGT8kwy6lbgukU84N
-fiNXXKybDvAVRWSDo8B-hBzkYPmu4m-1p8xcN4DWR
G80yXwY5XOb42lp8XipfHECDHVAu4mj9CzpKj327Wq
RtkIRv60fBSchU63L3FspplexptPe2gou3hWCyXaTDl
-aAb9aGtANxPmQx5o0_JIrD1LUHHuEJ_5rqn5aJ4fnGVOx
-XL9BqUrAXlQGnQFennCZqn71laTWA7ZM
-LhUEpnSVsx0vOw366DTr2dH-g9mpdQmqLHX83NxrvxwL3
HKoOaWqF9Y3LqaWGPIre13VPGGJ2ZPj2zHKB3fY3AR3iW
-Uk7A.zuIIYhwmbR3gvXDuNGQB8w.oXeHLdhY0I7m
-BGyLHWMoG5QMwtsNATpb0QCm1qbUTaklYDv6M
_zajFBtHCOgpTNrPp-b5at5riOtTZpaqiPKWc
9tKztwWUexNN28XZqJewbVkLygQOYh3
-0zXW1jutTVU-vUzXF4Q56KAJahc5DuvFyYpVMK
-Cy52BzUXhPgGJpkEFNjUhR0nx9bblQMfigGWMzxEIB4r
zp1L08Bi9ioVDu9ohQDI4ihnLjwI76C3wWtIfwXdwX3Z6F
-QuTeyEe2mMZWmY1pid7CzNkHiUdmQ5fvwKneZ3dPnbn
zNFWo_fWl4UanQ7ERtI60GCLJeb6sjsEVIT-ES1_kFgfd5Ev
-SmCiPLMztv262LbNny3yfq9agw1_uys1zH9frAmtETbw
RMfdrdXmiQRq1tkOfzCCK2LOGFsBCgFHa_SQc6kWrN
aFUslSwxVWHvEqbRXhRVuIRzWU7zmFNp9F4mvVjiPWq
_okSbz_JidWdh8sceRwJ9RTs5shont91QPmGWtUDHe73
gd15dxJKmtPzWtV5CHZiz3dLOJyUtrNJQihu3QGDzE8
-fQtwnJW4QqnPz2ezmWEra5EbaBmDF0Ir6f5DzTIVFrvc
W0xOA3GyhJ5hxMYWtCZoygNpi5GAfvDQLwvFDwmbCDZ
aDM1L5klp7VsN7FXpNmj87cFs4sR6JCOaoy7dFhuMsMIp
tcpKYCyhLyzv2IkQUexjH5I7pzSMTqQb6NEVmMWn3
_7tvRtemFHRNpWLf8VfIoZuUFmec_8iszbigT
_SNvlNxJELzsFEBSzxqNLBFbLVkXOQyGk0iSZJp
-ovRcQVczxVCV6lWlOY54jNVWJU
_4d9ER4dAWVhDYAB95IEVVBhFY3XTwXNUnXiIgnRRxP3t
-XxaKkTnHE_qt-7kEpUaNf0KJNNdivMvxjPzzhU1BqTA5naT
kPXf41gkTGn3VMqFIjECfh_rnIkYlmQ1yVMzOyyaoxqO0vhiU7
-M_S6bCvjX9PfS0TQbFxkXZZd1er31HMyBTid8PZzl7rZFq
7yXrRn98s2hyhBiySsocrbuP-4skXi6BOEfrLoU6HuHB
_aYkUlzQiAyIYZphEBAq0abVNSYrgNT2WvhJe
-duhMB7iJTWSdC1sQX-_1Kb4yeX
_S0REV9k0XikBN7yIDUyA0KELekw-zTDrRI20e9Bx
-hEVofJ-KPA9nsFQzmLY7vidAF_j8S6EIgsPblXkdCtTO
_uQy2tXOVcpY2eUec9tupsRnjnJPWA
_xxs7Kl74ksGdm7qA0gt9pfrxCdgg-lGJnWpd
```

Figure 6.17: Intercepted JARM



8:29

~~https~~://192.168.0.102:444/oauth/aut

Dummy Authorization

This is a dummy page in order to test redirects. It does not authorize any requests. The flow will fail after the redirect.

Implicit Redirect

Redirect back to the client app using an implicit intent. A fallback URL is provided.

This does not work in custom tabs.

Redirect

Explicit Redirect

Redirect back to the client app using an explicit intent. A fallback URL is provided.

Redirect

Figure 6.18: App2Web

7 Security Discussion

In this section, we analyse the security of our proposal with respect to our attacker model. Our attacker model is based on the attacker model defined by FAPI 2.0 and extended by multiple assumptions and attacker capabilities. The additional assumptions include e.g. that unlocking of the bootloader requires a factory data reset and that biometric authentication can be only passed by someone whose biometrics are registered. The additional attacker capabilities include e.g. a misconfigured auth app, a misconfigured request introspection endpoint, and a misconfigured request authorization endpoint. The motivation for these attacker capabilities is social engineering attacks.

We discuss several ways to redirect the user between applications. Our main goal is to securely redirect the user between the client and the auth app using today's technologies. We require integrity, confidentiality, source authentication, and target authentication when redirecting the user. Roughly speaking, this means that the user is redirected from the correct source app to the correct target app and that no attacker is able to read or write the sent data. The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback. Authentication is based on comparing package signing certificates. The secured redirect can not only be applied to OAuth 2.0 but can also be used to further secure other scenarios.

We discuss the security of our proposal with respect to our attacker model and list identified vulnerabilities. For example, a physical attacker can access the resources to which a client has access to.

7.1 Security Goals

In this section, we define the security goals under which the profiles claim to be secure with respect to the attacker model as defined below. The goals are taken from *FAPI 2.0 Attacker Model* [Fetb].

7.1.1 Authorization

Authorization means that "no attacker can access resources belonging to a user" [Fetb]. For example, if Alice grants a digital wallet app access to her bank information then no attacker can access her banking information.

Resources can be accessed at the resource endpoint when providing a valid access token. The goal "is fulfilled if no attacker can obtain and use the access token of an honest user" [Fetb]. This means e.g. that no attacker can either use the access token on his own or trick the client into using the access token on his instigation.

We require that the user, the client, the authorization server protecting the user's identity and resources, the resource server, and the auth app belonging to the authorization server are honest. We discuss these requirements in subsection 7.4.1.

7.1.2 Authentication

Authentication means that "no attacker is able to log in at a client under the identity of a user" [Fetb]. For example, no attacker can log in to Alice's account at a digital wallet app that is linked with her bank account.

The identity is contained in the id token that is returned during an OIDC flow. The goal "is fulfilled if no attacker can obtain and use an id token carrying the identity of a user for login." [Fetb]

In this discussion, we use OIDC authorization code flow and the id token returned at the token endpoint for user authentication. We have the same requirements concerning honest parties as for authorization.

7.1.3 Session Integrity

Session Integrity [Fetb] consists of a subgoal for authentication and one for authorization. For example, an attacker perform san CSRF attacks or session swapping attacks and injects an authorization code belonging to the attacker into the flow of an honest user. This would result in the user using the attackers' resources or identity. The goal is fulfilled if both subgoals are fulfilled.

Session integrity for Authentication means that "no attacker is able to force a user to be logged in under the identity of the attacker" [Fetb]. For example, if Alice logs in to her account at a digital wallet app then she must not be logged into the account of any attacker. Alice might link the digital wallet account with her bank account and grant therefore the attacker access to her banking information.

Session integrity for Authorization means that "no attacker is able to force a user to use resources of the attacker" [Fetb]. For example, Alice must not use the attacker's bank account.

We have the same requirements concerning honest parties as for authorization.

7.2 Attacker Model

In this section, we define our attacker model which is based on the *FAPI 2.0 Attacker Model* [Fetb]. Our assumptions include among others application-related assumptions, e.g. that no attacker can bypass biometric authentication. Our attacker capabilities include among others mobile-specific attacker capabilities, e.g. limited physical access or a malicious application.

Common mitigations are the use of biometrics, TLS, mTLS, hardware-backed keys, sender constrained tokens, secured IPC, device integrity, and the verification of the used endpoints at the respective server.

7.2.1 Assumptions

In the following, we explain our assumptions which are based on the *FAPI 2.0 Attacker Model* [Fetb]. These include e.g. that no attacker can issue signatures without the knowledge of the corresponding private key and that there are no security-related bugs in Android or the device.

T1 **Cryptography**: Cryptography works as intended. An attacker can e.g. not generate valid signatures without knowing the private key.

T2 **Attacker Complexity**: An attacker has unlimited resources but can not break any of our assumptions, e.g. **T1**.

T3 **TLS**: TLS works as intended which includes e.g data integrity and confidentiality. This means e.g. that for a not compromised endpoint the correct certificate is used and that no attacker has access to the corresponding private key [Fetb].

T4 **JWKS**: “Where applicable, key distribution mechanisms work as intended, i.e., encryption and signature verification keys of uncompromised parties are retrieved from the correct endpoints.” [Fetb]

T5 **Browsers**: Browsers work as intended. This means, that security mechanisms work as intended and that there are e.g. no bug exploits [Fetb].

T6 **Endpoints**: Endpoints work as intended if they are not controlled by an attacker. This means, that the endpoints implement the specified logic and that there are e.g. no bug exploits [Fetb].

T7 **Android**: Android 10 is used as OS on the device. Furthermore, we assume that Android is honest and works as intended. This means that Android security mechanisms work as intended and that there are e.g. no bug exploits. This especially includes the SafetyNet and key attestation mechanisms. For more information about the Android version see subsection 7.6.4.

T8 **Device**: The device is compliant with CDD [Andm]. This means that device security mechanisms work as intended and that there are e.g. no bug exploits. This especially includes that verified boot and hardware-backed key attestation is supported and that the installed operating system passes CTS.

T9 **Manufacturer**: The device manufacturer is honest. This means e.g. that the manufacturer did not implement any kind of backdoors into the device in order to break our goals.

T10 **Biometrics**: No attacker can pass biometric authentication if his biometrics are not registered. We discuss this assumption in subsection 7.6.2.

T11 **Locked Bootloader**: The device is shipped with a locked bootloader. Otherwise, an attacker can flash a malicious operating system without a factory data reset. A locked bootloader is not required according to [Andm] but is applied on most Android devices [Andac].

T12 **Unlocking Bootloader**: Unlocking the bootloader requires a factory data reset. We require this assumption to protect user data stored on a device that is e.g. in possession of a physical attacker.

There are inconsistent sources about this requirement. According to [Andac] the factory data reset is not required, but according to [Ands] the reset is required.

T13 **Rooting**: The device can only be rooted if the bootloader is unlocked or if there is a bug exploit.

T14 **Execution Hooking**: Technologies that are able to hook into the execution flow of an application require root access. We assume that these technologies require to tamper with the Android security mechanisms.

T15 **Credentials**: An attacker does have access to all credentials like passwords or PINs, e.g. by phishing. Furthermore, we assume that the attacker is not able to log in at the authorization server under the identity of the user using phished credentials.

T16 **Device User**: The device is only used by the resource owner. This means especially that only the user's biometrics are registered.

T17 **Auth App Cardinality**: An auth app might have several authorization servers and an authorization server might have several auth apps. Note, that an honest auth app might have several honest and malicious authorization servers and vice versa.

7.2.2 Attacker Capabilities

In this section, we describe the attacker capabilities which are based on the *FAPI 2.0 Attacker Model* [Fetb]. An attacker might hold several of the described capabilities.

The capabilities can be grouped into three sections. The first section contains capabilities as defined by *FAPI 2.0 Attacker Model* [Fetb]. We extend these by the assumptions that the auth app might be misconfigured at the client and that the request introspection and authorization endpoint might be misconfigured. The second section contains the capabilities of a limited physical attacker, e.g. a stolen device. A physical attacker is limited since he is e.g. not able to compromise the device. The last section contains capabilities to run code on the device, e.g. a malicious application or a rooted device.

A1 **Web Attacker**: The *Web Attacker* (see A1 [Fetb] and T15 [MSBK19]) is a standard web attacker who can send messages to any available endpoint and receive messages that have been sent to endpoints under his control. He can send links to the user to start an authorization flow, e.g. the user clicks on a link on an email.

This attacker is usually mitigated by using TLS.

A2 **Authorization Server Attacker**: The *Authorization Server Attacker* (see A1a [Fetb]) is a web attacker who acts as an authorization server. This attacker might e.g. try to inject a leaked access token during a flow.

This attacker is usually mitigated by signed id tokens and by informing the authorization server or resource server about the used endpoints.

A3 **Network Attacker**: The *Network Attacker* (see A2 [Fetb] and T4, T5, T6 [MSBK19]) can control the network, which includes e.g. HTTP, Bluetooth, and NFC traffic. He can intercept, block, tamper, or resend messages.

This attacker is usually mitigated by using TLS and Android security mechanisms.

A4 **Read Authorization Request Attacker**: The *Read Authorization Request Attacker* (see A3a [Fetb]) is a web attacker who additionally can read the authorization request. The authorization request in our case corresponds to the request URI. Reading the authorization request might be achieved by intercepting an implicit intent but also due to browser history access or even due to anti-virus software that intercepts TLS connections.

We explicitly define this attacker since the user is redirected through a complex stack that might contain the browser, a native application, and the OS. Another reason might be a misconfigured request introspection endpoint.

This attacker is usually mitigated by PKCE.

A5 **Read Authorization Response Attacker**: The *Read Authorization Response Attacker* (see A3b [Fetb]) is a web attacker who additionally can read the authorization response. The authorization response in our case corresponds to the authorization code. Reading the authorization response might be achieved by intercepting an implicit intent but also due to browser history access.

We explicitly define this attacker since the user is redirected through a complex stack that might contain the browser, a native application, and the OS.

This attacker is usually mitigated by issuing sender-constrained tokens, by PKCE and by client authentication.

A6 **Read-Write Token Attacker**: The *Read-Write Token Attacker* (see A5 [Fetb]) is a web attacker that controls the token endpoint that the client uses, e.g. a misconfigured token endpoint.

This attacker is usually mitigated by sender-constrained tokens and by informing the resource server about the used token endpoint.

A7 **Read Resource Attacker**: The *Read Resource Attacker* (see A7 [Fetb]) is a web attacker who additionally can read resource requests and responses, e.g. when the attacker has access to proxy logs.

We do not consider this attacker when discussing the goals. Mitigations against this attacker depend on business logic. A mitigation could be application-level encryption.

A8 **Read-Write Resource Attacker**: The *Read-Write Resource Attacker* (see A8 [Fetb]) is a read resource attacker who additionally can tamper with the resource responses, e.g. a misconfigured resource endpoint.

We do not consider this attacker when discussing the goals. Mitigations against this attacker depend on business logic. A mitigation could be application-level signatures and encryption and replay protection.

A9 **Request Introspection Attacker**: The *Request Introspection Attacker* is a web attacker that controls the request introspection endpoint that the auth app uses, e.g. a misconfigured request introspection endpoint. This attacker has access to the request URI. The motivation for this attacker is social engineering attacks on developers.

This attacker is usually mitigated by the user consent which is the signature over the request introspection response.

Note, the advanced profile might use a signed and encrypted JWT as request. Therefore, this attacker does have access to the request URI.

A10 **Request Authorization Attacker**: The *Request Authorization Attacker* is a web attacker that controls the request authorization endpoint that the auth app uses, e.g. a misconfigured request authorization endpoint. This attacker has access to the user consent. The motivation for this attacker is social engineering attacks on developers.

This attacker is usually mitigated by the user consent, which is sender-constrained and bound to a request, and by informing the authorization server about the used endpoint.

Note, the advanced profile might use a signed and encrypted JWT as request. Therefore, this attacker does have access to the user consent.

A11 **Limited Physical Attacker**: The *Limited Physical Attacker* has physical access to the device, e.g. when the device is stolen or during border control. The device might be powered off, locked, or unlocked.

The attacker is limited concerning his capabilities to compromise the device and Android. This means that **T7** and **T8** are given even in the presence of this attacker. For example, with the knowledge of the device credentials, this attacker can unlock the device or the bootloader. On the other hand, this attacker is e.g. not able to tamper with the application sandbox of the running system or tamper with the execution flow of an application.

This attacker is usually mitigated by device and Android security mechanisms, e.g. biometrics.

A12 **Malicious App Attacker**: The *Malicious App Attacker* (see T7, T9, T11, T12, T13, and T15 [MSBK19]) is a malicious native application installed on the user's device. The user might have e.g. installed the wrong application.

This attacker can e.g. send and receive intents, register deep links, and trick the user into choosing the malicious application during an app selection dialog. He might be a malicious client or a malicious auth app.

Note, that the malicious application is signed by different keys than an honest client or auth app.

This attacker is usually mitigated by MASVS compliance and by verifying package signatures.

A13 **Malicious Auth App Attacker**: The *Malicious Auth App Attacker* is a malicious app attacker. The client uses the malicious app as auth app during a flow, e.g. a misconfigured auth app. This includes that the client uses the package name and signing certificates of the malicious app for the secured redirect. The attacker has therefore full control over the auth app. The motivation for this attacker is social engineering attacks on developers.

This attacker is usually mitigated by informing the authorization server about the used auth app.

A14 **Rooted Device Attacker**: The *Rooted Device Attacker* tricked the user into using a rooted device. The attacker has full control over the rooted device and is not restricted by Android security mechanisms.

This attacker is usually mitigated by verifying the device integrity.

7.3 Secure Redirect

In the following, we discuss how a user can be securely redirected between two applications. We analyze various ways and concepts to redirect the user. To redirect a user means to start an activity in another application which will start in the foreground. We require integrity, confidentiality, source authentication, and target authentication when redirecting the user. Roughly speaking, this means that the user is redirected from the correct source app (source authentication) to the correct target app (target authentication) and that no attacker is able to read or write the sent data (integrity and confidentiality). The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback (see subsection 7.3.2). Authentication is based on comparing package signing certificates. We do not make use of permissions since every client would is required to have the expected permissions including malicious clients.

The secured redirect can not only be applied to OAuth 2.0 but can be used to further secure other scenarios. For example, the user is redirected to an image processing application to apply image filters. The resulting image is then uploaded to a social media platform. An attacker must not be able to exchange the resulting image which is then uploaded to the victim's profile.

In subsection 7.3.6, we briefly discuss the security of ways to redirect the user in web-based flows, e.g. app2web.

Note, we discuss application authentication in section 5.7 and intent sender/ receiver identification in subsubsection 2.3.2.5.

7.3.1 Start Activity with Implicit Intent

The user can be redirected by starting an activity using an implicit intent. The receiver of an implicit intent can not be identified since it is resolved by Android after the intent has been sent. The intent can be intercepted by an attacker who e.g. registers the same URL in the intent filter.

App links are meant to secure implicit links. If the application of the intended receiver is not installed then we have a scenario without app links and therefore the intent can be intercepted as discussed above. Therefore, app links are not suitable.

This concept does not fulfill our requirements since the redirect can be intercepted.

7.3.2 Start Activity with Explicit Intent

The user can be redirected by starting an activity using an explicit intent. An explicit intent is ensured by Android to be delivered to the intent package name which is set by the intent sender. The intent sender can therefore identify and authenticate the intent receiver.

The intent sender can register a result callback to receive a result. This means that the intent sender is not required to expose an activity. Only the intent receiver can use the result callback and return the user back to the sender. Since a result callback is registered the intent receiver can identify and authenticate the intent sender. Therefore, mutual authentication is possible.

Starting an activity using an explicit intent with a registered result callback enables mutual authentication and that no component needs to be exported by the intent sender to receive a result.

This concept is according to Chen et al. [CPC+14] a secure way to redirect users and fulfills our requirements.

7.3.3 Pending Intent

A sender could create a pending intent that the receiver can use to redirect the user back. A pending intent is a way of privilege delegation and bears the danger of privilege escalation (see subsubsection 2.3.2.4). We do no make use of pending intents due to the danger of privilege escalation.

7.3.4 Preceding Inter Process Communication

In general, any IPC could be used to redirect the user. For example, an intent can be used to bind to a service. The receiving component could start an activity in order to redirect the user. Another way is to use file-based IPC.

These approaches enable that verifications can be done by the sender and receiver before the user is redirected and that multiple messages could be exchanged. This enables e.g. to authenticate application instances using mTLS implemented using e.g. Messenger (see section 2.3).

7.3.5 Backend Requests

Instead of communicating directly between applications the sender could send a request to the backend of the receiver and make use of TLS. The backend pushes the request to the receiver which starts an activity.

This reminds of the concept that CIBA is using. Therefore this concept introduces the same problems as CIBA (see section 3.14)

7.3.6 Web Based Flows

In the following, we briefly discuss how and if the user can be securely redirected in an app2web and web2app flow. We do not discuss a web2web flow since the user is redirected inside the browser as usual. We propose features that should be implemented by Android, e.g. automated signing certificates verification, and mention not considered mechanisms that should be part of future analysis, e.g. a *postMessage* channel to custom tabs.

7.3.6.1 Browser Whitelisting

Browser whitelisting is the concept of authenticating the intent receiver as a trusted browser before redirecting the user. This trust is based on a list of allowed browsers. The list might contain e.g. ten of the most popular browsers or custom browsers required in an enterprise environment.

This concept is used by AppAuth. AppAuth additionally checks if the browser version is allowed to be used with custom tabs.

7.3.6.2 OAuth Manager

Shehab and Mohsen [SM14] proposes the concept of the *OAuth Manager*. Any client redirects the user to the oauth manager. The oauth manager is a by the client and authorization server trusted application that uses web views to support multiple authorization servers to share tokens. Such a trusted application could be implemented by Android. The concept can be improved by securing the redirect as described in subsection 7.3.2 or by using custom tabs to e.g. access the cookies of the browser. Furthermore, if Android integrates such an application as system application a custom scheme that is not allowed to be registered by third-party applications could be used, e.g. “oauth”.

7.3.6.3 App2Web

In the context of an app2web redirect flow we recommend using browser whitelisting while redirecting the user to the authorization endpoint. A web view must not be used since the client can gain access to credentials and cookies.

We can not recommend a secure way to redirect the user back to the app redirection endpoint. The intent receiver can not be authenticated from the web since the receiver’s package information are not available in the browser. An explicit intent is not reliable. The explicit intent can be intercepted by a malicious application that the user has installed and that has the same package name as the honest client. App links are not reliable as well. The client might not be installed and, therefore, the user might be redirected to a malicious application using a deep link.

Note, that if the client starts the flow and the user is redirected to the redirection endpoint using an explicit intent then the intent can not be intercepted since only one application at the same time can be installed on the device having the same package name. This mitigates only interception attacks if the client is installed. An attacker being in possession of a valid request URI can start a flow and intercept the response.

Instead of redirecting the user to the web, the user could be redirected to an oauth manager (see subsubsection 7.3.6.2). We are not aware of any production-ready implementation of an oauth manager.

PKCE has been designed to mitigate interception attacks. Additionally, the authorization code is sender-constrained. Therefore, it might be applicable to use explicit intents.

To fix these problems we encourage Android to implement at least one of the following intent features:

- *Signing Certificates Verification*: Specify signing certificates that are used by Android to authenticate the intent receiver.
- *Verified App Link Enforcement*: Enforce that the intent can only be resolved if a verified app link exists.

By using one of these features the correct intent receiver is ensured and the user can be securely redirected. These features could also be used for app2app, which would e.g. mitigate the risk of implementation bugs by application developers.

The following mechanisms are not considered during our discussion. Future analysis should research if these can be used to securely redirect the user:

- *Custom Tabs PostMessage* [Andr] enables to send a *postMessage* [Mozb] to the custom tab. There is no documentation about how to receive a *postMessage* from the custom tab.
- *Custom Tabs Connection Callback* [Andq] notifies the native application e.g. when a new page is loaded. This might possibly be used to read out the authorization response from the URL of the loaded page.
- *Custom Tabs Action Button* [Andq] provides a way to inject a pending intent in a button of the custom tab UI. This might possibly be used to redirect the user.

7.3.6.4 Web2App

In the context of web2app we face the same problems when redirecting the user to an app redirection endpoint as when redirecting the user to an app redirection endpoint during an app2web flow. Therefore, we can not recommend a secure way to redirect the user. One interesting approach is to use CIBA. The user enters his username in the browser and the authorization server uses the auth app as the authentication device.

To redirect the user to the redirection endpoint we recommend redirecting the user using an explicit intent and to use browser whitelisting. How the desired browser is identified depends on how the user is redirected to the auth app. For example, the desired browser can be passed as intent referrer. This information can not be trusted in general. When using browser whitelisting the user is redirected in the worst case to a wrong but trusted browser.

7.4 Discussion

In this section, we discuss if our proposal is secure with respect to our security goals and attacker model. Before we discuss the goals we first discuss our requirements concerning honest parties and discuss several building blocks. During our security discussion, we identified several vulnerabilities. For example, a physical attacker can access the same resources which a client has access to.

7.4.1 Party Setup

In general, we require that the user, client, auth app, authorization server, and resource server are honest. Note, that this does not mean that the client uses the honest authorization server or auth app. When talking about an honest user we assume that the identity and the resources of the honest user are protected by an honest authorization server and honest resource server. In the following, we explain these requirements. Note, that the source code of a native application can be reversed engineered which could expose a malicious party.

A malicious client can trivially break our goals. The user can not differentiate between the malicious and an honest client and might authorize the malicious client to access his resources. The malicious client can then forward the resources to the attacker. This would break the authorization goal. Similar attacks can be constructed for the remaining goals. Therefore, we require an honest client.

A malicious auth app belonging to an honest authorization server can e.g. display wrong authorization information which requests less access than actually requested. When the user gives consent, the malicious auth app signs the correct authorization information under the user signature key. This problem is also discussed in subsection 2.1.7. Therefore, we require an honest auth app.

A malicious authorization server that protects the identity and resources of an honest user can e.g. simply grant the attacker access to the resources of a user. Therefore, we require an honest authorization server. Note, that a malicious authorization server can e.g. not grant access to resources protected by an honest authorization server.

A malicious resource server that holds the resources of an honest user can e.g. simply send the resources to the attacker. Therefore, we require an honest resource server.

When talking about A misconfigured endpoint we mean that the client uses not the correct endpoint. For example, instead of the token endpoint of an honest authorization server, the token endpoint of a malicious authorization server is used, or the other way around. If the misconfigured endpoint requires client authentication and the native application has access to correct credentials, then the client will use these credentials for authentication. This is due to different interpretations of the OAuth 2.0 standard which does not clearly isolate multiple clients from each other that are e.g. part of the same native application. This does not apply to auth apps. Auth apps are clearly isolated from other auth apps. For example, they can not access credentials from other auth apps even if they are running in the same native application. This means that auth app authentication will fail if the auth app uses an endpoint under control of an honest authorization server for which the auth app is not registered.

When talking about a client that chooses the wrong authorization server we mean that the used authorization server is not protecting the resources that the client intends to access at a resource server. This means e.g. that the chosen authorization server is not able to issue access tokens for that resource server.

7.4.2 Vulnerabilities

In the following, we discuss identified vulnerabilities. For example, a physical attacker can access the same resources which a client has access to.

7.4.2.1 Device Credentials

An attacker is per assumption in possession of valid device credentials (see **T15**). This assumption is e.g. based on phishing attacks.

The lock screen is protected by device credentials and optional biometrics. Therefore the security is reduced to the device's credentials. The attacker can easily bypass the lock screen using the phished device credentials.

For the attack, we assume that there has been already a successful oauth flow and that the client has access to the resources of an honest user. The local authentication at the client is based on device credentials. The attacker can therefore pass the local authentication and access the resources using the client.

Concluding, a physical attacker can access the same resources which a client has access to.

We discuss in subsection 7.6.3 why we do not require biometrics for local authentication at the client. Note, that the attacker is not able to authorize requests at the auth app with the knowledge of device credentials. Authorizing a request requires biometric authentication.

7.4.2.2 Second Login

We assume that only the user's biometrics are registered and that the user is already logged in at the auth app. A physical attacker in possession of the device credentials can register his own biometrics. Existing hardware-backed keys are invalidated if a new biometric is registered. Therefore, the auth app does not have access to its state anymore. The attacker can e.g. not use the registered user signature key.

A problem occurs if the user logs in at the auth app a second time [Damb]. At this point, the newly generated keys are bound to the user's and the attacker's biometrics. Therefore, the physical attacker is able to pass the local authentication and can authorize requests. The auth app must warn the user that a new biometric has been registered.

7.4.2.3 Rooted Device

Device integrity can not be verified if there is no communication with the authorization server. A rooted device attacker can e.g. fake client and auth app registration by tampering with the source code. The attacker can e.g. access any resources that the user wants to upload to the resource server.

We do not consider this scenario in our discussion since there are no messages exchanged with the authorization server.

Note, that it is not possible to successfully send resource requests using a rooted device. An access token is bound to a hardware-backed certificate that can not be extracted. During registration, device integrity is ensured (see subsubsection 7.4.3.1) which means that the device is not rooted.

7.4.3 Building Blocks

In the following, we discuss several building blocks that we use during our goal discussion. For example, we discuss why device and app integrity is given, why no attacker is able to successfully tamper with the authorization response, and why no attacker can obtain and use a user consent. Especially we talk about device integrity and our assumption during the discussion that device integrity is given after registration (see subsubsection 7.4.3.1).

7.4.3.1 Device Integrity

Roughly speaking, *device integrity* means that the device and the system are untampered.

A client/ auth app is a native application that dynamically registers at the authorization server. During registration, the client/ auth app provides an oauth integrity attestation which contains a SafetyNet attestation. The SafetyNet attestation attests device integrity and pre-installation integrity at the time of registration. This means especially that the rooted device attacker can not be present and that the bootloader is locked. Note, that based on subsubsection 7.4.3.3 an attacker can not obtain and use a valid SafetyNet attestation.

Concluding, device integrity is given at the time of registration.

In our discussion we assume that device integrity does not change after registration. We base this assumption on the fact that our limited physical attacker can not compromise the system. We discuss this problem in subsection 7.6.1.

7.4.3.2 App Integrity

App Integrity consists of pre-installation, post-installation, and runtime integrity.

Pre-installation integrity is provided by the SafetyNet attestation during registration (see subsubsection 7.4.3.1). We assume that post-installation and runtime integrity, and therefore app integrity, hold as long as device integrity is given. We base this assumption on the fact that the Android and device security mechanisms protect the installed application and that our limited physical attacker can not compromise the system.

Concluding, app integrity is given as long as device integrity is given.

7.4.3.3 Attestation

In the following, we discuss why an attacker can not obtain and use a valid oauth integrity attestation, SafetyNet attestation, or key attestation generated by an honest client at an honest authorization server. The same applies to an honest auth app.

The client receives the integrity nonce from the source endpoint and sends the attestation to the target endpoint. Any valid attestation attests the used integrity nonce, source endpoint, target endpoint, and authorization server. The authorization server verifies each of these. Therefore, the client uses honest endpoints. This means that the attestation is not sent to an attacker.

Furthermore, the integrity nonce is bound to an hardware-backed certificate. Only the client that receives the integrity nonce can send the attestation to the target endpoint. Therefore, an attacker being in possession of a valid attestation can not send a valid attestation on his own.

A valid SafetyNet attestation can only be generated on an untampered device running an untampered Android system. Based on the device integrity due to the SafetyNet attestation, no attacker can tamper with the SafetyNet attestation mechanism. Furthermore, the SafetyNet attestation attests that the correct package has generated the attestation. Therefore, the client that received the integrity nonce, generated the SafetyNet attestation, and sent the SafetyNet attestation is the same client running on an untampered device running an untampered Android system.

The SafetyNet attestation and any additional integrity attestations are sent as part of an oauth integrity attestation. The oauth integrity attestation is signed by the client using a hardware-backed key and is sender-constrained based on the contained integrity nonce. Based on the contained SafetyNet attestation, the client is running on an untampered device and system.

During registration, the SafetyNet attestation attests key attestations as part of its request data. Based on the device integrity due to the SafetyNet attestation, no attacker can tamper with the key attestation mechanism. Note, the certificate used to bind the integrity nonce is not verified to be hardware-backed. The attestation challenge is set during generation of the certificate and can later not be changed. This enables that the authorization server can verify that the certificate has been generated exclusively for the authorization server.

Concluding, an attacker can not obtain and use a valid attestation generated by an honest client at an honest authorization server.

7.4.3.4 Device Binding

An application is *device-bound* if it is not possible to move the application and its state from one device to another device [Mueb]. For example, if an honest user is logged in at an auth app, then it should not be possible to move the auth app including the user signature key to the attacker's device who can then authorize requests. Device binding can be achieved by using hardware-backed keys [Mueb]. Hardware-backed keys can not be extracted and moved to another device.

Concluding, the client and the auth app are device-bound due to hardware-backed keys.

Note, that this does not mean that the client and the auth app must be running on the same device.

7.4.3.5 Hardware-Backed and Access-Restricted Keys

During client/ auth app registration each registered key requires a key attestation. A key attestation ensures that the key is hardware-backed and that the key can only be used by the client/ auth app.

The keys are bound to the biometrics that are registered at the time of key generation. This means that the keys are invalidated if a new biometric is registered. Since the keys are hardware-backed and can not be extracted the client is device-bound (see subsubsection 7.4.3.4). Note, that based on subsubsection 7.4.3.3 an attacker can not obtain and use a valid key attestation.

Concluding, registered keys are hardware-backed and access-restricted.

7.4.3.6 ID Token

In the following, we discuss why an attacker can not be in possession of a valid id token that is signed by an honest authorization server issued for an honest client [Dam18].

An honest client chooses a freshly generated nonce for every authentication flow. A valid id token is signed by the honest authorization server, addressed for the honest client, and holds the expected nonce. There is no attacker who has access to the corresponding private key. Such an id token is only generated at the token endpoint by the honest authorization server after the honest client authenticates himself.

Device and app integrity is given since the registration passed. This means that e.g. the client never sends the id token to the attacker and that e.g. no attacker exists that can tamper with the device in order to intercept the requests. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

Concluding, an attacker is not in possession of a valid id token that is signed by an honest authorization server.

7.4.3.7 User Consent

In the following, we discuss why an attacker can not obtain and use the user consent of an honest user at an honest authorization server.

User consent is a signature over the request introspection response using the user signature key. In comparison to all other private keys, the user signature key can only be used for one cryptographic operation after authenticating the device user using biometrics. No attacker is able to pass biometric authentication if his biometrics are not registered. Additionally, the user signature key is invalidated if a new biometric is registered (see subsubsection 7.4.3.5) and requires user confirmation.

Concluding, only the device user can give consent.

The attacker might be in possession of a leaked user consent. The user consent is sender-constrained. It is bound to a hardware-backed and access-restricted certificate. Only the auth app that generated the certificate can use the certificate. Note, the auth app can not be moved to another device (see subsubsection 7.4.3.4).

Concluding, only the auth app can send the user consent.

The attacker might try to trick the auth app into using a leaked user consent. The auth app receives the user consent from the Android Keystore. Based on our assumption that device integrity is given no attacker can tamper with this mechanism. Note, that the request authorization endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

Concluding, the auth app can not be tricked into using a leaked user consent.

7.4.3.8 Authorization Response Tampering

In the following, we discuss why an attacker can not successfully tamper with an authorization response that an honest client successfully uses during token exchange at the token endpoint of an honest authorization server. We assume that the client is honest but might have a misconfigured auth app or might use an honest auth app that might have a misconfigured request introspection/ authorization endpoint.

Note, we do not discuss in this section the case where the client is using a token endpoint under the control of the attacker. The potential of such attacks is covered during the discussion of the goals (see subsection 7.4.4).

We assume that an attacker can successfully tamper with an authorization response that an honest client successfully uses during token exchange at the token endpoint of an honest authorization server.

The client uses the authorization response that is returned to the app redirection endpoint by an intent result callback of an explicit intent sent to an app authorization endpoint. Explicit mitigation against mix-up attacks [FKS16] is performed, e.g. the client verifies that the result intent data matches the redirect URI which is distinct per issuer. Since a result callback is used only the intent receiver can return an answer. Therefore, the attacker needs to hijack the intent or have control over the expected auth app.

Hijacking the intent requires to break the secured redirect. The redirect is secured by authenticating the intent receiver as the expected auth app. Application authentication is based on verifying the signing certificates of an installed application. This information is retrieved from the package manager. An attacker is required to tamper with the package manager or the intent mechanism in order to break application authentication. Based on our assumption that device integrity is given no attacker can tamper with this mechanism. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

There are two cases where the attacker can have control over the auth app. Either the client is using a misconfigured auth app or the auth app is using a misconfigured request introspection endpoint or misconfigured request authorization endpoint. If the auth app is misconfigured at the client then the attacker has full control over the auth app and can return any response. Based on the mitigation presented in subsection 5.15.2 and required in the baseline profile the authorization response returned by a misconfigured auth app can not be successfully used: The client sends the used auth app along with the token request and the authorization server verifies that the correct auth app has been used. Therefore, this case is not possible.

If the request introspection endpoint is misconfigured and under the control of the attacker, the attacker can tamper with the request introspection response. As discussed in subsection 5.15.4 every response would lead to a user consent that is invalid for every flow. Therefore, this case is not possible.

If the request introspection endpoint is misconfigured and under the control of an honest authorization server, the request will fail since auth app authentication is required (see subsection 7.4.1). Therefore, this case is not possible.

If the request authorization endpoint is misconfigured and under the control of the attacker, the attacker receives the user consent of an honest user and can tamper with the response. The user consent might be based on a misconfigured request introspection endpoint. As discussed above the resulting user consent is invalid for every request. As discussed in subsubsection 7.4.3.7 an attacker can not make use of a leaked user consent. Based on the mitigation presented in subsection 5.15.3 and required in the baseline profile the attacker can not inject an authorization code and tamper with the authorization response: The auth app informs the client which informs the authorization server about the used request authorization endpoint. The authorizations server verifies that the correct request authorization endpoint has been used. Therefore, this case is not possible.

If the request introspection endpoint is misconfigured and under the control of an honest authorization server, the request will fail since auth app authentication is required (see subsection 7.4.1). Therefore, this case is not possible.

Since no case is possible we conclude that our assumption is wrong and that an attacker can not successfully tamper with an authorization response.

7.4.4 Goals

In the following, we discuss the goals with respect to our party setup and attacker model. We assume during the discussion that device integrity is given after a successful registration (see subsubsection 7.4.3.1).

7.4.4.1 Authorization

In this section, we discuss why authorization is given. Based on the vulnerability mentioned in subsubsection 7.4.2.1 we do not consider a physical attacker. We assume that authorization is not given. An attacker can successfully obtain and use an access token belonging to an honest user during a resource request.

The resource request is either sent directly by the attacker or by the client.

An access token is sender-constrained. It is bound to a client certificate. This binding is checked at the resource endpoint. The client certificate is hardware-backed and access-restricted and, therefore, can only be used by the client and not e.g. by another application installed on the same device. In addition, the client can not be moved to another device (see subsubsection 7.4.3.4). Therefore, the attacker can not send the access token on his own.

The client receives the access token from the token endpoint. There are two cases. Either the token endpoint is either under the control of the attacker or the token endpoint is under the control of the honest authorization server.

If the token endpoint is under the control of the attacker then the access token of the honest user is not returned from the token endpoint that issued the access token. Based on the mitigation presented in subsection 5.15.1 the resource server will reject the resource request which is a contradiction with our assumption.

If the token endpoint is under the control of the honest authorization server then the client uses the honest user's authorization code for code exchange. This means that the attacker either injects an authorization response containing the user's authorization code into the flow or the attacker gives consent on his own. As discussed in subsubsection 7.4.3.8 the first case is not possible, as discussed in subsubsection 7.4.3.7 the second case is not possible. Therefore, this case is not possible. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

Since no case is possible we conclude that our assumption does not hold and that authorization is given.

7.4.4.2 Authentication

In the following, we discuss why authentication is given. We assume that authentication is not given. An attacker can obtain and use an id token carrying the identity of a user for login.

The user is logged in for the identity that is contained in the id token that is returned from the token endpoint. There are two cases. Either the token endpoint is under the control of the honest authorization server or under the control of the attacker.

If the token endpoint is under the control of the honest authorization server, then the attacker needs to trick the client into using a leaked authorization code of the user or give consent on his own. As discussed in subsubsection 7.4.3.8 an attacker can not inject a leaked authorization code. As discussed in subsubsection 7.4.3.7 the attacker can not log in to the user's account on its own. Therefore, this case is not possible. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

If the token endpoint is under the control of the attacker, then the attacker needs to return a valid id token containing the identity of the user. There are two cases. Either the token endpoint is misconfigured or the client has chosen a wrong authorization server which is under the control of the attacker. If the token endpoint is misconfigured then the attacker must be in possession of a valid id token. As discussed in subsubsection 7.4.3.6 this is not possible. If the client has chosen a wrong authorization server under the control of the attacker then the attacker can log in to the user under any identity in the scope of the chosen authorization server. As discussed in subsection 7.4.1 the identity of the honest user is protected by an honest authorization server and not by an authorization server under control of the attacker. Therefore, this case is not possible.

Since no case is possible we conclude that our assumption does not hold and that authentication is given.

7.4.4.3 Session Integrity

In this section, we discuss why session integrity is given. Based on the lack of cookie integrity we do not consider network attackers (see subsection 7.6.6).

We assume that session integrity is not given. This means that either session integrity for authentication or session integrity for authorization is not given. In the following, we show by contradiction that both are given. This is a contradiction with our assumption, and therefore, session integrity is given.

Session Integrity for Authentication: In the following, we show that session integrity for authentication is given. We assume that the user is logged in under the identity of the attacker. This means that the client received a valid id token which holds the identity of the attacker from the token endpoint. There are two cases. Either the token endpoint is under the control of the attacker or under the control of the honest authorization server.

If the token endpoint is under the control of the attacker then the attacker must be in possession of a valid id token. There are two cases. Either the token endpoint is misconfigured or the user has chosen a malicious authorization server that is under the control of the attacker. If the token endpoint is misconfigured then the attacker must be in possession of a valid id token. As discussed in subsubsection 7.4.3.6 this is not possible. The client uses the authorization server that the user chooses. If the chosen authorization server is a malicious authorization server then the attacker can log in the user under any identity in the scope of the chosen authorization server. This case is out of scope of OAuth 2.0 since the client does not know the identity of the user to verify the returned identity. The client could verify this e.g. by checking that the returned id token holds an expected email that has been previously entered by the user. But this is not part of the specification. Therefore, this case is not possible resp. out of scope.

If the token endpoint is under the control of the honest authorization server then the client uses the attacker's authorization code for code exchange. This means that the attacker's authorization code is part of the authorization response. As discussed in subsubsection 7.4.3.8 this is not possible. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

Since no case is possible we conclude that our assumption does not hold and that session integrity for authentication is given.

Session Integrity for Authorization: In the following, we show that session integrity for authorization is given. We assume that the user uses the resources of the attacker. This means that the client uses the attacker's access token and that the client received the attacker's access token from the used token endpoint. There are two cases. Either the token endpoint is under the control of the attacker or the token endpoint is under the control of the honest authorization server.

If the token endpoint is under the control of the attacker then the attacker's access token is not returned from the token endpoint that issued the access token. Based on the mitigation presented in subsection 5.15.1 the resource server will reject the resource request which is a contradiction with our assumption.

If the token endpoint is under the control of the honest authorization server then the client uses the attacker's authorization code for code exchange. As discussed in subsubsection 7.4.3.8 this is not possible. Note, that the token endpoint requires an oauth integrity attestation which continuously attests device integrity. For more information see subsubsection 7.4.3.1.

Since no case is possible we conclude that our assumption does not hold and that session integrity for authorization is given.

7.5 Non-Repudiation

Non-repudiation proves that a party has performed some specific action like sending a message. The party can not deny the performed action. FAPI 2.0 defines the following non-repudiation requirements which are achieved by application-level signatures [Fetb]. These requirements must hold for the advanced profile.

N1 Authorization Request

N2 Authorization Response

N3 ID Token Contents

N4 Introspection Response

N5 Userinfo Response

N6 Resource Request

N7 Resource Response

We extend those requirements for user consent and for the communication between the auth app and the authorization server. We follow the given concept and use application-level signatures, which especially includes oauth integrity attestations and the user consent. Non-repudiation of an oauth integrity attestation and user consent is also provided by the baseline profile.

N8 User Consent

N9 Request Introspection Request

N10 Request Introspection Response

N11 Request Update Request

N12 Request Update Response

N13 Request Delete Request

N14 Request Delete Response

N15 Request Authorization Request

N16 Request Authorization Response

N17 Auth App Registration Request

N18 Auth App Registration Response

N19 Auth App Deregistration Request

N20 Auth App Deregistration Response

N21 Registration Request

7.6 Considerations

In the following, we talk about considerations regarding our design proposal and security discussion. We talk e.g. about our assumption of the given device integrity and why the auth app is not an OAuth 2.0 client. Additionally, we discuss experimental concepts: the *request code grant* and *auth app id endpoint* (see subsection 7.6.11 and subsection 7.6.12).

We do not mention common considerations or best practices as e.g. described by *OAuth 2.0 Security Best Current Practice* [LBLF20] or *JSON Web Token Best Current Practices* [SHJ20].

7.6.1 Continuous Device Integrity

During our discussion, we assume that device integrity is given after a successful registration (see subsubsection 7.4.3.1). We base this assumption on the fact that our limited physical attacker can not compromise the system. In reality, device integrity can change over time, e.g. because a physical attacker is not limited. To counteract this problem one approach is to continuously attest integrity.

The oauth integrity attestation is required for token and user consent exchange. Therefore, integrity is continuously attested during the flow. Based on the SafetyNet attestation, device integrity and pre-installation integrity is attested. In the future it might be possible to additionally attest post-installation and runtime integrity, e.g. if the application runs inside of a secure enclave to ensure correct execution.

The oauth integrity attestation can also be applied at the resource endpoint. The resource server could e.g. implement the integrity endpoint and require a valid attestation once a day. In our implementation, we require an oauth integrity attestation for each resource request.

Secure hardware should not be affected by a physical attacker. For example, when verified boot fails then the device can not boot [Ele14] [Andap], and when the root of trust changes then existing hardware-backed keys can not be used [Andf]. Additionally, device integrity states that the bootloader is locked. The device can not be rooted after registration without a factory data reset. Factory data reset is, based on our assumptions, required when unlocking the bootloader.

7.6.2 Biometrics Security

During our discussion, we assume that no attacker can pass biometric authentication if his biometrics are not registered. There are several attacks and issues in reality concerning biometric sensors.

According to Chen et al. biometrics are less secure than device credentials. This statement is based on the fact that complex passwords are hard to guess while e.g. face recognition might be fooled by a family member. That Android trusts device credentials more the biometrics is also noticeable considering the fact that the device must be unlocked every 72 hours at least once using device credentials even when a biometrics sensor is used that is classified with the strongest security level [Andad].

Common attacks on fingerprint sensors are based on 3D printed fingerprints [Eri20]. Such attacks can not be fully prevented since “no biometric system is foolproof” [CMCL]. FAPI 2.0 aims to be secure in high-risk environments, therefore, we can assume that attackers are able to print such a fingerprint.

There has been an attack on the fingerprint sensor of the Samsung Galaxy S10 [Sam19], one of the newest Samsung devices. The attack exploits screen protective films that are based on silicone. The fingerprint sensor recognizes the screen protection as a finger which enables that everybody can pass the fingerprint authentication.

7.6.3 Biometrics Requirement

In our proposal, we do not require biometrics for unlocking the hardware-backed keys. Biometrics are only required for the user signature key. There are two reasons for that. The first reason is that device credentials are seen by Android as more secure than biometrics (see subsection 7.6.2), the second is a system limitation. In the following, we describe the system limitation.

It is only possible to enforce biometrics when unlocking a hardware-backed key when the validity duration is not set. The unlocked key can therefore only be used for a single cryptographic operation. While this behaviour is intended for the user signature key it is not applicable for e.g. the client data key or the client certificate. That is the reason why the user is locally authenticated using device credentials when unlocking the client/ auth app and why the user is locally authenticated using biometrics when authorizing a request.

To circumvent this problem the single cryptographic operation could be used to decrypt a software-backed key that is used to manage the application state including e.g. the client certificate. This exposes the risk that the software-backed key can be extracted and used by an attacker to send resource requests.

In Android 11 it is possible to set an allowed authenticator, e.g. biometrics or device credentials, independently of the validity duration. This would mitigate this problem but does not solve the fact that biometrics are seen to be less secure than device credentials (see subsection 7.6.2). The allowed authenticator can be also set on the biometric prompt.

Android 10 displays the biometric prompt even when the application requests to locally authenticate the user only using device credentials. We assume that the reason behind this is that device credentials and biometrics can be used to unlock a key with a validity duration and biometrics to unlock a key that requires per-use authentication. Therefore, biometric authentication can always be used to unlock a key.

Our implementation uses *encrypted shared preferences* which are encrypted key-value pairs stored at the internal data [Andar] [Andt]. Encrypted shared preferences are part of an Android security library and are recommended as a best practice by Android. Android states that they provide security on a level that is appropriate for banking applications. he security library does not support biometrics and requires the use of device credentials if user authentication is required. Our implementation displays a warning that even though the biometric prompt indicates that biometrics can be used that only device credentials can be used.

7.6.4 Android Version

Our analysis is based on Android 10 which has been the newest Android version at the beginning of the analysis. At the given time Android 11 is the newest Android version.

We require that hardware-backed key attestation is available which is supported since Android 8 [HL]. According to [HL] we can assume that all mobile devices running Android 8 and higher support hardware-backed key attestation. Currently, 60.8% of all Android devices are running at least Android 8 while only 8.2% run Android 10 [1]. Without further analysis, we assume that our discussion also applies for Android 8 and newer.

7.6.5 Web Authentication

Our user consent concept reminds of the concept used by Web Authentication (WebAuthn) [W3C]. WebAuthn has been designed for multi-factor authentication based on a public key registered at a relying party while the corresponding private key is stored on an authenticator. The public key has been registered at a relying party along with an attestation that attests information about the keypair and the authenticator. During authentication, the user locally authenticates at the authenticator which issues a signature over a challenge issued using the private key. This signature provides a multi-factor authentication since the signature proves possession of the authenticator and a successful local authentication.

WebAuthn supports SafetyNet attestation or key attestation as attestation during registration. In comparison, we make use of both in order to verify device integrity, that the key is hardware-backed and access-restricted, and that the correct package has been installed. In general, we require a stricter verification. For example, we verify that the key has been generated by the correct application. In addition, we use a challenge that even binds the attestation to the endpoint from where the integrity nonce is received.

The use of WebAuthn is still interesting in terms of interoperability and support of extensions. The specification is not restricted to Android but supports all kinds of devices and operating systems. WebAuthn includes extensions that e.g. attest the current location of the device. Something similar can be included also in the oauth integrity attestation but must be specified first.

Note, that a malicious WebAuthn client can access any keys stored at the authenticator [Stö18]. This is extremely insecure if a roaming authenticator is used that also contains keys for other relying parties. A roaming authenticator is in comparison to an embedded authenticator not integrated into the device but is a separate device, e.g. a FIDO security key [Yub]. Future work should analyse the implications and security of WebAuthn as a replacement for user consent.

[1] These information are retrieved from Android Studio at 16.11.2020.

7.6.6 Lack of Cookie Integrity

A network attacker is able to set cookies from non-secure to secure origins [ZJL+15] and can, therefore, log in the user under his identity. There are new mechanisms to prevent this: The *secure* and *host* prefix [Moza]. If the cookie name starts with one of these prefixes, then the cookie can only be set on secure origins. Another mitigation is to not use cookies and send the session state or identifier as header. In our discussion, we do not assume that these mitigations are used.

7.6.7 Auth App as OAuth2.0 Client

The auth app could be designed as OAuth 2.0 client. This enables that the auth app can also use resource endpoints and provide more functionality than we require.

The *Resource Owner Password Grant* could be used where the user enters his credentials into the client [Har12]. This grant is deprecated since the client should not get access to the user's credentials [LBLF20]. Additionally, this grant does not support multi-factor authentication. In our specific scenario, the client would be a first party and not a third party. Therefore, this grant might be applicable.

The auth app could make use of the authorization code grant using an app2web flow. In subsection 7.3.6 we discuss the problem with respect to securely redirect the user in a web context.

Based on the above considerations we decided to not design the auth app as OAuth 2.0 client.

7.6.8 Background Processing

Access to hardware-backed keys is only granted after the user has been authenticated for a validity duration. This means that the client can not access resources e.g. at night in order to run expensive long-running tasks like a financial report. Depending on the scenario the client might use a secondary less secure communication channel to perform these tasks.

7.6.9 User Signature Oracle

The user consent is a signature over the request introspection response. There is the danger that the user consent might be a valid signature for another use case, e.g. in an extension of our concept. The user signature key must be only used for user consent and the user consent must only be used for user consent exchange.

7.6.10 Confidential Data in OAuth Integrity Attestation Request Data

The client data hash of the oauth integrity attestation might contain sensitive data, e.g. user credentials. This problem is mitigated by using a cryptographic hash function and by hashing the integrity nonce, a fresh cryptographic nonce. Note, if required and applicable, then the sensitive data can be simply removed from the request data.

7.6.11 Request Code Grant

Motivated by the question “Why do we need to pass the authorization code during the redirect?” we present the *Request Code Grant*. The request code is an authorization code that is already returned at the pushed authorization request endpoint but that is not valid until the request has been authorized. The authorization response serves only as a notification. Therefore, attacks that tamper with the authorization response in order to e.g. inject an authorization code are not possible. The concept is adopted from the CIBA ping mode. Our implementation supports the request code grant.

We define the following response modes. “intent.result.ping” is the default response mode:

- “intent.result.ping” uses a result callback and authenticates the sender before redirecting back the user. It does not return any data.
- “intent.ping” uses an explicit intent and authenticates the receiver before redirecting back the user. It does not return any data.
- “ping” uses an implicit intent. It does not return any data.

We do not make use of this grant. Future work should analyze the security implications of the request code grant.

7.6.12 Auth App ID Endpoint

The *auth app id endpoint* is an endpoint at the auth app which returns the auth app id. This endpoint is implemented by an AIDL service (see Listing 7.1). If the user is not logged in then *null* is returned. The endpoint is identified by the newly defined auth app metadata *apk authorization service*, the canonical name of the java class that implements the AIDL service, e.g. “de.milesstoetzner.dinobank.AuthorizationService”.

The endpoint might be used for multiple use cases which we did not further analyze. One use case might be to bind the flow to the auth app resp. device. Android does not provide a global handle for the device in order to protect the privacy of the user [Andl]. The idea is that the client fetches the auth app id of the auth app installed on the same device in order to bind the flow to the auth app resp. device.

Another use case might be to enable CIBA. The authorization server can identify the authentication device based on the auth app id.

We do not make use of this endpoint in our proposal since there are several problems. One problem is that the auth app id is part of the auth app data which are only available for a validity duration after local authentication of the user. Another problem is that if the native application contains multiple auth apps, then the correct auth app must be identified which might require user interaction.

Listing 7.1 Auth App ID Endpoint

```
package de.milesstoetzner.dinobank;

interface IAuthorizationService {
    /**
     * Get AuthAppID
     */
    @nullable String getAuthAppID();
}
```

8 Conclusion and Outlook

In this thesis we presented and analysed our proposal for an Android app2app redirect flow for FAPI 2.0. Our main goal is to securely redirect the user between the client and the auth app using today's technologies. The secure redirect is achieved by mutually authenticating the intent receiver and sender as well as by using a result callback. Authentication is based on comparing package signing certificates.

We described how the client and auth app must register at the authorization server and specified the communication between the auth app and the authorization server during a flow. Furthermore, we set requirements concerning hardware-backed keys, device integrity, and mobile security best practices.

Our proposal includes the oauth integrity attestation. This attestation contains a SafetyNet attestation and is used to continuously attest device integrity. The SafetyNet attestation is an attestation issued by Google attesting device integrity. The SafetyNet attestation also ensures that only the correct applications can register at the authorization server.

During a flow the client redirects the user to the auth app using an explicit intent. The client authenticates the intent receiver as the expected auth app and the auth app authenticates the intent sender as the expected client. When authorizing the request the user is required to pass a local authentication based on biometrics. The result of the passed authentication is a signature over the request information using a hardware-backed private key. This signature is called *user consent*. The corresponding public key has been registered at the authorization server during auth app registration. The auth app exchanges the user consent at the authorization server for an authorization code and redirects the user back to the client using an intent result callback.

During our security discussion, we identified some vulnerabilities. For example, a physical attacker with knowledge of the device credentials can access the same resources which a client has access to. The reason for this attack is that we do not require that access to the client is enforced by biometrics. This is due to Android Keystore not being able to enforce biometrics for keys if the key is not unlocked for each cryptographic operation.

During our security discussion, we assumed that the physical attacker is limited and that such an attacker can not compromise the system. To counteract this problem one approach is to continuously attest device integrity. This is done during each flow since an oauth integrity attestation is required for token and user consent exchange. Future work should analyse how a not limited physical attacker can be mitigated. Additionally, a formal analysis of our proposal should be performed.

During our redirect discussion, we also discussed web-based flows. Our secured redirect can not be applied to web-based flows since the used technologies are not available from the web. It is e.g. not possible from the web to authenticate the intent sender. We mention several technologies that might be used to achieve a secure redirect, e.g. it seems to be possible to send postMessages between an application and a custom tab. Future work should analyse these.

Our proposal defines an own concept for user consent that is similar to WebAuthn. Future work should analyse if our concept should be replaced by WebAuthn.

To mitigate misconfigurations we apply a concept of informing the server about the used endpoint. Future work should analyse if this approach can be applied for more endpoints. For example, FAPI 2.0 does not assume that the pushed request authorization endpoint is misconfigured.

Bibliography

[Anda] Android. *Activity*. URL: https://developer.android.com/guide/topics/manifest/activity-element.html (cit. on p. 27).

[Andb] Android. *Android Debug Bridge*. URL: https://developer.android.com/studio/command-line/adb (cit. on p. 29).

[Andc] Android. *Android Developer Best Practices*. URL: https://developer.android.com/topic/security/best-practices (cit. on p. 23).

[Andd] Android. *Android Developer Security Tips*. URL: https://developer.android.com/training/articles/security-tips (cit. on p. 27).

[Ande] Android. “Android Enterprise Security White Paper 2020”. In: () (cit. on p. 19).

[Andf] Android. *Android Keystore Features*. URL: https://source.android.com/security/keystore/features (cit. on pp. 22, 111).

[Andg] Android. *Android keystore system*. URL: https://developer.android.com/training/articles/keystore (cit. on pp. 21, 22).

[Andh] Android. *Android Runtime (ART) and Dalvik*. URL: https://source.android.com/devices/tech/dalvik (cit. on p. 19).

[Andi] Android. *Application Fundamentals*. URL: https://developer.android.com/guide/components/fundamentals (cit. on pp. 23, 24).

[Andj] Android. *Application Sandbox*. URL: https://source.android.com/security/app-sandbox (cit. on p. 25).

[Andk] Android. *Application Signing*. URL: https://source.android.com/security/apksigning (cit. on p. 24).

[Andl] Android. *Best practices for unique identifiers*. URL: https://developer.android.com/training/articles/user-data-ids.html (cit. on p. 115).

[Andm] Android. *Compatibility Definition Android 10*. URL: https://source.android.com/compatibility/10/android-10-cdd.pdf (cit. on pp. 20, 21, 93).

[Andn] Android. *Compatibility Test Suite*. URL: https://source.android.com/compatibility/cts (cit. on p. 20).

[Ando] Android. *Create Deep Links to App Content*. URL: https://developer.android.com/training/app-links/deep-linking (cit. on pp. 27, 28).

[Andp] Android. *Custom Tabs*. URL: https://developers.google.com/web/android/custom-tabs (cit. on p. 23).

[Andq] Android. *Custom Tabs Implementation guide*. URL: https://developers.google.com/web/android/custom-tabs/implementation-guide (cit. on p. 100).

[Andr] Android. *CustomTabsSession postMessage*. URL: https://developer.android.com/reference/androidx/browser/customtabs/CustomTabsSession#postMessage(java.lang.String,%20android.os.Bundle) (cit. on p. 100).

[Ands] Android. *Device State*. URL: https://source.android.com/security/verifiedboot/device-state (cit. on p. 93).

[Andt] Android. *EncryptedSharedPreferences*. URL: https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences (cit. on pp. 80, 81, 112).

[Andu] Android. *Feature Preview: SafetyNet Attestation API evaluationType*. URL: https://groups.google.com/g/safetynet-api-clients/c/lpDXBNeV7Fg (cit. on p. 31).

[Andv] Android. *File-Based Encryption*. URL: https://source.android.com/security/encryption/file-based (cit. on p. 21).

[Andw] Android. *fs-verity Integration*. URL: https://source.android.com/security/features/apk-verity (cit. on p. 25).

[Andx] Android. *Full-Disk Encryption*. URL: https://source.android.com/security/encryption/full-disk (cit. on p. 20).

[Andy] Android. *Getting a result from an activity*. URL: https://developer.android.com/training/basics/intents/result (cit. on p. 28).

[Andz] Android. *Hardware-backed Keystore*. URL: https://source.android.com/security/keystore (cit. on p. 22).

[Andaa] Android. *Intent*. URL: https://developer.android.com/reference/android/content/Intent (cit. on p. 26).

[Andab] Android. *Intents and Intent Filters*. URL: https://developer.android.com/guide/components/intents-filters (cit. on p. 27).

[Andac] Android. *Locking/Unlocking the Bootloader*. URL: https://source.android.com/devices/bootloader/locking_unlocking (cit. on pp. 20, 93).

[Andad] Android. *Measuring Biometric Unlock Security*. URL: https://source.android.com/security/biometric/measure (cit. on p. 111).

[Andae] Android. *PackageManager*. URL: https://developer.android.com/reference/android/content/pm/PackageManager (cit. on p. 23).

[Andaf] Android. *Permissions on Android*. URL: https://developer.android.com/guide/topics/permissions/overview (cit. on p. 25).

[Andag] Android. *Platform Architecture*. URL: https://developer.android.com/guide/platform (cit. on p. 19).

[Andah] Android. *Protected Confirmation*. URL: https://source.android.com/security/protected-confirmation (cit. on p. 22).

[Andai] Android. *Run embedded DEX code directly from APK*. URL: https://developer.android.com/topic/security/dex (cit. on p. 32).

[Andaj] Android. *SafetyNet Attestation*. URL: https://developer.android.com/training/safetynet/attestation (cit. on pp. 30, 31).

[Andak] Android. *Security and Privacy Enhancements in Android 10*. URL: https://source.android.com/security/enhancements/enhancements10 (cit. on p. 22).

[Andal] Android. *Security-Enhanced Linux in Android*. URL: https://source.android.com/security/selinux (cit. on p. 21).

[Andam] Android. *Show a biometric authentication dialog*. URL: https://developer.android.com/training/sign-in/biometric-auth (cit. on p. 22).

[Andan] Android. *Trusted Web Activity*. URL: https://developers.google.com/web/android/trusted-web-activity (cit. on p. 23).

[Andao] Android. *Verify Android App Links*. URL: https://developer.android.com/training/app-links/verify-site-associations (cit. on p. 28).

[Andap] Android. *Verifying Boot*. URL: https://source.android.com/security/verifiedboot/verified-boot (cit. on p. 111).

[Andaq] Android. *Verifying hardware-backed key pairs with Key Attestation*. URL: https://developer.android.com/training/articles/security-key-attestation (cit. on pp. 22, 68, 69).

[Andar] Android. *Work with data more securely*. URL: https://developer.android.com/topic/security/data (cit. on p. 112).

[App] Approov. *Stop API Security Threats at the Edge*. URL: https://www.approov.io (cit. on p. 56).

[Bec] A. Bechtsoudis. *Examining the value of SafetyNet Attestation as an Application Integrity Security Control*. CENSUS. URL: https://census-labs.com/news/2017/11/17/examining-the-value-of-safetynet-attestation-as-an-application-integrity-security-control (cit. on pp. 31, 32).

[CBSL20] B. Campbell, J. Bradley, N. Sakimura, T. Lodderstedt. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. Feb. 2020. DOI: 10.17487/RFC8705. URL: https://rfc-editor.org/rfc/rfc8705.txt (cit. on p. 44).

[CFGW11] E. Chin, A. P. Felt, K. Greenwood, D. Wagner. “Analyzing Inter-Application Communication in Android”. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’11. Bethesda, Maryland, USA: Association for Computing Machinery, 2011, pp. 239–252. ISBN: 9781450306430. DOI: 10.1145/1999995.2000018. URL: https://doi.org/10.1145/1999995.2000018 (cit. on p. 18).

[Chra] Chrome. *Android Intents with Chrome*. URL: https://developer.chrome.com/multidevice/android/intents (cit. on p. 28).

[Chrb] Chromium. *Android IPC Security Considerations*. URL: https://chromium.googlesource.com/chromium/src.git/+/master/docs/security/android-ipc.md (cit. on p. 62).

[CMCL] H. Chen, V. Mohan, K. Chyn, L. Louis. *Lockscreen and authentication improvements in Android 11*. URL: https://android-developers.googleblog.com/2020/09/lockscreen-and-authentication.html (cit. on pp. 21, 111, 112).

[CPC+14] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, P. Tague. "OAuth Demystified for Mobile Application Developers". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 892–903. ISBN: 9781450329576. DOI: 10.1145/2660267.2660323. URL: https://doi.org/10.1145/2660267.2660323 (cit. on pp. 18, 62, 98).

[Dama] I. Damier. *One Biometric API Over all Android*. URL: https://android-developers.googleblog.com/2019/10/one-biometric-api-over-all-android.html?m=1 (cit. on p. 21).

[Damb] I. Damier. *Using BiometricPrompt with CryptoObject: how and why*. URL: https://medium.com/androiddevelopers/using-biometricprompt-with-cryptoobject-how-and-why-aace500ccdb7 (cit. on p. 102).

[Dam18] S. Damabi. *Security Analysis of the OpenID Financial-grade API*. University of Stuttgart, 2018. URL: https://books.google.de/books?id=d2AkwAEACAAJ (cit. on pp. 51, 75, 105).

[DB17] W. Denniss, J. Bradley. *OAuth 2.0 for Native Apps*. RFC 8252. Oct. 2017. DOI: 10.17487/RFC8252. URL: https://rfc-editor.org/rfc/rfc8252.txt (cit. on pp. 18, 23, 44).

[Ele14] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. 1st. USA: No Starch Press, 2014. ISBN: 1593275811 (cit. on pp. 20, 24, 26, 29, 30, 111).

[Eri20] J. Erickson. *The Good and Bad of Biometrics*. Mar. 11, 2020. URL: https://duo.com/labs/research/the-good-and-bad-of-biometrics (cit. on p. 112).

[Fer] G. Fernandez. *OpenID Connect Client Initiated Backchannel Authentication Flow. openid-client-initiated-backchannel-authentication-core-03*. URL: https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html (cit. on p. 45).

[Feta] D. Fett. *FAPI 2.0 Advanced Profile*. OpenID Foundation. URL: https://bitbucket.org/openid/fapi/src/4dbc5c9596f1e07a683ec8797b84bc4e1060bd91/FAPI_2_0_Advanced_Profile.md (cit. on p. 49).

[Fetb] D. Fett. *FAPI 2.0 Attacker Model*. OpenID Foundation. URL: https://bitbucket.org/openid/fapi/src/c157699fd701283b40ce98b03fdf41d463f59421/FAPI_2_0_Attacker_Model.md (cit. on pp. 47, 91–95, 110).

[Fetc] D. Fett. *FAPI 2.0 Baseline Profile*. OpenID Foundation. URL: https://bitbucket.org/openid/fapi/src/4dbc5c9596f1e07a683ec8797b84bc4e1060bd91/FAPI_2_0_Baseline_Profile.md (cit. on pp. 47, 48, 58).

[FKS16] D. Fett, R. Küsters, G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *CoRR* abs/1601.01229 (2016). arXiv: 1601.01229. URL: http://arxiv.org/abs/1601.01229 (cit. on pp. 41, 48, 106).

[Fri] Frida. *FRIDA*. URL: https://frida.re/ (cit. on p. 30).

[Goo] Google. *Gigital Asset Links Getting Started*. URL: https://developers.google.com/digital-asset-links/v1/getting-started (cit. on p. 25).

[GTH18] S. Groß, A. Tiwari, C. Hammer. “PIAnalyzer: A Precise Approach for PendingIntent Vulnerability Analysis”. In: *Computer Security*. Ed. by J. Lopez, J. Zhou, M. Soriano. Cham: Springer International Publishing, 2018, pp. 41–59. ISBN: 978-3-319-98989-1 (cit. on p. 29).

[Hal] P. Hallam-Baker. *HTTP Integrity Header. draft-hallambaker-httpintegrity-02*. URL: https://tools.ietf.org/id/draft-hallambaker-httpintegrity-02.html (cit. on p. 57).

[Har12] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt (cit. on pp. 77, 114).

[HL] M. Hata, R. Lindemann. *FIDO Alliance White Paper: Hardware-backed KeystoreAuthenticators (HKA) on Android 8.0 orLater Mobile Devices*. URL: https://media.fidoalliance.org/wp-content/uploads/Hardware-backed_Keystore_White_Paper_June2018.pdf (cit. on p. 113).

[JBS15a] M. Jones, J. Bradley, N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://rfc-editor.org/rfc/rfc7515.txt (cit. on p. 39).

[JBS15b] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://rfc-editor.org/rfc/rfc7519.txt (cit. on p. 39).

[JH15] M. Jones, J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. May 2015. DOI: 10.17487/RFC7516. URL: https://rfc-editor.org/rfc/rfc7516.txt (cit. on p. 39).

[LBLF20] T. Lodderstedt, J. Bradley, A. Labunets, D. Fett. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-16. Work in Progress. Internet Engineering Task Force, Oct. 2020. 50 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-16 (cit. on pp. 44, 111, 114).

[LC] T. Lodderstedt, B. Campbell. *Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)*. OpenID Foundation. URL: https://openid.net/specs/openid-financial-api-jarm-ID1.html (cit. on p. 41).

[LCS+20] T. Lodderstedt, B. Campbell, N. Sakimura, D. Tonge, F. Skokan. *OAuth 2.0 Pushed Authorization Requests*. Internet-Draft draft-ietf-oauth-par-04. Work in Progress. Internet Engineering Task Force, Sept. 2020. 21 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-par-04 (cit. on p. 42).

[LDS13] T. Lodderstedt, S. Dronia, M. Scurtescu. *OAuth 2.0 Token Revocation*. RFC 7009. Aug. 2013. DOI: 10.17487/RFC7009. URL: https://rfc-editor.org/rfc/rfc7009.txt (cit. on p. 68).

[LRC19] T. Lodderstedt, J. Richer, B. Campbell. *OAuth 2.0 Rich Authorization Requests*. Internet-Draft draft-lodderstedt-oauth-rar-03. Work in Progress. Internet Engineering Task Force, Nov. 2019. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-lodderstedt-oauth-rar-03 (cit. on pp. 42, 43).

[Moza] Mozilla. *Set-Cookie*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie (cit. on p. 114).

[Mozb] Mozilla. *Window postMessage*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage (cit. on p. 100).

[MSBK19] R. Mayrhofer, J. V. Stoep, C. Brubaker, N. Kralevich. *The Android Platform Security Model*. 2019. arXiv: 1904.05572 [cs.CR] (cit. on pp. 18, 20, 94, 96).

[Muea] B. Mueller. *Mobile App Security Verification Standard*. URL: https://mobile-security.gitbook.io/masvs/ (cit. on pp. 21, 32).

[Mueb] B. Mueller. *Mobile Security Testing Guide*. URL: https://mobile-security.gitbook.io/mobile-security-testing-guide/ (cit. on pp. 30, 32, 104).

[Mul] C. Mulliner. *Inside Android's SafetyNetAttestation: Attack andDefense*. URL: https://www.mulliner.org/collin/publications/inside_safetynet_attestation_attacks_and_defense_mulliner2017_ekoparty.pdf (cit. on p. 32).

[Ope] Open Banking Implementation Entity. *Open Banking*. URL: https://standards.openbanking.org.uk/ (cit. on p. 47).

[Rah] M. Rahman. *Magisk may no longer be able to hide bootloader unlocking from apps*. URL: https://www.xda-developers.com/magisk-no-longer-hide-bootloader-unlock-status/ (cit. on pp. 31, 32).

[RBT16] J. Richer, J. Bradley, H. Tschofenig. *A Method for Signing HTTP Requests for OAuth*. Internet-Draft draft-ietf-oauth-signed-http-request-03. Work in Progress. Internet Engineering Task Force, Aug. 2016. 13 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-signed-http-request-03 (cit. on p. 57).

[Ric15] J. Richer. *OAuth 2.0 Token Introspection*. RFC 7662. Oct. 2015. DOI: 10.17487/RFC7662. URL: https://rfc-editor.org/rfc/rfc7662.txt (cit. on p. 37).

[RJB+15] J. Richer, M. Jones, J. Bradley, M. Machulak, P. Hunt. *OAuth 2.0 Dynamic Client Registration Protocol*. RFC 7591. July 2015. DOI: 10.17487/RFC7591. URL: https://rfc-editor.org/rfc/rfc7591.txt (cit. on pp. 40, 41, 53, 57).

[Rod] O. Rodriguez. *10 things you might be doing wrong when using the SafetyNet Attestation API*. URL: https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html (cit. on p. 31).

[Saka] N. Sakimura. *Financial-grade API - Part 1: Baseline Security Profile*. OpenID Foundation. URL: https://openid.net/specs/openid-financial-api-part-1.html (cit. on p. 47).

[Sakb] N. Sakimura. *Financial-grade API - Part 2: Advanced Security Profile*. OpenID Foundation. URL: https://openid.net/specs/openid-financial-api-part-2.html (cit. on p. 47).

[Sakc] N. Sakimura. *CIBA FAPI*. URL: https://bitbucket.org/openid/fapi/src/master/Financial_API_WD_CIBA.md (cit. on p. 45).

[Sam19] Samsung. *Statement on Fingerprint Recognition Issue*. Oct. 18, 2019. URL: https://news.samsung.com/global/statement-on-fingerprint-recognition-issue (cit. on p. 112).

[SBA15] N. Sakimura, J. Bradley, N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. Sept. 2015. DOI: 10.17487/RFC7636. URL: https://rfc-editor.org/rfc/rfc7636.txt (cit. on p. 43).

[SBJ20] N. Sakimura, J. Bradley, M. Jones. *The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)*. Internet-Draft draft-ietf-oauth-jwsreq-30. Work in Progress. Internet Engineering Task Force, Sept. 2020. 35 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-oauth-jwsreq-30` (cit. on p. 41).

[SHJ20] Y. Sheffer, D. Hardt, M. Jones. *JSON Web Token Best Current Practices*. RFC 8725. Feb. 2020. DOI: `10.17487/RFC8725`. URL: `https://rfc-editor.org/rfc/rfc8725.txt` (cit. on pp. 67, 111).

[SM14] M. Shehab, F. Mohsen. "Towards Enhancing the Security of OAuth Implementations in Smart Phones". In: *2014 IEEE International Conference on Mobile Services*. 2014, pp. 39–46 (cit. on pp. 23, 99).

[Spa] Sparkasse. *Die Apps der Sparkassen*. URL: `https://www.sparkasse.de/unsere-loesungen/privatkunden/rund-ums-konto/sparkassen-apps.html` (cit. on p. 66).

[Stö18] M. Stötzner. *Über die (Un-)Sicherheit des W3C-WebAuthentication-Entwurfs: Eine Beschreibung und Sicherheitsanalyse*. de. 2018. DOI: `10.18419/OPUS-10372`. URL: `http://elib.uni-stuttgart.de/handle/11682/10389` (cit. on p. 113).

[W3C] W3C. *Web Authentication:An API for accessing Public Key CredentialsLevel 1*. URL: `https://www.w3.org/TR/webauthn/` (cit. on p. 113).

[Wu] J. Wu. *Magisk - The Magic Mask for Android*. URL: `https://github.com/topjohnwu/Magisk` (cit. on p. 32).

[Yub] Yubico. *Protect your digital world with YubiKey*. URL: `https://www.yubico.com/` (cit. on p. 113).

[ZJL+15] X. Zheng, J. Jiang, J. Liang, H.-X. Duan, S. Chen, T. Wan, N. Weaver. "Cookies Lack Integrity: Real-World Implications". In: *USENIX Security Symposium*. 2015 (cit. on p. 114).

All links were last followed on November 25, 2020.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift