An Extensive Formal Security Analysis of the OpenID Financial-grade API

Daniel Fett
yes.com AG
mail@danielfett.de

Pedram Hosseyni
University of Stuttgart, Germany
pedram.hosseyni@sec.uni-stuttgart.de

Ralf Küsters
University of Stuttgart, Germany
ralf.kuesters@sec.uni-stuttgart.de

***Abstract*—Forced by regulations and industry demand, banks worldwide are working to open their customers' online banking accounts to third-party services via web-based APIs. By using these so-called *Open Banking* APIs, third-party companies, such as FinTechs, are able to read information about and initiate payments from their users' bank accounts. Such access to financial data and resources needs to meet particularly high security requirements to protect customers.**

One of the most promising standards in this segment is the *OpenID Financial-grade API (FAPI)*, currently under development in an open process by the OpenID Foundation and backed by large industry partners. The FAPI is a profile of OAuth 2.0 designed for high-risk scenarios and aiming to be secure against very strong attackers. To achieve this level of security, the FAPI employs a range of mechanisms that have been developed to harden OAuth 2.0, such as *Code and Token Binding* (including mTLS and OAUTB), *JWS Client Assertions*, and *Proof Key for Code Exchange*.

In this paper, we perform a rigorous, systematic formal analysis of the security of the FAPI, based on an existing comprehensive model of the web infrastructure—the *Web Infrastructure Model (WIM)* proposed by Fett, Küsters, and Schmitz. To this end, we first develop a precise model of the FAPI in the WIM, including different profiles for read-only and read-write access, different flows, different types of clients, and different combinations of security features, capturing the complex interactions in a web-based environment. We then use our model of the FAPI to precisely define central security properties. In an attempt to prove these properties, we uncover partly severe attacks, breaking authentication, authorization, and session integrity properties. We develop mitigations against these attacks and finally are able to formally prove the security of a fixed version of the FAPI.

Although financial applications are high-stakes environments, this work is the first to formally analyze and, importantly, verify an Open Banking security profile.

By itself, this analysis is an important contribution to the development of the FAPI since it helps to define exact security properties and attacker models, and to avoid severe security risks before the first implementations of the standard go live.

Of independent interest, we also uncover weaknesses in the aforementioned security mechanisms for hardening OAuth 2.0. We illustrate that these mechanisms do not necessarily achieve the security properties they have been designed for.

I. Introduction

Delivering financial services has long been a field exclusive to traditional banks. This has changed with the emergence of FinTech companies that are expected to deliver more than 20% of all financial services in 2020 [1]. Many FinTechs provide services that are based on access to a customers online banking account information or on initiating payments from a customers bank account.

For a long time, screen scraping has been the primary means of these service providers to access the customer's data at the bank. Screen scraping means that the customer enters online banking login credentials at the service provider's website, which then uses this data to log into the customer's online banking account by emulating a web browser. The service provider then retrieves account information (such as the balance or recent activities) and can trigger, for example, a cash transfer, which may require the user to enter her second-factor authentication credential (such as a TAN) at the service provider's web interface.

Screen scraping is inherently insecure: first of all, the service provider gets to know all login credentials, including the second-factor authentication of the customer. Also, screen scraping is prone to errors, for example, when the website of a bank changes.

Over the last years, the terms *API banking* and *Open Banking* have emerged to mark the introduction of standardized interfaces to financial institutions' data. These interfaces enable third parties, in particular FinTech companies, to access users' bank account information and initiate payments through well-defined APIs. All around the world, API banking is being promoted by law or by industry demand: In Europe, the *Payment Services Directive 2 (PSD2)* regulation mandates all banks to introduce Open Banking APIs by September 2019 [2]. The U.S. Department of the Treasury recommends the implementation of such APIs as well [3]. In South Korea, India, Australia, and Japan, open banking is being pushed by large financial corporations [4].

One important open banking standard currently under development for this scenario is the *OpenID Financial-grade API (FAPI)*.[1] The FAPI [5] is a profile (i.e., a set of concrete protocol flows with extensions) of the *OAuth 2.0 Authorization Framework* and the identity layer *OpenID Connect* to provide a secure authorization and authentication scheme for high-risk scenarios. The FAPI is under development at the OpenID Foundation and supported by many large corporations, such as Microsoft and the largest Japanese consulting firm, Nomura Research Institute. The OpenID Foundation is also cooperating

[1] In its current form, the FAPI does not (despite its name) define an API itself, but defines a security profile for the access to APIs.

with other banking standardization groups: The UK Open Banking Implementation Entity, backed by nine major UK banks, has adopted the FAPI security profile.

The basic idea behind the FAPI is as follows: The owner of the bank account (*resource owner*, also called user in what follows) visits some website or uses an app which provides some financial service. The website or app is called a *client* in the FAPI terminology. The client redirects the user to the *authorization server*, which is typically operated by the bank. The authorization server asks for the user's bank account credentials. The user is then redirected back to the client with some token. The client uses this token to obtain bank account information or initiate a payment at the *resource server*, which is typically also operated by the bank.

The FAPI aims to be secure against much stronger attackers than its foundations, OAuth 2.0 and OpenID Connect: the FAPI assumes that sensitive tokens leak to an attacker through the user's browser or operating system, and that endpoint URLs can be misconfigured. On the one hand, both assumptions are well motivated by real-world attacks and the high stakes nature of the environment where the FAPI is to be used. On the other hand, they directly break the security of OAuth 2.0 and OpenID Connect.

To provide security against such strong attackers, the FAPI employs a range of OAuth 2.0 security extensions beyond those used in plain OAuth 2.0 and OpenID Connect: the FAPI uses the so-called Proof Key for Code Exchange (PKCE)[2] extension to prevent unauthorized use of tokens. For client authentication towards the authorization server, the FAPI employs *JWS Client Assertions* or *mutual TLS*. Additionally, *OAuth token binding*[3] or *certificate-bound access tokens*[4] can be used as holder-of-key mechanisms. To introduce yet another new feature, the FAPI is the first standard to make use of the so-called JWT Secured Authorization Response Mode (JARM).

The FAPI consists of two main so-called *parts*, here also called modes, that stipulate different security profiles for read-only access to resource servers (e.g., to retrieve bank account information) and read-write access (e.g., for payment initiation). Both modes can be used by *confidential* clients, i.e., clients that can store and protect secrets (such as web servers), and by *public* clients that cannot securely store secrets, such as JavaScript browser applications. Combined with the new security features, this gives rise to many different settings and configurations in which the FAPI can run (see also Figure 3).

This, the expected wide adoption, the exceptionally strong attacker model, and the new security features make the FAPI a particularly interesting, challenging, and important subject for a detailed security analysis. While the security of (plain) OAuth 2.0 and OpenID Connect has been studied formally and informally many times before [6]–[21], there is no such analysis for the FAPI—or any other open banking API—so far. In particular, there are no results in the strong attacker model adopted for the FAPI, and there has been no formal security analysis of the additional OAuth security mechanisms employed by the FAPI (PKCE, JWS Client Assertions, mTLS Client Authentication, OAuth Token Binding, Certificate-Bound Access Tokens, JARM), which is of practical relevance in its own right.

In this paper, we therefore study the security of the FAPI in-depth, including the OAuth security extensions. Based on a detailed formal model of the web, we formalize the FAPI with its various configurations as well as its security properties. We discover four previously unknown and severe attacks, propose fixes, and prove the security of the fixed protocol based on our formal model of the FAPI, again considering the various configurations in which the FAPI can run. Importantly, this also sheds light on new OAuth 2.0 security extensions. In detail, our contributions are as follows:

Contributions of this Paper: We build a **detailed formal model of the FAPI** based on a comprehensive formal model of the web infrastructure proposed by Fett et al. in [22], which we refer to as the Web Infrastructure Model (WIM). The WIM has been successfully used to find vulnerabilities in and prove the security of several web applications and standards [6], [7], [22]–[24]. It captures a wide set of web features from DNS to JavaScript in unrivaled detail and comprehensiveness. In particular, it accounts for the intricate inner workings of web browsers and their interactions with the web environment. The WIM is ideally suited to identify logical flaws in web protocols, detect a range of standard web vulnerabilities (like cross-site request forgery, session fixation, misuse of certain web browser features, etc.), and even to find new classes of web attacks.

Based on the generic descriptions of web servers in the WIM, our models for FAPI clients and authorization servers contain all important features currently proposed in the FAPI standards. This includes the flows from both parts of the FAPI, as well as the different options for client authentication, holder-of-key mechanisms, and token binding mentioned above.

Using this model of the FAPI, we define precise **security properties** for authorization, authentication, and session integrity. Roughly speaking, the authorization property requires that an attacker is unable to access the resources of another user at a bank, or act on that user's behalf towards the bank. Authentication means that an attacker is unable to log in at a client using the identity of another user. Session integrity means that an attacker is unable to force a user to be logged in at a client under the attackers identity, or force a user to access (through the client) the attacker's resources instead of the user's own resources (session fixation).

During our first attempts to prove these properties, we **discovered four unknown attacks** on the FAPI. With these attacks, adversaries can gain access to the bank account of a user, break session integrity, and, interestingly, circumvent certain OAuth security extensions, such as PKCE and Token Binding, employed by the FAPI.

We notified the OpenID FAPI Working Group of the attacks and vulnerabilities found by our analysis and are working together with them to fix the standard. To this end, we first

[2]Pronounced *pixie*, RFC 7636.

[3]https://tools.ietf.org/html/draft-ietf-oauth-token-binding-07

[4]https://tools.ietf.org/html/draft-ietf-oauth-mtls-11

developed mitigations against the vulnerabilities. We then, as another main contribution of our work and to support design decisions during the further development of the FAPI, implemented the fixes in our formal model and provided the **first formal proof of the security of the FAPI** (with our fixes applied) within our model of the FAPI, including all configurations of the FAPI and the various ways in which the new OAuth security extensions are employed in the FAPI (see Figure 3). This makes the FAPI the only open banking API to enjoy a thorough and detailed formal security analysis.

Our findings also show that (1) several **OAuth 2.0 security extensions** do not necessarily achieve the security properties they have been designed for and that (2) combining these extensions in a secure way is far from trivial. These results are relevant for all web applications and standards which employ such extensions.

Structure of this Paper: We first, in Section II, recall OAuth 2.0 and OpenID Connect as the foundations of the FAPI. We also introduce the new defense mechanisms that set the FAPI apart from "traditional" OAuth 2.0 and OpenID Connect flows. This sets the stage for Section III where we go into the details of the FAPI and explain its design and features. In Section IV, we present the attacks on the FAPI (and the new security mechanisms it uses), which are the results of our initial proof attempts, and also present our proposed fixes. The model of the FAPI and the analysis are outlined in Section V, along with a high-level introduction to the Web Infrastructure Model we use as the basis for our formal model and analysis of the FAPI. We conclude in Section VI. The appendix contains further details. Full details and proofs are provided in our technical report [25].

II. OAuth and New Defense Mechanisms

The *OpenID Financial-grade API* builds upon the OAuth 2.0 Authorization Framework [26]. Compared to the original OAuth 2.0 protocol, the FAPI aims at providing a much higher degree of security. For achieving this, the FAPI security profiles incorporate mechanisms defined in *OpenID Connect* [27] (which itself builds upon OAuth 2.0), and importantly, security extensions for OAuth 2.0 developed only recently by the IETF and the OpenID Foundation.

In the following, we give a brief overview of both OAuth 2.0 and OpenID Connect, and their security extensions used (among others) within the FAPI, namely *Proof Key for Code Exchange*, *JWS Client Assertions*, *OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens*, *OAuth 2.0 Token Binding* and the *JWT Secured Authorization Response Mode*. The FAPI itself is presented in Section III.

A. Fundamentals of OAuth 2.0 and OpenID Connect

OAuth 2.0 and OpenID Connect are widely used for various authentication and authorization tasks. In what follows, we first explain OAuth 2.0 and then briefly OpenID Connect, which is based on OAuth 2.0.

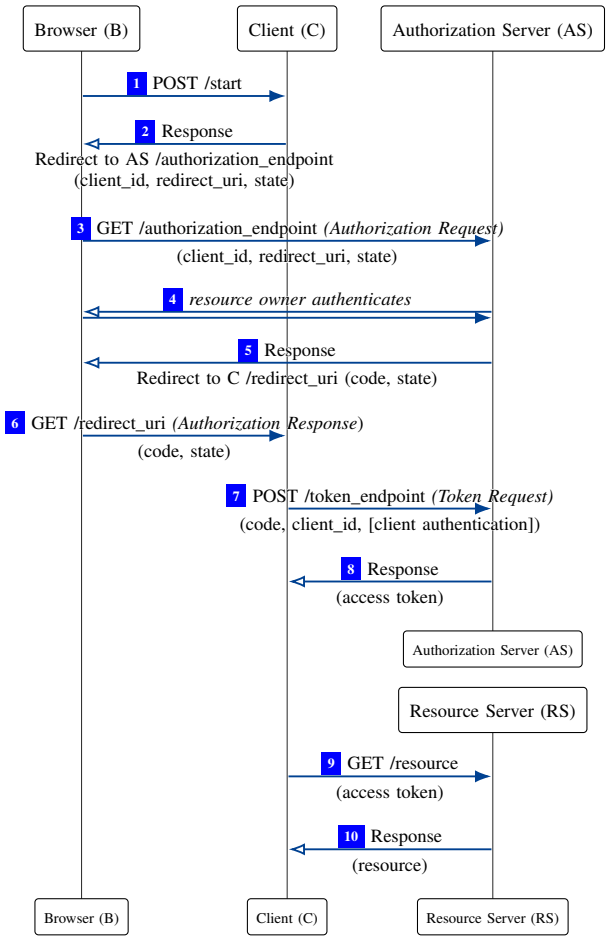

Figure 1. Overview of the OAuth Authorization Code Flow

1) OAuth 2.0: On a high level, OAuth 2.0 allows a *resource owner*, or user, to enable a *client*, a website or an application, to access her resources at some *resource server*. In order for the user to grant the client access to her resources, the user has to authenticate herself at an *authorization server*.

For example, in the context of the FAPI, resources include the user's account information (like balance and previous transactions) at her bank or the initiation of a payment transaction (cash transfer). The client can be a FinTech company which wants to provide a financial service to the user via access to the user's bank account. More specifically, the client might be the website of such a company (*web server client*) or the company's app on the user's device. The resource and authorization servers would typically be run by the user's bank. One client can make use of several authorization and resource servers.

RFC 6749 [26] defines multiple modes of operation for OAuth 2.0, so-called *grant types*. We here focus on the *authorization code grant* since the other grant types are not used in the FAPI.

Figure 1 shows the authorization code grant, which works as follows: The user first visits the client's website or opens

the client's app on her smartphone and selects to log in or to give the client access to her resources (Step 1). The client then redirects the user to the so-called *authorization endpoint* at the authorization server (AS) in Steps 2 and 3. (Endpoints are URIs used in the OAuth flow.) In this redirection, the client passes several parameters to the AS, for example, the *client id* which identifies the client at the AS, a *state* value that is used for CSRF protection,[5] a *scope* parameter (not shown in Figure 1) that describes the permissions requested by the client, and a redirection URI explained below. Note that if the client's app is used, the redirection from the app to the AS (Step 2) is done by opening the website of the AS in a browser window. The AS authenticates the user (e.g., by the user entering username and password) in Step 4 and asks for her consent to give the client access to her resources. The AS then creates a so-called *authorization code* (typically a nonce) and redirects the user back to the so-called *redirection endpoint* of the client via the user's browser in Steps 5 and 6. (If the client's app is used, a special redirect URI scheme, e.g., some-app://, is used which causes the operating system to forward the URI to the client's app.) At the AS, one or more redirection endpoints for a client are preregistered.[6] In Step 2, the client chooses one of these preregistered URIs. The authorization response (Step 5) is a redirection to this URI, with the authorization code, the state value from the request, and optionally further values appended as URI parameters.

When receiving the request resulting from the redirection in Step 6, the client first checks that the state value is the same as the one in the authorization request, typically by looking it up in the user's session with the client. If it is not the same, then the client suspects that an attacker tried to inject an authorization code into the client's session (cross-site request forgery, CSRF) and aborts the flow (see also Footnote 5). Otherwise, the client now exchanges the code for an *access token* at the so-called *token endpoint* of the AS in Steps 7 and 8. For this purpose, the client might be required to authenticate to the AS (see below). With this access token, the client can finally access the resources at the resource server (RS), as shown in Steps 9 and 10.

The RS can use different methods to check the validity of an access token presented by a client. The access token can, for example, be a document signed by the AS containing all necessary information. Often, the access token is not a structured document but a nonce. In this case, the RS uses Token Introspection [28], i.e., it sends the access token to the *introspection endpoint* of the AS and receives the information associated with the token from the AS. An RS typically has only one (fixed) AS, which means that when the RS receives an access token, it sends the introspection request to this AS.

[5]The state value is a nonce. The client later ensures that it receives the same nonce in the authorization response. Otherwise, an attacker could authenticate to the AS with his own identity and use the corresponding authorization response for logging in an honest user under the attacker's identity with a CSRF attack. This attack is also known as *session swapping*.

[6]Without preregistration, a malicious client starting a login flow with the client id of an honest client could receive a code associated with the honest client.

Public and Confidential Clients: Depending on whether a client can keep long-term secrets, it is either called a *public* or a *confidential* client. If the client is not able to maintain secrets, as is typically the case for applications running on end-user devices, the client is not required to authenticate itself at the token endpoint of the AS. These kinds of clients are called *public* clients. Clients able to maintain secrets, such as web server clients, must authenticate to the token endpoint (in Step 7 of Figure 1) and are called *confidential* clients.

For confidential clients, client authentication ensures that only a legitimate client can exchange the authorization code for an access token. OAuth 2.0 allows for several methods for client authentication at the token endpoint, including sending a password or proving possession of a secret [26, Section 2.3]. For public clients, other measures are available, such as PKCE (see below), to obtain a sufficient level of security.

2) OpenID Connect: OAuth 2.0 is built for *authorization* only, i.e., the client gets access to the resources of the user only if the user consented to this access. It does not per se provide *authentication*, i.e., proving the identity of the user to the client. This is what OpenID Connect [27] was developed for. It adds an *id token* to OAuth 2.0 which is issued by the AS and contains identity information about the end-user. ID tokens can be issued in the response from the authorization endpoint (Step 5 of Figure 1) and/or at the token endpoint (Step 8 of Figure 1). They are signed by the AS and can be bound to other parameters of the response, such as the hash of authorization codes or access tokens. Therefore, they can also be used to protect responses against modification.

B. Proof Key for Code Exchange

The *Proof Key for Code Exchange* (PKCE) extension (RFC 7636) was initially created for OAuth public clients and independently of the FAPI. Its goal is to protect against the use of intercepted authorization codes. Before we explain how it works, we introduce the attack scenario against which PKCE should protect according to RFC 7636.

This attack starts with the leakage of the authorization code after the browser receives it in the response from the authorization endpoint (Step 5 of Figure 1). A multitude of problems can lead to a leak of the code, even if TLS is used to protect the network communication:

- On mobile operating systems, multiple apps can register themselves onto the same custom URI scheme (e.g., some-app://redirection-response). When receiving the authorization response, the operating system may forward the response (and the code) to a malicious app instead of the honest app (see [29, Section 1] and [30, Section 8.1]).
- Mix-up attacks, in which a different AS is used than the client expects (see [6] for details), can be used to leak an authorization code to a malicious server.
- As highlighted in [7], a Referer header can leak the code to an adversary.
- The code can also appear in HTTP logs that can be disclosed (accidentally) to third parties or (intentionally) to administrators.

In a setting with a public client (i.e., without client authentication at the token endpoint), an authorization code leaked to the attacker can be redeemed directly by the attacker at the authorization server to obtain an access token.

RFC 7636 aims to protect against such attacks even if not only the authorization response leaks but also the authorization request as well. Such leaks can happen, for example, from HTTP logs (Precondition 4b of Section 1 of RFC 7636) or unencrypted HTTP connections.

PKCE works as follows: Before sending the authorization request, the client creates a random value called *code verifier*. The client then creates the *code challenge* by hashing the verifier[7] and includes the challenge in the authorization request (Step 2 of Figure 1). The AS associates the generated authorization code with this challenge. Now, when the client redeems the code in the request to the token endpoint (Step 7 of Figure 1), it includes the code verifier in the token request. This message is sent directly to the AS and protected by TLS, which means that the verifier cannot be intercepted. The idea is that if the authorization code leaked to the attacker, the attacker still cannot redeem the code to obtain the access token since he does not know the code verifier.

C. Client Authentication using JWS Client Assertions

As mentioned above, the goal of client authentication is to bind an authorization code to a certain confidential client such that only this client can redeem the code at the AS. One method for client authentication is the use of JWS Client Assertions [27, Section 9], which requires proving possession of a key instead of sending a password directly to the authorization server, as in plain OAuth 2.0.

To this end, the client first generates a short document containing its client identifier and the URI of the token endpoint. Now, depending on whether the *client secret* is a private (asymmetric) or a symmetric key, the client either signs or MACs this document. It is then appended to the token request (Step 7 of Figure 1). As the document contains the URI of the receiver, attacks in which the attacker tricks the client into using a wrong URI are prevented, as the attacker cannot reuse the document for the real endpoint (cf. Section III-C4). Technically, the short document is encoded as a JSON Web Token (JWT) [31] to which its signature/MAC is attached to create a so-called JSON Web Signature (JWS) [32].

D. OAuth 2.0 Mutual TLS

OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens (mTLS) [33] provides a method for both client authentication and token binding.

OAuth 2.0 Mutual TLS Client Authentication makes use of *TLS client authentication*[8] at the token endpoint (in Step 7 of Figure 1). In TLS client authentication, not only the server authenticates to the client (as is common for TLS) but the client also authenticates to the server. To this end, the client proves that it knows the private key belonging to a certificate that is either (a) self-signed and preconfigured at the respective AS or that is (b) issued for the respective client id by a predefined certificate authority within a public key infrastructure (PKI).

Token binding means binding an access token to a client such that only this client is able to use the access token at the RS. To achieve this, the AS associates the access token with the certificate used by the client for the TLS connection to the token endpoint. In the TLS connection to the RS (in Step 9 of Figure 1), the client then authenticates using the same certificate. The RS accepts the access token only if the client certificate is the one associated with the access token.[9]

E. OAuth 2.0 Token Binding

OAuth 2.0 Token Binding (OAUTB) [34] is used to bind access tokens and/or authorization codes to certain TLS connections. It is based on the *Token Binding* protocol [35]–[38] and can be used with all TLS versions. In the following, we first sketch token binding in general before we explain OAuth 2.0 Token Binding.

1) Basics: For simplicity of presentation, in the following, we assume that a browser connects to a web server. The protocol remains the same if the browser is replaced by another server. (In the context of OAuth 2.0, in some settings in fact the client takes the role of the browser as explained below.)

At its core, token binding works as follows: When a web server indicates (during TLS connection establishment) that it wants to use token binding, the browser making the HTTP request over this TLS connection creates a public/private key pair for the web server's origin. It then sends the public key to the server and proves possession of the private key by using it to create a signature over a value unique to the current TLS connection. Since the browser re-uses the same key pair for future connections to the same origin, the web server will be able to unambiguously recognize the browser in future visits.

Central for the security of token binding is that the private key remains secret inside the browser. To prevent replay attacks, the browser has to prove possession of the private key by signing a value that is unique for each TLS session. To this end, token binding uses the *Exported Keying Material* (EKM) of the TLS connection, a value derived from data of the TLS handshake between the two participants, as specified in [38]. As long as at least one party follows the protocol, the EKM will be unique for each TLS connection.

We can now illustrate the usage of token binding in the context of a simplified protocol in which a browser B requests a token from a server S: First, B initiates a TLS connection to S, where B and S use TLS extensions [36] to negotiate the use of token binding and technical details thereof. Browser B then creates a public/private key pair $(k_{B,S}, k'_{B,S})$ for the origin of S, unless such a key pair exists already. The public key $k_{B,S}$

[7] If it is assumed that the authorization request never leaks to the attacker, it is sufficient and allowed by RFC 7636 to use the verifier as the challenge, i.e., without hashing.

[8] As noted in [33], Section 5.1 this extension supports all TLS versions with certificate-based client authentication.

[9] As mentioned above, the RS can read this information either directly from the access token if it is a signed document, or uses token introspection to retrieve the data from the AS.

(together with technical details about the key, such as its bit length) is called *Token Binding ID* (for the specific origin).

When sending the first HTTP request over the established TLS connection, B includes in an HTTP header the so-called *Token Binding Message*:

$$\mathsf{TB\text{-}Msg}[k_{B,S}, \mathsf{sig}(EKM, k'_{B,S})] \tag{1}$$

It contains both the Token Binding ID (i.e., essentially $k_{B,S}$) and the signed EKM value from the TLS connection, as specified in [39]. The server S checks the signature using $k_{B,S}$ as included in this message and then creates a token and associates it with the Token Binding ID as the unique identifier of the browser.

When B wants to redeem the token in a new TLS connection to S, B creates a new Token Binding Message using the same Token Binding ID, but signs the new EKM value:

$$\mathsf{TB\text{-}Msg}[k_{B,S}, \mathsf{sig}(\overline{EKM}, k'_{B,S})] \tag{2}$$

As the EKM values are unique to each TLS connection, S concludes that the sender of the message knows the private key of the Token Binding ID, and as the sender used the same Token Binding ID as before, the same party that requested the token in the first request is using it now.

The above describes the simple situation that B wants to redeem the token received from S again at S, i.e., from the same origin. In this case, we call the token binding message in (1) a *provided* token binding message. If B wants to redeem the token received from S at another origin, say at C, then instead of just sending the provided token message in (1), B would in addition also send the so-called *referred* token binding message, i.e., instead of (1) B would send

$$\begin{aligned} &\mathsf{TB\text{-}prov\text{-}Msg}[k_{B,S}, \mathsf{sig}(EKM, k'_{B,S})], \\ &\mathsf{TB\text{-}ref\text{-}Msg}[k_{B,C}, \mathsf{sig}(EKM, k'_{B,C})]. \end{aligned} \tag{3}$$

Note that the EKM is the same in both messages, namely the EKM value of the TLS connection between B and S (rather than between B and C, which has not happened yet anyway). Later when B wants to redeem the token at C, B would use $k_{B,C}$ in its (provided) token message to C.

2) Token Binding for OAuth: In the following, we explain how token binding is used in OAuth in the case of app clients. The case of web server clients is discussed below.

The flow is shown in Figure 2. Note that in this case, token binding is used between the OAuth client and the authorization and resource servers; the browser in Figure 1 is not involved.

The client has two token binding key pairs, one for the AS and one for the RS (if these key pairs do not already exist, the client creates them during the flow). When sending the authorization request (Step 2 of Figure 2), the client includes the hash of the Token Binding ID it uses for the AS as a PKCE challenge (cf. Section II-B). When exchanging the code for an access token in Step 7, the client proves possession of the private key of this Token Binding ID, and the AS only accepts the request when the hash of the Token Binding ID is the same as the PKCE challenge. Therefore, the code can only be exchanged by the participant that created the authorization

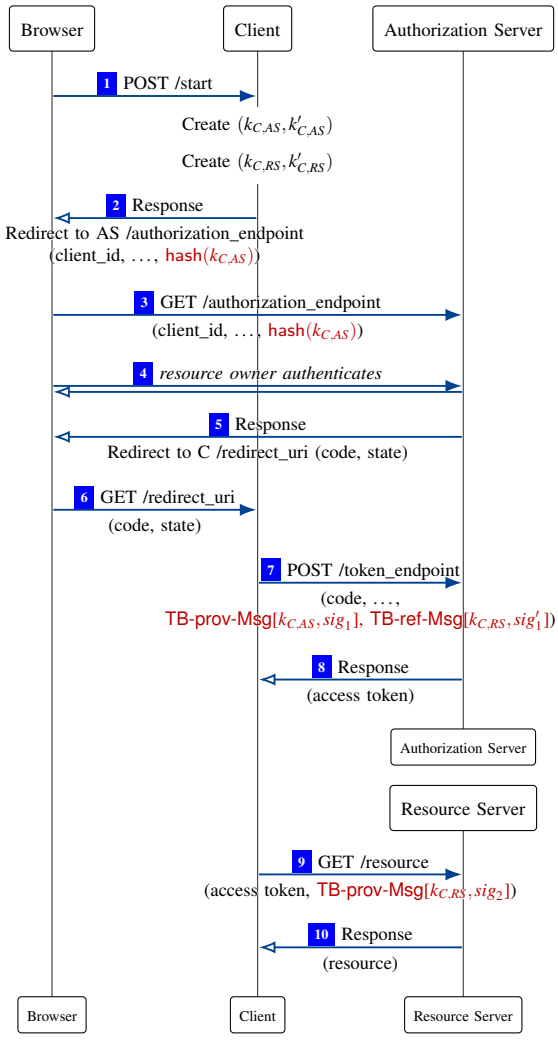

Figure 2. OAUTB for App Clients

request. Note that for this purpose the AS only takes the *provided* token binding message sent to the AS in Step 7 into account. However, the AS also checks the validity of the *referred* token binding message (using the same EKM value) and associates $k_{C,RS}$ with the token issued by the AS in Step 8.

The token binding ID $k_{C,RS}$ is used in Step 9 by the client to redeem the token at the RS. The RS then checks if this is the same token binding ID that is associated with the access token. This information can be contained in the access token if it is structured and readable by the RS or via token introspection.

Altogether, Token Binding for OAuth (in the case of app clients) is supposed to bind both the authorization code and the access token to the client. That is, only the client who initiated the flow (in Step 2) can redeem the authorization code at the AS and the corresponding access token at the RS, and hence, get access to the resource at the RS.

3) Binding Authorization Codes for Web Server Clients: In the case that the client is a web server, the binding of the authorization code to the client is already done by client

authentication, as a web server client is always confidential (cf. Section II-A1). Therefore, the client does *not* include the hash of a Token Binding ID in the authorization request (Step 2 of Figure 2). Instead, the mechanism defined in OAUTB aims at binding the authorization code to the browser/client pair. (The binding of the access token to the client is done in the same way as for an app client).

More precisely, for web server clients, the authorization code is bound to the token binding ID that the browser uses for the client. For this purpose, the client includes an additional HTTP header in the first response to the browser (Step 2 of Figure 2), which signals the browser that it should give the token binding ID it uses for the client to the authorization server. When sending the authorization request to the authorization server in Step 3, the browser thus includes a provided and a referred token binding message, where the referred message contains the token binding ID, that the browser later uses for the client (say, $k_{B,C}$). When generating the authorization code, the authorization server associates the code with $k_{B,C}$.

When redirecting the code to the client in Step 6, the browser includes a token binding message for $k_{B,C}$, thereby proving possession of the private key.

When sending the token request in Step 7, the client includes $k_{B,C}$. We highlight that the client does not send a token binding message for $k_{B,C}$ since the client does not know the corresponding private key (only the browser does).

The authorization server checks if this key is the same token binding ID it associated the authorization code with, and therefore, can check if the code was redirected to the client by the same browser that made the authorization request. In other words, by this the authorization code is bound to the browser/client pair.

F. JWT Secured Authorization Response Mode

The recently developed *JWT Secured Authorization Response Mode* (JARM) [40] aims at protecting the OAuth authorization response (Step 5 of Figure 1) by having the AS sign (and optionally encrypt) the response. The authorization response is then encoded as a JWT (see Section II-C). The JARM extension can be used with any OAuth 2.0 flow.

In addition to the regular parameters of the authorization response, the JWT also contains its issuer (identifying the AS) and its audience (client id). For example, if combined with the Authorization Code Flow, the response JWT contains the issuer, audience, authorization code, and state values.

By using JARM, the authorization response is integrity protected and injection of leaked authorization codes is prevented.

III. The OpenID Financial-grade API

The OpenID Financial-grade API [5] currently comprises two implementer's drafts. One defines a profile for read-only access, the other one for read-write access. Building on Section II, here we describe both profiles and the various configurations in which these profiles can run (see Figure 3). Furthermore, we explain the assumptions made within the FAPI standard and the underlying OAuth 2.0 extensions.

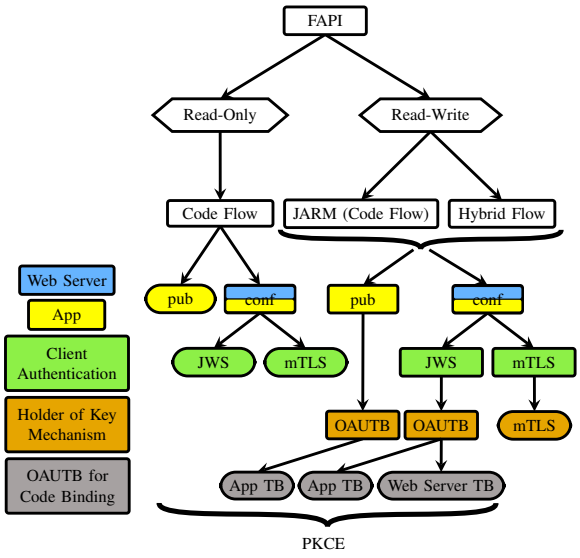

Figure 3. Overview of the FAPI. One path (terminated by a box with rounded corners) describes one possible configuration of the FAPI. The paths marked with PKCE use PKCE. JARM and Hybrid flows both allow for the configurations shown.

A. Financial-grade API: Read-Only Profile

In the following, we explain the Read-Only flow as described in [41]. The Read-Only profile aims at providing a secure way for accessing data that needs a higher degree of protection than regular OAuth, e.g., for read access to financial data.

The Read-Only flow is essentially an **OAuth Authorization Code flow** (cf. Section II). Additionally, the client can request an ID Token (see Section II-A2) from the token endpoint by adding a *scope* parameter to the authorization request (Step 2 of Figure 1) with the value `openid`.

In contrast to regular OAuth and OpenID Connect, the client is required to have a different set of **redirection URIs** for each authorization server. This separation prevents mix-up attacks, where the authorization response (Step 6 in Figure 1) comes from a different AS than the client expects (see [6] and [42] for more details on mix-up attacks). When receiving the authorization response, the client checks if the response was received at the redirection URI specified in the authorization request (Step 2 in Figure 1).

One of the main additions to the regular OAuth flow is the use of **PKCE** as explained in Section II-B. The PKCE challenge is created by hashing a nonce.

The FAPI furthermore requires **confidential clients to authenticate** at the token endpoint (in Step 7 of Figure 1) using either *JWS Client Assertions* (cf. Section II-C) or *Mutual TLS* (cf. Section II-D). Public clients do not use client authentication.

B. Financial-grade API: Read-Write Profile

The Read-Write profile [43] aims at being secure under stronger assumptions than the Read-Only profile, in order to

be suitable for scenarios such as write access to financial data. The full set of assumptions is described in Section III-C.

The flow can be either an **OpenID Connect (OIDC) Hybrid flow**, which means that both the authorization response (Step 5 in Figure 1) and the token response (Step 8 in Figure 1) contain an id token (see Section II-A2), or any other OAuth-based flow used together with **JARM** (see Section II-F). When using the Hybrid flow, the FAPI profile also requires that the hash of the state value is included in the first id token.

In addition to the parameters of the Read-Only flow, the authorization request prepared by the client (Step 2 of Figure 1) is required to contain a **request JWS**, which is a JWT, signed by the client, containing all request parameters together with the audience of the request (cf. Section II-C).

One of the main security features of the profile is the **binding of the authorization code and the access token** to the client, which is achieved by using either mTLS (cf. Section II-D) or OAUTB (OAuth 2.0 Token Binding, see Section II-E). A public client is required to use OAUTB, while a confidential client can use either OAUTB or mTLS.

If the client is a confidential client using mTLS, the request does not contain a PKCE challenge. When using OAUTB, the client uses a **variant of PKCE**, depending on whether the client is a web server client or an app client (cf. Section II-E).

In the case of a confidential client, the **client authentication at the token endpoint** is done in the same way as for the Read-Only flow, i.e., by using either JWS Client Assertions (cf. Section II-C) or Mutual TLS (cf. Section II-D).

C. Overview of Assumptions and Mitigations

In the following, we explain the conditions under which the FAPI profiles and the OAuth extensions aim to be secure according to their specifications.

1) Leak of Authorization Response: As described in Section II-B in the context of PKCE, there are several scenarios in which the authorization response (Step 6 of Figure 1), and hence, the authorization code, can leak to the attacker (in clear), in particular in the case of app clients. In our model of the FAPI, we therefore assume that the authorization response is given to the attacker if the client is an app. At first glance, leakage of the authorization code is indeed mitigated by the use of PKCE since an attacker does not know the code verifier, and hence, cannot redeem the code at the AS. However, our attack described in Section IV-C shows that the protection provided by PKCE can be circumvented.

2) Leak of Authorization Request: The Read-Only profile of the FAPI explicitly states that the PKCE challenge should be created by hashing the verifier. The use of hashing should protect the PKCE challenge even if the authorization request leaks (e.g., by leaking HTTP logs, cf. Section II-B), and therefore, we assume in our model that the authorization request (Step 2 of Figure 1) leaks to the attacker.

3) Leak of Access Token: In the Read-Write profile, it is assumed that the access token might leak due to phishing [43, Section 8.3.5]. In our model, we therefore assume that the access token might leak in Step 5 of Figure 1. This problem is seemingly mitigated by using either mTLS or OAUTB, which bind the access token to the legitimate client, and hence, only the legitimate client should be able to redeem the access token at the RS even if the access token leaked. The FAPI specification states: "When the FAPI client uses MTLS or OAUTB, the access token is bound to the TLS channel, it is access token phishing resistant as the phished access tokens cannot be used." [43, Section 8.3.5]. However, our attack presented in Section IV-A shows that this is not the case.

4) Misconfigured Token Endpoint: An explicit design decision by the FAPI working group was to make the Read-Write profile secure even if the token request (Step 7 of Figure 1) leaks. The FAPI specification describes this attack as follows: "In this attack, the client developer is social engineered into believing that the token endpoint has changed to the URL that is controlled by the attacker. As the result, the client sends the code and the client secret to the attacker, which will be replayed subsequently." [43, Section 8.3.2].

Therefore, we make this assumption also in our FAPI model. Seemingly, this problem is mitigated by code binding through client authentication or OAUTB, which means that the attacker cannot use the stolen code at the legitimate token endpoint. "When the FAPI client uses MTLS or OAUTB, the authorization code is bound to the TLS channel, any phished client credentials and authorization codes submitted to the token endpoint cannot be used since the authorization code is bound to a particular TLS channel." [43, Section 8.3.2]. Note that in the FAPI the client does not authenticate by using the client secret as a password, but by proving possession (either using JWS Client Assertions or mTLS), which means that the attacker cannot reuse credentials.

However, our attack presented in Section IV-B shows that this intuition is misleading.

IV. Attacks

As already mentioned in the introduction, in Section V we present our rigorous formal analysis of the FAPI based on the Web Infrastructure Model. Through this formal analysis of the FAPI with the various OAuth 2.0 extensions it uses, we not only found attacks on the FAPI but also on some of the OAuth 2.0 extensions, showing that (1) these extensions do not achieve the security properties they have been designed for and (2) that combining these extensions in a secure way is far from trivial. Along with the attacks, we also propose fixes to the standards. Our formal analysis presented in Section V considers the fixed versions.

We start by describing two attacks on Token Binding, followed by an attack on PKCE, and one vulnerability hidden in the assumptions of PKCE.

We emphasize that our attacks work even if all communication uses TLS and even if the attacker is merely a web attacker, i.e., does not control the network but only certain parties.

As already mentioned in the introduction, we notified the OpenID FAPI Working Group of the attacks found by our analysis and are working together with them to fix the standard.

A. Cuckoo's Token Attack

As explained in Section III-C3, the Read-Write profile of the FAPI aims at providing security even if the attacker obtains an access token, e.g., due to phishing. Intuitively, this protection seems to be achieved by binding the access token to the client via mTLS (see Section II-D) or OAUTB (see Section II-E).

However, these mechanisms prevent the attacker only from directly using the access token in the same flow. As illustrated next, in a second flow, the attacker *can* inject the bound access token and let the client (to which the token is bound) use this token, which enables the attacker to access resources belonging to an honest identity.

This attack affects all configurations of the Read-Write profile (see Figure 3). Also, the Read-Only profile is vulnerable to this attack; this profile is, however, not meant to defend against stolen access tokens.

We note that the underlying principle of the attack should be relevant to other use-cases of token binding as well, i.e., whenever a token is bound to a participant, the involuntary use of a leaked token (by the participant to which the token is bound) should be prevented.

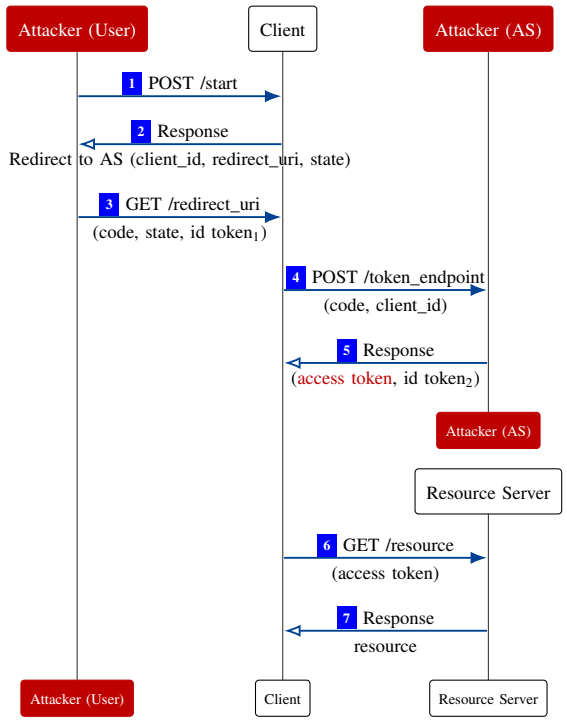

Figure 4. Cuckoo's Token Attack

Figure 4 depicts the attack for the OIDC Hybrid Flow, i.e., when both responses of the AS contain id tokens (see Section III-B). The attack works analogously for the code flow in combination with JARM (see Section III-B).

As explained, we assume that the attacker already obtained (phished) an access token issued by an honest AS to an honest client for accessing resources of an honest user. We also assume that the honest client supports the use of several ASs (a common setting in practice, as already mentioned in Section II), where in this case one of the ASs is dishonest.[10]

First, the attacker starts the flow at the client and chooses his own AS. Since he is redirected to his own AS in Step 2, he can skip the user authentication step and return an authorization response immediately. Apart from that, the flow continues normally until Step 4, where the client sends the code to the attacker AS. In Step 5, the attacker AS returns the previously phished access token together with the second id token.

Until here, all checks done by the client pass successfully, as the attacker AS adheres to the protocol. The only difference to an honest authorization server is that the attacker AS returns a phished access token. In Step 6, the resource server receives the (phished) access token and provides the client access to the honest resource owner's resources for the phished access token,[11] which implies that now the attacker has access to these resources through the client.

To prevent the use of leaked access tokens, the client should include, in the request to the RS, the identity of the AS the client received the access token from. The client can take this value from the second id token. Now, the RS would only continue the flow if its belief is consistent with the one of the RS. We apply an analogous fix for flows with JARM. These fixes are included in our model and shown to work in Section V.

B. Access Token Injection with ID Token Replay

As described in Section III-C3, the Read-Write profile aims to be secure if an attacker acquires an access token for an honest user. The profile also aims to be secure even if the token endpoint URI is changed to an attacker-controlled URI (see Section III-C4). Now, interestingly, these two threat scenarios combined in this order are the base for the attack described in the following. In this attack, the attacker returns an access token at the misconfigured token endpoint. While the attack looks similar to the previous attack at first glance, here the attacker first interacts with the *honest* AS and later *replays an id token* at the token endpoint. Both attacks necessitate different fixes. The outcome, however, is the same, and, just as the previous attack, this attack affects all configurations of the Read-Write profile, even if JARM is used. We explain the attack using the Hybrid Flow.

Figure 5 shows how the attack proceeds. The attacker initiates the Read-Write flow at the client and follows the regular flow until Step 6. As the authorization response was

[10] We highlight that we do not assume that the attacker controls the AS that issued the access token (i.e., the AS at which the honest user is registered). This means that the (honest) user uses an honest client and an honest authorization server.

[11] Which RS is used in combination with an AS depends on the configuration of the client, which is acquired through means not defined in OAuth. Especially in scenarios where this configuration is done dynamically, a dishonest AS might be used in combination with an honest RS. But also if the client is configured manually, as is often the case today, it might be misconfigured or social engineered into using specific endpoints. Recall from Section II-A that the access token might be a document signed by the (honest) AS containing all information the RS needs to process the access token. Alternatively, and more common, the RS performs token introspection, if the access token is just a nonce. The RS typically uses only one AS (in this case, the honest AS) to which it will send the introspection request.

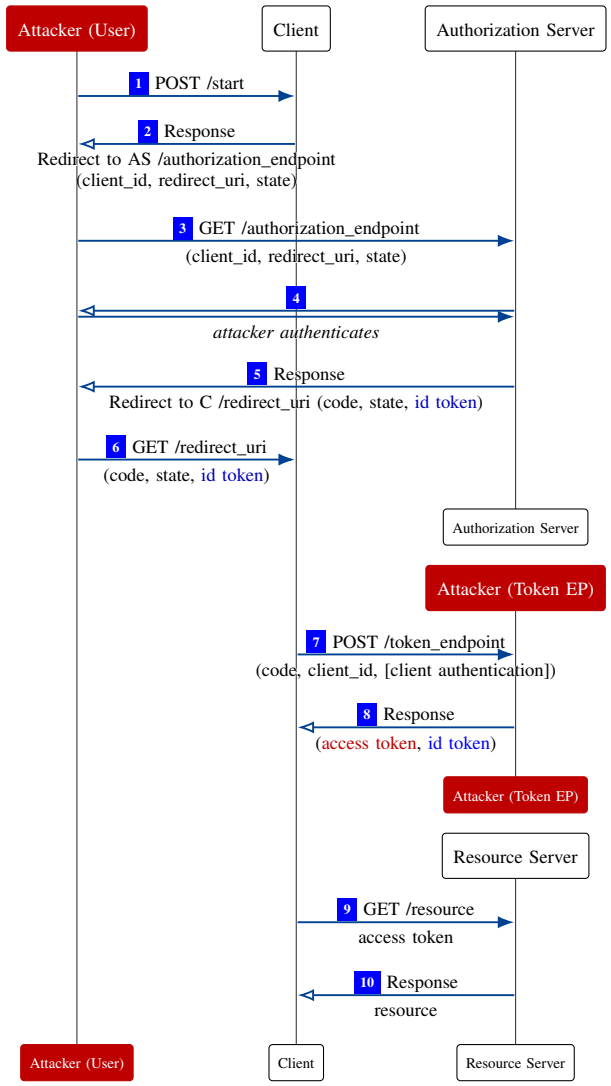

Figure 5. Access Token Injection with ID Token Replay Attack

created by the honest AS, the state and all values of the id token are correct and the client accepts the authorization response.

In Step 7, the client sends the token request to the misconfigured token endpoint controlled by the attacker. The value of the code and the checks regarding client authentication and proof of possession of keys are not relevant for the attacker.

In Step 8, the attacker sends the token response containing the phished access token. As the flow is an OIDC Hybrid Flow, the attacker is required to return an id token. Here, he returns the same id token that he received in Step 5, which is signed by the honest AS. The client is required to ensure that both id tokens have the same subject and issuer values, which in this case holds true since they are identical.

The client sends the access token to the honest resource server, by which the attacker gets read-write access to the resource of the honest resource owner through the client.

As we show in our security analysis (see Section V), this scenario is prevented if the second id token is required to contain the hash of the access token that is returned to the client, as the attacker cannot create id tokens with a valid signature of the AS. A similar fix also works for flows with JARM. The fixes are already included in our model.

C. PKCE Chosen Challenge Attack

As detailed in Section III-C1, the FAPI uses PKCE in order to protect against leaked authorization codes. This is particularly important for public clients as these clients, unlike confidential ones, do not authenticate to an AS when trying to exchange the code for an access token.

Recall that the idea of PKCE is that a client creates a PKCE challenge (hash of a nonce), gives it to the AS, and when redeeming the authorization code at the AS, the client has to present the correct PKCE verifier (the nonce). This idea works when just considering an honest flow in which the code leaks to the attacker, who does not know the PKCE verifier. However, our attack shows that the protection can be circumvented by an attacker who pretends to be an honest client.

This attack affects public clients who use the Read-Only profile of the FAPI. It works as follows (see Figure 6): As in RFC 7636, two apps are installed on a user's device, an honest app and a malicious app. The honest app is a client of an honest AS with the client identifier *hon_client_id* and the redirection URI *hon_redir_uri*. The malicious app is not registered at the AS.

The Read-Only flow starts at the malicious app, which prompts the user to log in. Now, the malicious app prepares an authorization request containing the client id and a redirect URI of the honest client (Step 2). At this point, the malicious app also creates a PKCE verifier and includes the corresponding challenge in the authorization request.

The flow continues until the browser receives the authorization response in Step 5. As the redirection URIs are preregistered at the AS, the redirection URI in the authorization request was chosen from the set of redirect URIs of the honest app, and therefore, the authorization response is redirected to the honest client after the browser receives it.

As described in Sections II-B and III-C1, at this point, the authorization response with the authorization code might leak to the attacker (Step 6). The malicious app is now able to exchange the code (associated with the honest client) at the token endpoint in Steps 7 and 8, as it knows the correct PKCE verifier and, as the honest app is a public client, without authenticating to the AS.

To prevent this scenario, an honest AS must ensure that the PKCE challenge was created by the client with the id *hon_client_id*. To achieve this, for public clients in the Read-Only flow we use the same mechanism that the FAPI uses for public clients in the Read-Write flow, namely the authorization request should contain a signed JWT (see also Section II-C, although JWTs are now used in a different way). This ensures that the client stated in the request actually made the request, and hence, no other client should know the PKCE verifier. Note that by using signed JWTs for public clients the FAPI assumes that public clients can store some secrets (which might, for

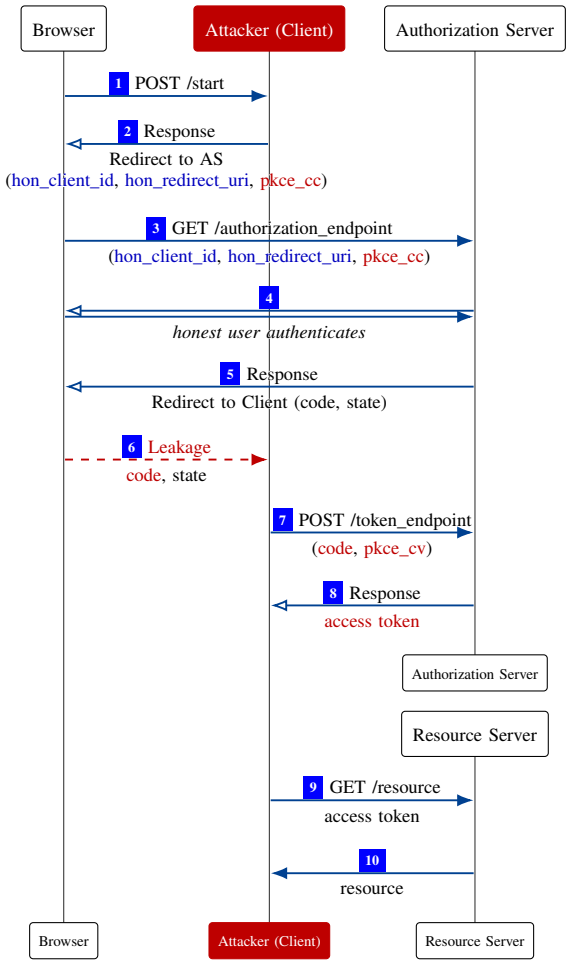

Figure 6. PKCE Chosen Challenge Attack

example, be protected by user passwords). Our fix is already included in the model and our analysis (Section V) shows that it works.

D. Authorization Request Leak Attacks

As explained in Section III-C2, the PKCE challenge is created such that PKCE is supposed to work even if the authorization request leaks (see also Section II-B).

However, if a leak of the authorization request occurs not only the PKCE challenge leaks to the attacker but also the state value, since both values are contained in the authorization request. Our attack shows that an attacker who knows the state value can circumvent the CSRF protection the state value was supposed to provide. As a result of the attack, the honest user is logged in under the identity of the attacker and uses the resources of the attacker, which breaks session integrity. The details of this attack are presented in Appendix A.

This is a well-known class of attacks for plain OAuth flows [44], but it is important to highlight that the protections designed into the FAPI do not sufficiently protect most flows against such attacks, even though PKCE explicitly foresees the attack vector.

To prevent this attack, one essentially has to prevent CSRF forgery in this context. However, this is non-trivial because of the very strong attacker model considered by the OpenID FAPI Working Group: leaks and misconfigurations are assumed to occur at various places. As further explained in Appendix A, just assuming that the authorization request does not leak to the attacker would not fix the problem in general; one at least would have to assume that the authorization response does not leak either. Making these assumptions, however, of course contradicts the OpenID FAPI Working Group's intention, namely providing security even in the presence of very strong attackers.

Fortunately, we can prove that regular FAPI web server clients which use OAUTB are not vulnerable to this attack even in the presence of the strong attackers assumed by the OpenID FAPI Working Group and throughout this paper. More specifically, we can prove session integrity of the FAPI for such clients (and strong attackers), which in particular excludes the above attack (see Section V). For all other types of clients, our attack works, and there does not seem to be a fix which would not massively change the flows, and hence, the standards, as argued in Appendix A. In this sense, our results for session integrity appear to be the best we can obtain for the FAPI.

V. Formal Security Analysis

In this section, we present our formal analysis of the FAPI. We start by very briefly recalling the Web Infrastructure Model (WIM), followed by a sketch of our formal model of the FAPI, which as already mentioned uses the WIM as its basic web infrastructure model. We then introduce central security properties the FAPI is supposed to satisfy, along with our main theorem stating that these properties are satisfied.

Since we cannot present the full formal details here, we provide some more details in the appendix, with full details and proofs provided in our technical report [25]. This includes the precise formalization of clients, authorization servers, and resource servers, as well as full detailed proofs.

A. The Web Infrastructure Model

The Web Infrastructure Model (WIM) was introduced by Fett, Küsters, and Schmitz in [22] (therefore also called the FKS model) and further developed in subsequent work. The appendix of [45] provides a detailed description of the model; a comparison with other models and a discussion of its scope and limitations can be found in [22]–[24]. We here only give a brief overview of the WIM following the description in [7], with some more details presented in Appendix B. As explained there, we slightly extend the WIM, among others to model OAUTB. We choose the WIM for our work because, as mentioned in the introduction, the WIM is the most comprehensive model of the web infrastructure to date.

The WIM is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. Among

others, HTTP(S) requests and responses,[12] including several headers, such as cookie, location, referer, authorization, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as modern technologies, such as web storage, web messaging (via postMessage), and referrer policies. JavaScript is modeled in an abstract way by so-called *scripts* which can be sent around and, among others, can create iframes, access other windows, and initiate XMLHttpRequests.

The WIM defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes that listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on. The WIM follows the Dolev-Yao approach (see, e.g., [46]). That is, messages are expressed as formal terms over a signature Σ which contains constants (for addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures).

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

The so-called *attacker process* records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties, browsers, and servers.

A *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes, but run in and interact with the browser. Similar to an attacker process, an *attacker script* can (non-deterministically) perform every action a script can possibly perform within a browser.

A *system* is a set of processes. A *configuration* of a system is a tuple of the form (S,E,N) where S maps every process of the system to its state, E is the pool of waiting events, and N is a sequence of unused nonces. In what follows, s_0^p denotes the initial state of process p. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step.

[12]We note that the WIM models TLS at a high level of abstraction such that messages are exchanged in a secure way.

A *web system* formalizes the web infrastructure and web applications. It contains a system consisting of honest and attacker processes. Honest processes can be web browsers, web servers, or DNS servers. Attackers can be either *web attackers* (who can listen to and send messages from their own addresses only) or *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A web system further contains a set of scripts (comprising honest scripts and the attacker script).

In our FAPI model, we need to specify only the behavior of servers and scripts. These are not defined by the WIM since they depend on the specific application, unless they become corrupted, in which case they behave like attacker processes and attacker scripts. We assume the presence of a strong network attacker which also controls all DNS servers (but we assume a working PKI).

B. Sketch of the Formal FAPI Model

A **FAPI web system (with a network attacker)**, denoted by $\mathcal{FAPI}$, is a web system (as explained in Section V-A) and can contain an unbounded finite number of clients, authorization servers, resource servers, browsers, and a network attacker. Note that a network attacker is the most powerful attacker, which subsumes all other attackers. Except for the attacker, all processes are initially honest and can become (dynamically) corrupted by the attacker at any time.

In a FAPI web system, clients, authorization servers, and resource servers act according to the specification of the FAPI presented in Section III. (As mentioned in Section V-A, the behavior of browsers is fixed by the standards. Their modeling is independent of the FAPI and already contained in the WIM.) Our models for clients and servers follow the latest recommendations regarding the security of OAuth 2.0 [42] to mitigate all previously known attacks. The model also contains the fixes pointed out in Section IV, as otherwise, we would not be able to prove the desired security properties (see below).

The primary goal of the FAPI is to provide a high degree of security. Its flows are intended to be secure even if information leaks to an attacker. As already outlined in Section III-C, we model this by sending the authorization response (in the case of an app client), the access token (in the case of a Read-Write flow), and the authorization request to an arbitrary (non-deterministically chosen) IP address. Furthermore, in the Read-Write profile, the token request can be sent to an arbitrary URI.

Importantly, one FAPI web system contains all possible settings in which the FAPI can run, as depicted in Figure 3, in particular, we consider all OAuth 2.0 extensions employed in the FAPI. More precisely, every client in a FAPI web system runs one of the possible configurations (i.e., it implements on one path in Figure 3). Different clients may implement different configurations. Every authorization and resource server in a FAPI web system supports all configurations at once. When interacting with a specific client, a server just chooses the configuration the client supports. In our model, the various endpoints (authorization, redirection, token), the information which client supports which FAPI configuration,

client credentials, etc. are preconfigured and contained in the initial states of the processes. How this information is acquired is out of the scope of the FAPI.

We emphasize that when proving security properties of the FAPI, we prove these properties for all FAPI web systems, where different FAPI web systems can differ in the number of clients and servers, and their preconfigured information.

Furthermore, we note that there is no notion of time in the WIM, hence, tokens do not expire. This is a safe overapproximation as it gives the attacker more power.

To give a feel for our formal FAPI model, an excerpt of the model is provided in Appendix C.

C. Security Properties and Main Theorem

In the following, we define the security properties the FAPI should fulfill, namely authorization, authentication, and session integrity. These properties have been central to also OAuth 2.0 and OpenID Connect [6], [7]. But as mentioned, the FAPI has been designed to fulfill these properties under stronger adversaries, therefore using various OAuth extensions. While our formulations of these properties are inspired by those for OAuth 2.0 and OpenID Connect, they had to be adapted and extended for the FAPI, e.g., to capture properties of resource servers, which previously have not been modeled. We also state our main theorem.

We give an overview of each security property. For the authorization property, we provide an in-depth explanation, together with the formal definition. Appendix D contains a proof sketch for the authorization property. Full details and proofs of all properties are given in our technical report [25].

1) Authorization: Informally speaking, for authorization we require that an attacker cannot access resources belonging to an honest user (browser). A bit more precise, we require that in all runs ρ of a FAPI web system $\mathcal{FAPI}$ if an honest resource server receives an access token that is associated with an honest client, an honest authorization server, and an identity of an honest user, then access to the corresponding resource is not provided to the attacker in any way. We highlight that this does not only mean that the attacker cannot access the resource directly at the resource server, but also that the attacker cannot access the resource through a client.

In order to formalize this property, we first need to define what it means for an access token to be associated with a client, an AS, and a user identity (see below for an explanation of this definition).

Definition 1 (Access Token associated with C, AS and ID). Let c be a client with client id *clientId* issued to c by the authorization server *as*, and let $id \in \mathsf{ID}^{as}$, where ID^{as} denotes the set of identities governed by as. We say that an *access token t is associated with c, as and id* in state S of the configuration (S,E,N) of a run ρ of a FAPI web system, if there is a sequence $s \in S(as).\texttt{accessTokens}$ such that $s \equiv \langle id, clientId, t, \texttt{r}\rangle$, $s \equiv \langle \texttt{MTLS}, id, clientId, t, key, \texttt{rw}\rangle$ or $s \equiv \langle \texttt{OAUTB}, id, clientId, t, key', \texttt{rw}\rangle$, for some *key* and *key*$'$.

Intuitively, an access token t is associated with a client c, authorization server *as*, and user identity *id*, if t was created by the authorization server *as* and if the AS has created t for the client c and the identity *id*.

More precisely, the access token is exchanged for an authorization code (at the token endpoint of the AS), which is issued for a specific client. This is also the client to which the access token is associated with. The user identity with which the access token is associated is the user identity that authenticated at the AS (i.e., logged in at the website of the AS). In the model, the AS associates the access token with the client identifier and user identity by storing a sequence containing the identity, the client identifier and the access token (i.e., $\langle id, clientId, t, \texttt{r}\rangle$, $\langle \texttt{MTLS}, id, clientId, t, key, \texttt{rw}\rangle$ or $\langle \texttt{OAUTB}, id, clientId, t, key', \texttt{rw}\rangle$). Furthermore, the last entry of the sequence indicates if the client is using the Read-Only or the Read-Write flow. In addition to this, for the Read-Write flow, the AS stores whether the access token is bound via mTLS or OAUTB (along with the corresponding key with which the access token is associated).

We can now define authorization formally, again the explanation of this definition follows below.

Definition 2 (Authorization Property). We say that the FAPI web system with a network attacker $\mathcal{FAPI}$ *is secure w.r.t. authorization* iff for every run ρ of $\mathcal{FAPI}$, every configuration (S,E,N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S with $s_0^{as}.\texttt{resource\_servers}$ being domains of honest resource servers used by *as*, every identity $id \in \mathsf{ID}^{as}$ for which the corresponding browser, say b, is honest in S, every client $c \in \mathsf{C}$ that is honest in S with client id *clientId* issued to c by *as*, every resource server $rs \in \mathsf{RS}$ that is honest in S such that $id \in s_0^{rs}.\texttt{ids}$ (set of IDs handled by *rs*), $s_0^{rs}.\texttt{authServ} \in \mathsf{dom}(as)$ (set of domains controlled by *as*) and with $dom_{rs} \in s_0^{as}.\texttt{resource\_servers}$ (with $dom_{rs} \in \mathsf{dom}(rs)$), every access token t associated with c, *as* and *id* and every resource access nonce $r \in s_0^{rs}.\texttt{rNonce}[id] \cup s_0^{rs}.\texttt{wNonce}[id]$ it holds true that:

If r is contained in a response to a request m sent to *rs* with $t \equiv m.header[\texttt{Authorization}]$, then r is not derivable from the attackers knowledge in S.

As outlined above, the authorization property states that if the honest resource server receives an access token associated with a client identifier, authorization server, and user identifier, then the corresponding resource access is not given to the attacker. Access to resources is modeled by nonces called *resource access nonces*. For each user identity, there is one set of nonces representing read access, and another set representing write access. In our model of the FAPI, when a resource server receives an access token associated with a user from a client, the resource server returns to the client one of the resource access nonces of the user, which in turn the client forwards to the user's browser. The above security property requires that the attacker does not obtain such a resource access nonce (under the assumptions state in the property). This captures that there should be no direct or indirect way for the attacker

to access the corresponding resource. In particular, the attacker should not be able to use a client such that he can access the resource through the client.

For the authorization property to be meaningful, we require that the involved participants are honest. For example, we require that the authorization server at which the identity is registered is honest. If this is not the case (i.e., the attacker controls the AS), then the attacker could trivially access resources. The same holds true for the client for which the access token is issued: If the user chooses a client that is controlled by the attacker, then the attacker can trivially access the resource (as the user authorized the attacker client to do so). In our model of the FAPI, the client (non-deterministically) chooses a resource server that the authorization server supports (this can be different for each login flow). As in the Read-Only flow, the access token would trivially leak to the attacker if the resource server is controlled by the attacker, we require that the resource servers that the AS supports are honest. Furthermore, in the WIM, the behavior of the user is subsumed in the browser model, therefore, we require that the browser that is responsible for the user identity that is involved in the flow should be honest. Otherwise, the attacker could trivially obtain the credentials of the user.

2) Authentication: Informally speaking, the authentication property states that an attacker should not be able to log in at a client under the identity of an honest user. More precisely, we require that in all runs ρ of a FAPI web system $\mathcal{FAPI}$ if in ρ a client considers an honest user (browser) whose ID is governed by an honest AS to be logged in (indicated by a service token which a user can use at the client), then the adversary cannot obtain the service token.

3) Session Integrity: There are two session integrity properties that capture that an honest user should not be logged in under the identity of the attacker and should not use resources of the attacker. As shown in Section IV-D, session integrity is not given for all configurations available in the FAPI. Therefore, we show a limited session integrity property that captures session integrity for web server clients that use OAUTB.

Nonetheless, our session integrity property here is stronger than those used in [6], [7] in the sense that we define (and prove) session integrity not only in the presence of web attackers, but also for the much stronger network attacker. (This is enabled by using the *__Secure-* prefix for cookies.)

Session Integrity for Authorization for Web Server Clients with OAUTB: Intuitively, this property states that for all runs ρ of a FAPI web system $\mathcal{FAPI}$, if an honest user can access the resource of some identity u (registered at AS as) through the honest web server client c, where c uses OAUTB as the holder of key mechanism, then (1) the user started the flow at c and (2) if as is honest, the user authenticated at the as using the identity u.

Session Integrity for Authentication for Web Server Clients with OAUTB: Similar to the previous property, this property states that for all runs ρ of a FAPI web system $\mathcal{FAPI}$, if an honest user is logged in at the honest client c under some identity u (registered at AS as), with c being a web server client using OAUTB as the holder of key mechanism, then (1) the user started the flow at c and (2) if as is honest, the user authenticated at the as using the identity u.

By *Session Integrity for Web Server Clients with OAUTB* we denote the conjunction of both properties.

Now, our main theorem says that these properties are satisfied for all FAPI web systems.

Theorem 1. Let $\mathcal{FAPI}$ be a FAPI web system with a network attacker. Then, $\mathcal{FAPI}$ is secure w.r.t. authorization and authentication. Furthermore, $\mathcal{FAPI}$ is secure w.r.t. session integrity for web server clients with OAUTB.

We emphasize that the FAPI web systems take into account the strong attacker the FAPI is supposed to withstand as explained in Section III-C. Such attackers immediately break plain OAuth 2.0 and OpenID Connect. This, together with the various OAuth 2.0 security extensions which the FAPI uses and combines in different ways, and which have not formally been analyzed before, makes the proof challenging.

VI. Conclusion

In this paper, we performed the first formal analysis of an Open Banking API, namely the OpenID Financial-grade API. Based on the Web Infrastructure Model, we built a comprehensive model comprising all protocol participants (clients, authorization servers, and resource servers) and all important options employed in the FAPI: clients can be app clients or web server clients and can make use of either the Read-Only or the Read-Write profile. We modeled all specified methods for authenticating at the authorization server and both mechanisms for binding tokens to the client, namely, Mutual TLS and OAuth 2.0 Token Binding. We also modeled PKCE, JWS Client Assertions, and the JWT Secured Authorization Response Mode (JARM).

Based on this model, we then defined precise security properties for the FAPI, namely authorization, authentication, and session integrity. While trying to prove these properties for the FAPI, we found several vulnerabilities that can enable an attacker to access protected resources belonging to an honest user or perform attacks on session integrity. We developed fixes against these attacks and formally verified the security of the (fixed) OpenID FAPI.

This is an important result since the FAPI enjoys wide industry support and is a promising candidate for the future lead in open banking APIs. Financial-grade applications entail very high security requirements that make a thorough formal security analysis, as performed in this paper, indispensable.

Our work also constitutes the very first analysis of various OAuth security extensions, namely PKCE, OAuth mTLS, OAUTB, JARM, and JWS Client Assertions.

Acknowledgements. This work was partially supported by *Deutsche Forschungsgemeinschaft* (DFG) through Grant KU 1434/10-2.

References

[1] "Blurred Lines: How FinTech Is Shaping Financial Services," 2016. PwC Global Fin-Tech Report.

[2] European Union, "DIRECTIVE (EU) 2015/2366 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL." https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32015L2366&from=DE.

[3] S. T. Mnuchin and C. S. Phillips, "A Financial System That Creates Economic Opportunities – Nonbank Financials, Fintech, and Innovation." https://home.treasury.gov/sites/default/files/2018-08/A-Financial-System-that-Creates-Economic-Opportunities---Nonbank-Financials-Fintech-and-Innovation_0.pdf.

[4] M. Leszcz, "The UK Open Banking Implementation Entity Adopts the OpenID Foundation Financial-Grade API (FAPI) Specification & Certification Program." https://openid.net/2018/07/12/the-uk-open-banking-implementation-entity-adopts-the-openid-foundation-financial-grade-api-fapi-specification-certification-program/.

[5] OpenID Financial-grade API Working Group, "OpenID Foundation Financial-grade API (FAPI)." Aug. 23, 2018. https://bitbucket.org/openid/fapi/src/ceb0f829bc532e9c540efaa94f6f96d007371ca2/.

[6] D. Fett, R. Küsters, and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," in *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, pp. 1204–1215, ACM, 2016.

[7] D. Fett, R. Küsters, and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines," in *IEEE 30th Computer Security Foundations Symposium (CSF 2017)*, IEEE Computer Society, 2017.

[8] A. Kumar, "Using automated model analysis for reasoning about security of web protocols," in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC'12*, Association for Computing Machinery (ACM), 2012.

[9] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering Concrete Attacks on Website Authorization by Formal Analysis," in *25th IEEE Computer Security Foundations Symposium, CSF 2012* (S. Chong, ed.), pp. 247–262, IEEE Computer Society, 2012.

[10] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Discovering Concrete Attacks on Website Authorization by Formal Analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

[11] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pp. 399–314, USENIX Association, 2013.

[12] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal Verification of OAuth 2.0 Using Alloy Framework," in *CSNT '11 Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*, pp. 655–659, Proceedings of the International Conference on Communication Systems and Network Technologies, 2011.

[13] S. Chari, C. S. Jutla, and A. Roy, "Universally Composable Security Analysis of OAuth v2.0," *IACR Cryptology ePrint Archive*, vol. 2011, p. 526, 2011.

[14] S.-T. Sun and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems," in *ACM Conference on Computer and Communications Security, CCS'12* (T. Yu, G. Danezis, and V. D. Gligor, eds.), pp. 378–390, ACM, 2012.

[15] W. Li and C. J. Mitchell, "Security issues in OAuth 2.0 SSO implementations," in *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, pp. 529–541, 2014.

[16] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu, "Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pp. 651–662, ACM, 2016.

[17] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. R. B. Butler, "More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, vol. 9148 of *Lecture Notes in Computer Science*, pp. 239–260, Springer, 2015.

[18] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "OAuth Demystified for Mobile Application Developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pp. 892–903, 2014.

[19] M. Shehab and F. Mohsen, "Towards Enhancing the Security of OAuth Implementations in Smart Phones," in *2014 IEEE International Conference on Mobile Services*, Institute of Electrical & Electronics Engineers (IEEE), 6 2014.

[20] W. Li and C. J. Mitchell, "Analysing the Security of Google's Implementation of OpenID Connect," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, vol. 9721, pp. 357–376, 2016.

[21] V. Mladenov, C. Mainka, J. Krautwald, F. Feldmann, and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect," *CoRR*, vol. abs/1508.04324v2, 2016.

[22] D. Fett, R. Küsters, and G. Schmitz, "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System," in *35th IEEE Symposium on Security and Privacy (S&P 2014)*, pp. 673–688, IEEE Computer Society, 2014.

[23] D. Fett, R. Küsters, and G. Schmitz, "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pp. 1358–1369, ACM, 2015.

[24] D. Fett, R. Küsters, and G. Schmitz, "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web," in *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, vol. 9326 of *Lecture Notes in Computer Science*, pp. 43–65, Springer, 2015.

[25] D. Fett, P. Hosseyni, and R. Küsters, "An Extensive Formal Security Analysis of the OpenID Financial-grade API," Tech. Rep. arXiv:1901.11520, arXiv, 2019. Available at http://arxiv.org/abs/1901.11520.

[26] D. Hardt (ed.), "RFC6749 – The OAuth 2.0 Authorization Framework." IETF. Oct. 2012. https://tools.ietf.org/html/rfc6749.

[27] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1." OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-core-1_0.html.

[28] J. Richer (ed.), "RFC7662 – OAuth 2.0 Token Introspection." IETF. Oct. 2015. https://tools.ietf.org/html/rfc7662.

[29] N. Sakimura (Ed.), J. Bradley, and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients." RFC 7636 (Proposed Standard), Sept. 2015.

[30] W. Denniss and J. Bradley, "OAuth 2.0 for Native Apps," *RFC*, vol. 8252, pp. 1–21, 2017.

[31] M. Jones, J. Bradley, and N. Sakimura, "RFC7519 – JSON Web Token (JWT)." IETF. May 2015. https://tools.ietf.org/html/rfc7519.

[32] M. Jones, J. Bradley, and N. Sakimura, "RFC7515 – JSON Web Signature (JWS)." IETF. May 2015. https://tools.ietf.org/html/rfc7515.

[33] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt, "OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens," Internet-Draft draft-ietf-oauth-mtls-09, Internet Engineering Task Force, June 2018. Work in Progress.

[34] M. Jones, B. Campbell, J. Bradley, and W. Denniss, "OAuth 2.0 Token Binding - draft-ietf-oauth-token-binding-07." https://www.ietf.org/id/draft-ietf-oauth-token-binding-07.txt.

[35] A. Popov, M. Nystrom, D. Balfanz, A. Langley, and J. Hodges, "The Token Binding Protocol Version 1.0." RFC 8471, Oct. 2018.

[36] A. Popov, M. Nystrom, D. Balfanz, and A. Langley, "Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation." RFC 8472, Oct. 2018.

[37] A. Popov, M. Nystrom, D. Balfanz, A. Langley, N. Harper, and J. Hodges, "Token Binding over HTTP." RFC 8473, Oct. 2018.

[38] E. Rescorla, "Keying Material Exporters for Transport Layer Security (TLS)." RFC 5705, Mar. 2010.

[39] A. Popov, M. Nystrom, D. Balfanz, A. Langley, N. Harper, and J. Hodges, "Token Binding over HTTP," internet-draft, Internet Engineering Task Force, June 2018. Work in Progress.

[40] T. Lodderstedt (ed.), "JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)." Aug. 23, 2018. https://bitbucket.org/openid/fapi/src/ceb0f829bc532e9c540efaa94f6f96d007371ca2/Financial_API_JWT_Secured_Authorization_Response_Mode.md.

[41] OpenID Financial-grade API Working Group, "Financial API - Part 1: Read-Only API Security Profile." Aug. 23, 2018. https://bitbucket.

org/openid/fapi/src/ceb0f829bc532e9c540efaa94f6f96d007371ca2/Financial_API_WD_001.md.
[42] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "OAuth 2.0 Security Best Current Practice," 10 2018. https://tools.ietf.org/html/draft-ietf-oauth-security-topics.
[43] OpenID Financial-grade API Working Group, "Financial API - Part 2: Read and Write API Security Profile." Aug. 23, 2018. https://bitbucket.org/openid/fapi/src/ceb0f829bc532e9c540efaa94f6f96d007371ca2/Financial_API_WD_002.md.
[44] T. Lodderstedt (ed.), M. McGloin, and P. Hunt, "RFC6819 – OAuth 2.0 Threat Model and Security Considerations." IETF. Jan. 2013. https://tools.ietf.org/html/rfc6819.
[45] D. Fett, R. Küsters, and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Analysis and Security Guidelines," Tech. Rep. arXiv:1704.08539, arXiv, 2017. Available at http://arxiv.org/abs/1704.08539.
[46] M. Abadi and C. Fournet, "Mobile Values, New Names, and Secure Communication," in *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001)*, pp. 104–115, ACM Press, 2001.
[47] A. Barth and M. West, "Cookies: HTTP State Management Mechanism." https://httpwg.org/http-extensions/rfc6265bis.html.

Appendix A
Authorization Request Leak Attack – Details

We here provide further details about the authorization request leak attack, which was only sketched in Section IV-D.

A concrete instantiation of this attack is shown in Figure 7, where the scenario is based on the Read-Only flow of a public client. As explained below, similar attacks also work for all other configurations of the FAPI (except for web server clients which use OAUTB, for which, as mentioned, we show that they are not susceptible in Section V).

In the Authorization Request Leak Attack, the client sends the authorization request to the browser in Step 2, where it leaks to the attacker in Step 3. From here on, the attacker behaves as the browser and logs himself in (Step 5), hence, the authorization code received in Step 6 is associated with the identity of the attacker.

The state value used in the authorization request aims at preventing Cross-Site Request Forgery (CSRF) attacks. However, as the state value leaks, this protection does not work. For showing that this is the case, we assume that a CSRF attack happens. If, for example, the user is visiting a website that is controlled by the attacker, then the attacker can send, from the browser of the user, a request to the AS containing the code and the state value (Step 8). As the state received by the client is the same that it included in the authorization request, the client continues the flow and uses the code to retrieve an access token in Steps 9 and 10.

This access token is associated with the attacker, which means that the honest user is accessing resources belonging to the attacker.

As a result, the honest user can be logged in under the identity of the attacker if the authorization server returns an id token. In the case of the Read-Write flow, the honest user can modify resources of the attacker: for example, she might upload personal documents to the account of the attacker.

As noted above, this attack might happen for all configurations, except for the Read-Write flow when the client is a web server client using OAUTB (see Figure 3).

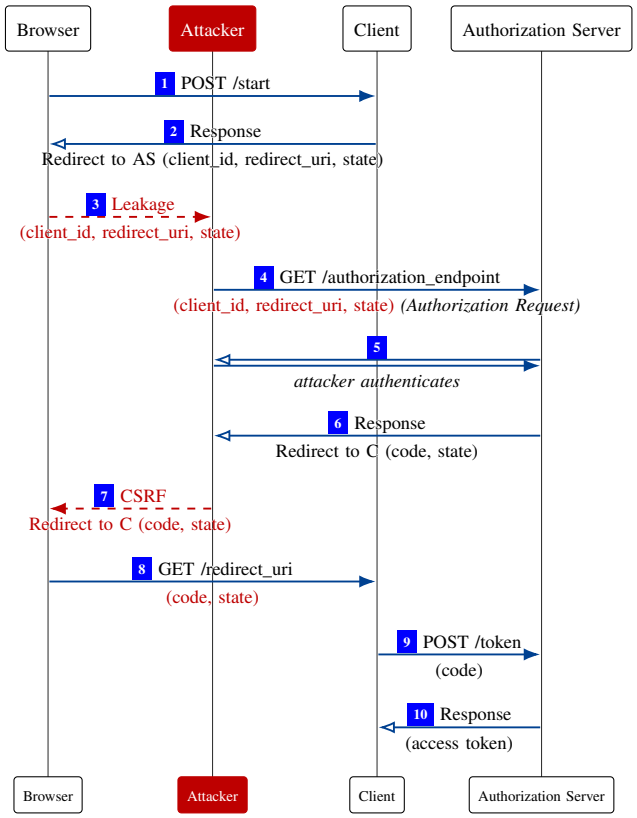

Figure 7. Leakage of Authorization Request Attack

In all other configurations, this attack can happen as the attacker can behave exactly like the browser of the honest user, i.e., after receiving the authorization request, the attacker can send this request to the AS, log in under his own identity, and would then receive a response that the client accepts. The only flow in which this is different is the Read-Write flow where the client is a web server and uses OAUTB, as here, the browser (and therefore, also the attacker) needs to prove possession of a key pair (i.e., the key pair used for the client). As the attacker cannot prove possession of the private key of the key pair which the browser uses for the client, the AS would then stop the flow. (In the other flows, the AS does not check if the response was sent by the browser that logged in the user.)

If we say that the FAPI is not required to be secure if the authorization request leaks (i.e., if we remove the assumption that the authorization request leaks), then the flow is still not secure, as the authorization response might still leak to the attacker (see Section III-C1), which also contains the state value. More precisely, the authorization response might leak in the case of app clients due to the operating system sending the response to the attacker app (for details, see Section II-B). After receiving the authorization response, the attacker app knows the state value and can start a new flow using this value. The attacker can then continue from Step 3 (Figure 7), and when receiving the authorization response (which is a URI

containing the OAuth parameters), he could, using his own app that runs on the device of the victim, call the legitimate client app with this URI (i.e., with the code that is associated with the identity of the attacker and the state value with which the client started the flow). The effect of this is that the legitimate app, at which the honest user started the flow, would continue the flow using an authorization code associated with the attacker. Therefore, the honest user would either be logged in with the identity of the attacker or use the resources of the attacker.

We note that even encrypting the state value contained in the authorization request does not solve the problem, as the attacker is using the whole authorization request. (Strictly speaking, he acts as the browser of the honest user).

APPENDIX B
THE WIM: SOME BACKGROUND

We here provide more details about the Web Infrastructure Model.

a) Signature and Messages: As mentioned, the WIM follows the Dolev-Yao approach where messages are expressed as formal terms over a signature Σ. For example, in the WIM an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP request for the URI http://ex.com/show?p=1 is represented as $r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{ex.com}, /\mathtt{show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$ where the body and the list of request headers is empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k'\rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

b) Scripts: A *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

c) Running a system: As mentioned, a run of a system is a sequence of configurations. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example, $Q = (S, E, N) \rightarrow (S', E', N')$ to denote the transition from the configuration (S, E, N) to the configuration (S', E', N'), where S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces.

d) Web Browsers: An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the web browser. User credentials are stored in the initial state of the browser and are given to selected web pages when needed. Besides user credentials, the state of a web browser contains (among others) a tree of windows and documents, cookies, and web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded web page and contains (among others) a script and a list of subwindows (modeling iframes). The script, when triggered by the browser, is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and web storage data, and create iframes. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant web standards.

A browser can output messages on the network of different types, namely DNS and HTTP(S) (including XMLHttpRequests), and it processes the responses. Several HTTP(S) headers are modeled, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a so-called trigger message upon which the browser non-deterministically chooses an action, for instance, to trigger a script in some document. The script now outputs a command, as described above, which is then further processed by the browser. Browsers can also become corrupted, i.e., be taken over by web and network attackers. Once corrupted, a browser behaves like an attacker process.

As detailed in our technical report [25], we extended the browser model of the WIM slightly in order to incorporate OAUTB in the browser model. We furthermore added the behavior of the *__Secure-* prefix of cookies to the model, which specifies that such cookies shall only be accepted when they are transmitted over secure channels [47]. Note that for the FAPI, mTLS is only needed between clients and servers. Therefore, mTLS has been modeled on top of the WIM, i.e., as part of the modeling of FAPI clients and servers. The servers we modeled for the FAPI of course also support OAUTB.

APPENDIX C
EXCERPT OF CLIENT MODEL

In this section, we provide a brief excerpt of the client model in order to give an impression of the formal model. See our technical report [25] for the full formal model of the FAPI.

The excerpt given in Algorithm 1 shows how the client prepares and sends the token request to the authorization server, i.e., the part in which the client sends the authorization code in exchange for an access token (and depending on the flow, also an id token).

This function is called by the client. The first two inputs are the session identifier of the session (i.e., the session of the resource owner at the client) and the authorization code that the client wants to send to the AS. The value *responseValue* contains information related to mTLS or OAUTB (if used for the current flow). The last input is the current state of the client.

In Lines 5 to 8, the client chooses either the token endpoint of the AS or some URL that was chosen non-deterministically. This models the assumption shown in Section III-C4, which requires the Read-Write profile of the FAPI to be secure even if the token endpoint is misconfigured.

Starting from Line 15, the function chooses the parameters of the request that depend on the flow and configuration (see Figure 3).

If the client uses the Read-Only profile, the token request always contains the PKCE verifier (Line 15). For a confidential client (which means that the client has to authenticate at the token endpoint), the client either authenticates using JWS Client Assertions (Line 20, see also Section II-C), or with mTLS (Line 26; for details on our model of mTLS refer to our technical report [25]).

If the client uses the Read-Write profile, the client uses either mTLS (again Line 26) or OAUTB (Line 32; for details on our model of OAUTB refer to our technical report [25]).

Appendix D
Proof Sketch of Theorem 1, Authorization

We here provide a proof sketch of Theorem 1 that is concerned with the authorization property. The complete formal proof of this theorem is given in the technical report [25].

For proving the authorization property, we show that when a participant provides access to a resource, i.e., by sending a resource access nonce, this access is not provided to the attacker:

a) Resource server does not provide the attacker access to resources: We show that the resource server does not provide the attacker access to resources of an honest user.

In case of the Read-Only flow, we show that an access token associated with an honest client, an honest authorization server, and an honest identity does not leak to the attacker, and therefore, the attacker cannot obtain access to resources.

In case of the Read-Write flow, such an access token might leak to the attacker, but this token cannot be used by the attacker at the resource server due to Token Binding, either via OAUTB or mTLS.

b) Web server client does not provide the attacker access to resources: App clients are only usable via the device they are running on, i.e., they are not usable over the network (by which we mean that if, for example, the user wants to view one of her documents with an app client, she does this directly using the device). Therefore, we only look at the case of web server clients, as such a client can be used over the network, e.g., by the browser of the end-user or by the attacker.

In the following, we show that honest web server clients do not provide the attacker access to resources belonging to an

Algorithm 1: Client R^c – Request to token endpoint.

```
1: function SEND_TOKEN_REQUEST(sessionId, code,
   responseValue, s')
2:   let session := s'.sessions[sessionId]
3:   let identity := session[identity]
4:   let issuer := s'.issuerCache[identity]
5:   if session[misconfiguredTEp] ≡ ⊤ then
6:     let url := session[token_ep]
7:   else
8:     let url := s'.oidcConfigCache[issuer][token_ep]
9:   let credentials := s'.clientCredentialsCache[issuer]
10:  let clientId := credentials[client_id]
11:  let clientType := credentials[client_type]
12:  let profile := credentials[profile]
13:  let isApp := credentials[is_app]
14:  let body := [grant_type:authorization_code, code:code,
       ↪ redirect_uri:session[redirect_uri],
       ↪ client_id:clientId]
15:  if profile ≡ r then
16:    let body[pkce_verifier] := session[pkce_verifier]
17:  if profile ≡ r ∧ clientType ≡ pub then
18:    let message := ⟨HTTPReq, v2, POST, url.domain, url.path,
   url.parameters, ⊥, body⟩
19:    call HTTPS_SIMPLE_SEND([responseTo:TOKEN,
         ↪ session:sessionId], message, s')
20:  else if profile ≡ r ∧ clientType ≡ conf_JWS then
21:    let clientSecret := credentials[client_secret]
22:    let jwt := [iss:clientId, aud:url.domain]
23:    let body[assertion] := mac(jwt, clientSecret)
24:    let message := ⟨HTTPReq, v2, POST, url.domain, url.path,
   url.parameters, ⊥, body⟩
25:    call HTTPS_SIMPLE_SEND([responseTo:TOKEN,
         ↪ session:sessionId], message, s')
26:  else if clientType ≡ conf_MTLS then    → both profiles
27:    if responseValue[type] ≢ MTLS then
28:      stop
29:    let body[TLS_AuthN] := responseValue[mtls_nonce]
30:    let message := ⟨HTTPReq, v2, POST, url.domain, url.path,
   url.parameters, ⊥, body⟩
31:    call HTTPS_SIMPLE_SEND([responseTo:TOKEN,
         ↪ session:sessionId], message, s')
32:  else    → rw with OAUTB
33:    if responseValue[type] ≢ OAUTB then
34:      stop
35:    let ekm := responseValue[ekm]
36:    let TB_AS := s'.TBindings[url.host]    → priv. key
37:    let TB_RS := s'.TBindings[session[RS]]    → priv. key
38:    let TB_Msg_prov := [id:pub(TB_AS),
         ↪ sig:sig(ekm, TB_AS)]
39:    let TB_Msg_ref := [id:pub(TB_RS), sig:sig(ekm, TB_RS)]
40:    let headers := [Sec-Token-Binding:[prov:TB_Msg_prov,
         ↪ ref:TB_Msg_ref]]
41:    if clientType ≡ conf_OAUTB then    → client authentication
42:      let clientSecret := credentials[client_secret]
43:      let jwt := [iss:clientId, aud:url.domain,]
44:      let body[assertion] := mac(jwt, clientSecret)
45:    if isApp ≡ ⊥ then    → W.S. client: TBID used by browser
46:      let body[pkce_verifier] := session[browserTBID]
47:    let message := ⟨HTTPReq, v2, POST, url.domain, url.path,
   url.parameters, headers, body⟩
48:    call HTTPS_SIMPLE_SEND([responseTo:TOKEN,
         ↪ session:sessionId], message, s')
```

honest identity. We show this for all possible configurations that could trick the client into doing so, e.g., with a misconfigured token endpoint or with an authorization server controlled by the attacker that returns a leaked access token.

The access to the resource is provided to the sender of the redirection request. To access a resource, this means that the attacker must have sent the request to the redirection endpoint of the client.

For a Read-Only flow, the token endpoint is configured correctly. This means that the attacker must include a code in the request such that the client can exchange it for an access token. We show that such a code (associated with an honest identity and the client) does not leak to an attacker.

For a Read-Write flow, the token endpoint can be misconfigured such that it is controlled by the attacker, and we also assume that access tokens leak to the attacker (see Section III-C).

We show that a leaked access token cannot be used at the client by the attacker. If only the token endpoint is controlled by the attacker, he must include an id token (when using the OIDC Hybrid flow, see below for the Authorization Code flow with JARM) in the token response such that it contains the hash of the access token and be signed by the honest authorization server (the hash of the access token was not included in the original draft and was included by us as a mitigation in Section IV-B). However, such an id token does not leak to the attacker, which prevents the use of leaked access tokens at misconfigured token endpoints. For the Authorization Code flow with JARM, the attacker would need a response JWS. As in the case of the Hybrid flow, we show that the response JWS needed by the client for accessing resources of an honest identity does not leak.

A leaked access token can also be used by the attacker if the client chooses an authorization server under the control of the attacker. Here, the id tokens are created by the attacker and accepted by the client. For preventing the use of this access token, the client includes the issuer of the second id token (or of the response JWS defined by JARM) in the request to the resource server, as detailed in Section IV-A. As each resource server has one preconfigured authorization server, the resource server does not provide access to a resource in this case.

The only remaining case is that the attacker includes a code associated with the honest user in the request to the redirection endpoint of the client. For the Hybrid flow, both id tokens contained in the authorization response and in the token response are required to have the same subject attribute and the same issuer value, which means that they are both signed by the authorization server. However, such an id token does not leak to the attacker, which means that the client will stop the flow when receiving the second id token contained in the token response. When using JARM, this would require the attacker to send a response JWS signed by the authorization server that contains the code that belongs to an honest client and an honest user identity. In the technical report [25], we show that such a response JWS does not leak to the attacker.