RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Analysis of the Financial-Grade API (FAPI)

Johanna Schenkel

Bachelor's Thesis – May 9, 2022.
Chair for Network and Data Security.

Supervisor: Prof. Dr. Jörg Schwenk
Advisor: Dr.-Ing. Vladislav Mladenov and M.Sc. Louis Jannett
Advisor: Dr.-Ing. Christian Mainka (Hackmanit GmbH)

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

The *financial-grade API (FAPI)* offers secure and interoperable profiles for OAuth and OpenID Connect. It was created to fit the high-security standard needed to deal with sensitive data such as banking details. The profiles employ existing security measures of the OpenID Connect and OAuth standardizations, such as *JWT-Secured Authorization Request (JAR)*, *Pushed Authorization Request (PAR)* or *Mutual TLS (MTLS)*. With concepts as sender-constrained access token, signed requests and secure client authentication, the FAPI aims to protect APIs with high risks. Interoperability is achieved by limiting possible configurations and a certification process offered by the *OpenID Foundation*. Thus, the FAPI profiles ensure compatibility between FAPI certified clients and authorization servers.

This bachelor's thesis aims to evaluate the security promised by the FAPI. Therefore, we outlined the security extensions employed by the FAPI. We explained the FAPI profiles and highlighted their differences. Additionally, we compared the security features used by the FAPI to features used by OAuth and OpenID Connect. Then, we set up a FAPI-certified client and authorization server and created a testing tool. The tool verifies the functionality of the setup and enables further manual testing in the environment. It relies on the *mitmproxy* addon framework. Moreover, we created a security catalog, including known attacks on OAuth and OpenID Connect and the FAPI's countermeasures.

In summary, we extracted the variety of security measures the FAPI employs. We showed the security improvements the FAPI provides. These include interoperability, enforcement of security best practices, and security extensions. We provide comparison tables of features and a security catalog of know attacks and the FAPI's countermeasures. These may serve as a reference for future work. We created a testing tool for a FAPI-certified implementation. The implementation can easily be extended for further testing.

Contents

1 Introduction

This chapter introduces the bachelor's thesis "Analysis of the Financial-grade API (FAPI)". First, the motivation is presented, highlighting the need for a secure OAuth and OpenID Connect (OIDC) profile in a high-risk environment, such as FinTec. Then, related work, covering the financial-grade API (FAPI), is presented. Finally, the contribution of this thesis is emphasized, and its organization is outlined.

1.1 Motivation

When the *Payment Service Directive 2 (PSD2)* [67] was published by the European Parliament in 2015, about 6,000 banks were required to become API providers by the end of 2017 [56]. With *Open Banking* [21], end-users should gain the freedom to securely share their financial data with third parties, including transaction details and payment initiation rights [21]. It further offers opportunities for more innovations in the increasingly important FinTech sector by providing Open APIs [68].

In the past, (FinTech) applications used *screenscraping* to access the user's data. With screenscraping, users are impersonated through password sharing. This method is not only inefficient, but also highly insecure [22].
The need for a more secure and efficient solution emerged, but OAuth's functionalities did not suffice since the core framework had significant security problems back in 2015 [56]. Moreover, OAuth 2.0, being an authorization framework, was too "underspecified" to deal with the need for interoperability and security [56].

For this reason, OAuth was adapted to fit the high-security standard needed to deal with sensitive data such as banking details [68]. Therefore, the *financial-grade API Working Group* was established by the Open ID Foundation [20]. The group applied existing security measures of the OpenID Connect standardization to strengthen OAuth API authorization. The result is called the financial-grade API (FAPI) and offers a safe and interoperable alternative to screenscraping [56].

1.2 Related Work

In 2019, Fett et al. [59] were the first to formally analyze and prove the security of the Financial Grade API profile. The analysis was conducted utilizing the Web Infrastructure Model (WIM) to cover a variety of possible combinations of profiles, security measures, and flows. It guarantees previously defined security characteristics. The authors discovered attack vectors regarding authentication, authorization, and session management and were also able to present suitable mitigations.

In 2021, Fett [58] introduced the FAPI 2.0, an updated, simpler, and more interoperable version of the original FAPI specification. He describes the learnings from practical applications and the developing process of the FAPI working group by outlining the FAPI 2.0 security model and core elements of the specification.

In 2018, Damabi [65] performed a security analysis of the OpenID Financial-grade API by modeling the two profiles in the FKS Web Model. The formal analysis revealed attacks regarding Proof Key for Code Exchange (PKCE) and token reuse, against which countermeasures were presented.

In 2021, Mohamed [66] extended a prototype implementation to fit the requirements of the second version of the OpenID financial-grade API. The FAPI 2.0 baseline and the FAPI 2.0 advanced profiles were introduced and included in the prototype to help developers and researchers thoroughly understand FAPI 2.0.

In 2013, Lodderstedt et al. [15] published the "OAuth 2.0 Threat Model and Security Considerations", extending the security considerations stated in RFC 6749 [13]. The authors created a threat model, defining potential threats, such as open redirectors or an access token leak in the browser history. New security considerations were introduced, such as token binding or limiting the token expiration time.

Since 2017, the Internet Engineering Task Force (IETF) is continuously working on the "OAuth 2.0 Security Best Current Practice" [10], recommending security measures and other practices, gained form practical experience. The Best Current Practice aims to extend the OAuth threat model and collect up-to-date mitigation techniques.

In addition to the *OAuth 2.0 Threat Model* and the *OAuth 2.0 Security Best Current Practices*, other standards are related to this work. They provide further information on the techniques introduced in this thesis.

- The OAuth 2.0 Authorization Framework [13]
- OAuth 2.0 Device Authorization Grant [19]
- The OAuth 2.1 Authorization Framework [**?**]

- OpenID Connect Core 1.0 [7]
- OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0 [54]
- Financial-grade API Security Profile 1.0 - Part 1: Baseline [22]
- Financial-grade API Security Profile 1.0 - Part 2: Advanced [23]
- Financial-grade API Security Profile 2.0 - Part 1: Baseline [30]
- Financial-grade API Security Profile 2.0 - Part 2: Advanced [28]
- Financial-grade API Client Initiated Backchannel Authentication Profile [53]
- Financial-grade API 2.0 Attacker Model [29]
- Financial-grade API JWT Secured Authorization Response Mode for OAuth 2.0 (JARM) [2]
- The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR) [32]
- Proof Key for Code Exchange by OAuth Public Clients [17]
- OAuth 2.0 Pushed Authorization Requests [33]
- OAuth 2.0 Rich Authorization Requests [27]
- OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens [31]
- OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP) [34]

1.3 Contribution

In this bachelor's thesis, the financial-grade API standards are systematically analyzed. The thesis aims to answer the following research questions:

1. Which concepts enable the, according to the FAPI Working group [22], high-security level of the FAPI?
2. Do the FAPI profiles provide security improvements compared to the classic OAuth and OIDC flows?
3. Which known attacks are applicable on FAPI implementations - clients and providers?

For this purpose, we elaborate the security measures presented in the FAPI standards in detail. The two FAPI standards and their profiles are explained, and their differences are highlighted. Additionally, we compare the FAPI standards to the underlying standards OAuth and OIDC regarding their security features. Then, we present well-known attacks on OAuth and OIDC, and evaluate which security measures the FAPI employs as protection. Further, we set up a FAPI-certified client and an authorization server (AS) and document their security features. We create a testing tool to ensure their functionality and to enable further testing. Finally, we evaluate the security of the FAPI by considering the results of the comparison, the security catalog, and the outcomes of working with the certified FAPI implementations.

1.4 Organization of this Thesis

This bachelor's thesis is organized into eight chapters.

Chapter 1 introduces this thesis by motivating the FAPI and presenting related work. Further, the contribution of this thesis is highlighted and the organization of it is displayed.

Chapter 2 explains fundamental OAuth and OIDC concepts that form the basis of the FAPI. The concepts of OAuth 2.0, OAuth 2.1, and OIDC are described, the attacker models are introduced, and additional OAuth and OIDC security extensions are presented.

Chapter 3 introduces the FAPI. The baseline and advance profile of each FAPI 1.0 and 2.0 are explained. Further, the FAPI Client-Initiated Backchannel Authentication (CIBA) profile is described and the FAPI attacker model is presented.

Chapter 4 displays a tabular comparison of the security features of the FAPI profiles and standards, compared to OAuth 2.0, OAuth 2.1, and OIDC. The FAPI CIBA profile is compared to the OIDC CIBA standard and the OAuth 2.0 Device Authorization Grant. Additionally, the different attacker models are compared to put the comparison in perspective.

Chapter 5 presents an exemplary setup of a FAPI-certified client and AS. The selection of the *Gluu Server 4.2* and *oxd Client API 4.2* is explained. We describe the testing environment we have created. Further, the security features of both client and AS are documented.

Chapter 6 depicts a collection of known OAuth and OIDC attacks. The security catalog describes each attack and its countermeasures. Additionally, the security measures employed by the FAPI are noted for each attack.

Chapter 7 evaluates the results of the conducted analysis. It aims to answer the research questions by evaluating the results of the comparison and the security catalog. Additionally, the impressions gained from working with real-world FAPI implementations are incorporated as well.

Chapter 8 concludes this work and suggests further work such as an extension of the testing tool.

2 Fundamentals

The financial-grade API is a security profile for OAuth, protected by OIDC security measures and additional extensions, described in this chapter. OAuth is the de-facto standard designed for API authorization. Without password sharing, the end-user allows a client application to access their data stored in another application. The authorization of the client application is delegated to an authorization server, where the end-user already has an account. OpenID Connect (OIDC) is a security protocol built on top of OAuth, which adds an authentication layer to enable third-party logins. With OIDC, Single Sign-On (SSO) scenarios can be realized.

2.1 OAuth

OAuth 2.0 is an authorization framework that was developed by the Internet Engineering Task Force (IETF) in 2012 [13]. The security standard enables an end-user to permit a third-party application access to their resources in another application. The specified ways a user grants authorization is called a *authorization grant*. For the communication between the different participants, protocol endpoints are defined.

In OAuth 2.0 grants, four different parties participate in the authorization process. The *resource owner (RO)* can be an end-user who permits the client to access their protected resources at the resource server.
The *resource server (RS)* holds protected resources and allows access if a valid access token is provided.
The *client* is an application which requests access to the users data at the authorization server. After successful authorization, the client can send resource requests to the RS, to access the user's protected resources.
In OAuth, two different types of clients are defined. The *confidential* client is able to store secrets, meaning that it can authenticate at the authorization server with its client credentials, which consist of a `client_id` and a `client_secret`. An example for a Confidential client can be a webserver. On the other hand, the *public* client cannot keep secrets and with that cannot authenticate at the authorization server since it might be a native or a single-page application. The client has one protocol endpoint, called the *Redirection Endpoint*, which is used by the authorization server for returning authorization credentials, as the authorization code, through the ROs

user agent (UA).
The *authorization server (AS)* can authenticate the RO and issue tokens. Access tokens and refresh tokens are issued to the client after receiving authorization consent by the RO. In some cases, the AS and the RS are deployed on the same machine. In a financial context, it can be a bank. The AS has different protocol endpoints (EP), used by the client. At the *Authorization Endpoint* the client obtains authorization from the end-user. For this, the RO is redirected to the endpoint, where they are authenticated and grant permission, before being redirected to the client redirection endpoint. At the *Token Endpoint*, the client receives an access token and an optional refresh token. Additionally, confidential clients need to authenticate by using their client credentials.

The AS can issue authorization codes, access tokens and refresh tokens. Authorization codes are bound to the `client_id` and `redirect_uri` and have a short lifetime. They can only be exchanged once for an access token. Access tokens are bearer tokens by default, meaning that the party which holds the access token can redeem it at the AS, even if it was not issued for them. The access token is a string representing the granted scopes and the lifetime of the token. This way, the duration of access can be limited. Thus, the RO can grant finer-grained access to their resources. When the access token becomes invalid, e.g., it expires, the client can receive a new one in exchange for the refresh token. The new access token might have fewer access rights and a shorter lifetime. Refresh tokens are strings which represent the scopes granted by the RO and are only send to the AS not to RSs.

2.1.1 OAuth 2.0 Authorization Code Grant

OAuth defines four types of grants: the *authorization code* grant, the *implicit* grant, the *resource owner password credentials* grant and the *client credentials* grant [13]. The *authorization code grant* is also employed in the FAPI.
In this grant, the AS issues an authorization code and sends it through the RO's UA to the client, who can redeem the code at the token endpoint in exchange for an access token. Not sending the access token through the RO's UA is a security benefit of the authorization code grant. The transmission of the access token through the backchannel significantly reduces the attack surface. The access token is not disclosed to the end-user or others.

The authorization code grant is displayed in Figure 2.1. In **step 1** and **step 2** in Figure 2.1, the RO demands access to their resources and starts the protocol flow. The client then sends the *authorization request* through the RO's UA to the AS's authorization endpoint in **step 3**. The authorization request includes the `response_type`, the `client_id`, the `redirect_uri` and might include a `redirect_uri` and the `scope` and `state` parameters. The `response_type` specifies the grant type, in this case its value is `code`. The client also includes its public `client_id`,

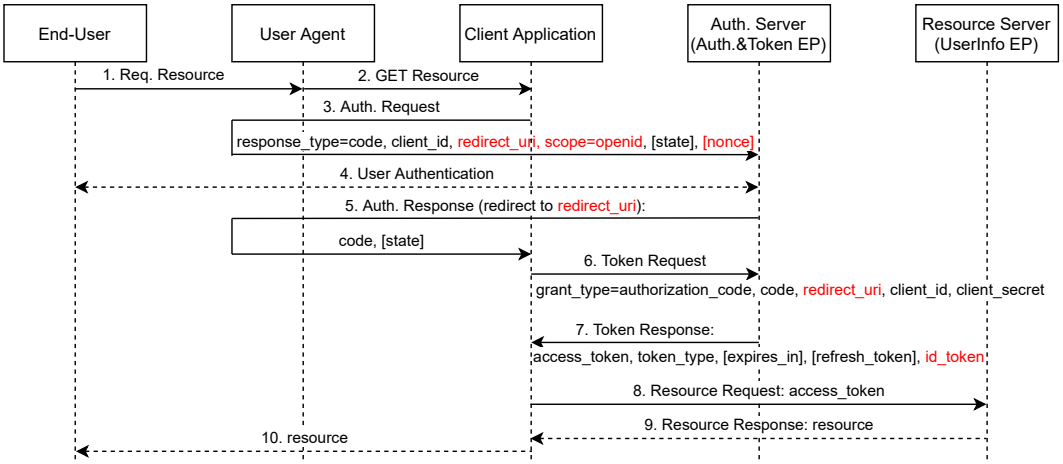


Figure 2.1: Authorization Code Grant. In red: additional parameters for the OpenID Connect Authorization Code Flow.

which has been issued previously by the AS after an initial client registration. The string identifies the client since it is unique to the AS. The optional `redirect_uri` specifies the address to which the AS shall redirect the authorization response, while the optional `scope` parameter defines a list of strings which express the access range of the access token the end-user agrees to. Moreover, the framework recommends including the `state` parameter in the request to prevent CSRF (Cross-site request forgery) attacks.

After the authorization request, the AS authenticates the end-user and obtains consent to the scope of the access token (in **step 4**). In the case of approval, the AS redirects the *authorization response* to the client's specified `redirect_uri` (**step 5**). The response contains the issued authorization `code` and the previously received `state` value.

With the *token request* in **step 6**, the client redeems the `code` at the token endpoint to receive an access token. The request further includes the `grant_type`, the `redirect_uri`, if it was sent in the authorization request, and the `client_id` if the client is public. In case of a confidential client, it has to authenticate with its client credentials, consisting of `client_id` and `client_secret`, before receiving the access token.

After a successful client authentication and validation of the authorization code as well as the `redirect_uri`, the AS sends the *token response* (**step 7**), containing an access token. The response also contains the `token_type` and an optional expiration date `expires_in`. If requested, the response might include an refresh token.

In **step 8**, the client can request the protected resource at the RS, with the respective

tokens. After the RS successfully validates the received token, it responds with the requested resource (**step 9**), which can be forwarded to the end-user (**step 10**).

2.1.2 OAuth 2.0 Device Authorization Grant

In 2019, the "OAuth 2.0 Device Authorization Grant" [19] was published as an OAuth 2.0 extension. We introduce its concepts to compare this grant to the OIDC CIBA flow and the FAPI CIBA flow (see section 4.4). It specifies an OAuth grant for client devices, such as smart TVs or printers, that can not provide a browser or receive user input. The grant is also feasible for devices that cannot receive requests. Therefore, the "Device Flow" is designed without UA-based authorization and user authentication on the device itself. Instead, the RO validates the authorization request in the UA of a separate device, such as a smartphone. Since there is no direct communication between the client device and the end-user, the device must be capable of displaying codes and URIs to the user.

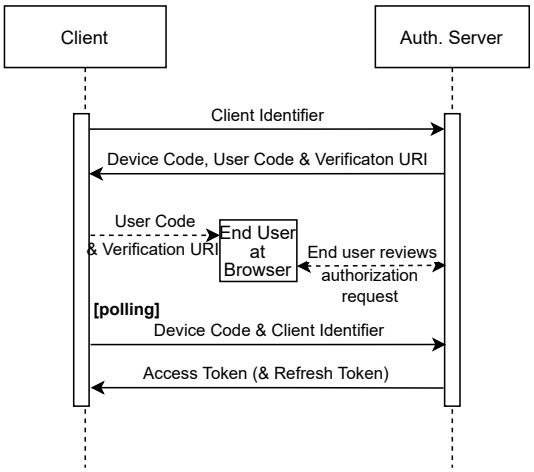


Figure 2.2: Device Authorization Flow. [19]

The device authorization flow, as displayed in Figure 2.2, begins with the client sending the authorization request, which includes its client identifier. The AS answers with the authorization response. The response includes a device code, a user code, and a verification URI. Then, the client displays the user code and the verification URI to the end-user, which then visits the URI on another device. There, the AS authenticates the RO, which enters the received user code. After the AS has checked the provided user code, the end-user validates the authorization request. During this process, the client polls the AS for an access token. Thereby, both device code and client identifier are included in the polling requests. The AS sends the token response, if the authorization request is accepted and the device code, included in the polling request, is validated. The token response includes an access token and

can contain a refresh token. If the client is not granted access, the AS responds with an error. If the user has not yet validated the authorization request, the AS informs the client about that.

2.1.3 OAuth 2.0 Attacker Model

The "OAuth 2.0 Threat Model and Security Considerations" [15] defines certain capabilities and limitations of a potential attacker. An OAuth 2.0 attacker is able to monitor any communication between clients and ASs as well as clients and RSs, but not between ASs and RSs. They can fully access the network, are capable of performing attacks without a resource limitation, and can control up to two of the three communication participants to attack the remaining one(s).

2.1.4 Updated OAuth 2.0 Attacker Model

An updated version of the OAuth 2.0 attacker model is described in the "OAuth Security Best Current Practice"-Draft [10]. It defines the attacker types A1-A5. The OAuth communication participants must be protected against the first two attacker types. Further, they should be protected against the remaining three attacker types. The five attacker types can work together, mostly the web attacker or the network attacker in combination with the remaining three attacker types.

A1 - Web Attacker This attacker can control different network endpoints, such as browsers and servers. The respective RO, AS and RS that function as communication participants in an OAuth flow and need to be protected, are excluded from this. Instead, the web attackers can operate their own clients, ASs and RSs. They can also function as an end-user by employing their credentials or obtained secrets, such as authorization codes. Further, the A1 attacker can cause end-users to follow malicious URIs, e.g., through phishing. Web Attackers can not manipulate messages not intended for them.

A2 - Network Attacker This attacker has the capabilities of the web attacker with additional total network control. Network attackers cannot break TLS but eavesdrop, manipulate, spoof, and block messages.

A3 - Read Authorization Response This attacker can read authorization responses but cannot change their contents (e.g., through a leak).

A4 - Read Authorization Request The A4 attacker can read authorization requests but not change their contents (e.g., through a leak).

A5 - Obtain Access Token This attacker can obtain access tokens, which can be redeemed by themselves at an honest RS.

2.1.5 OAuth 2.1

OAuth 2.0 was published over a decade ago. Since 2012, plenty of extensions and best practices were added, resulting in the confusing landscape of RFCs we have today. To correctly and securely implement an OAuth 2.0 instance, one has to read around ten different RFCs. Therefore, OAuth 2.1 [11] is currently under development. The draft aims to simplify the framework by limiting decisions that had to be made by developers before. OAuth 2.1 does not add any new extensions or features. It unites the OAuth 2.0 Security Best Current Practice [10] and additional extensions, such as the PKCE [17] in a single standard. According to Parecki [12], OAuth 2.1 mainly differs from the initial framework in the requirement of PKCE for all OAuth clients using the code flow. Also, the removal of the implicit grant and the RO password credentials grant due to security reasons is a difference. Additionally, the `redirect_uri` values in the token and authorization requests must be compared, using exact string matching. Moreover, bearer tokens cannot be used in query strings of URIs, while refresh token have to be sender-constrained or one-time use when issued for public clients [12].

2.2 OpenID Connect

OIDC is an authentication protocol that adds an identity layer on top of OAuth 2.0. The OpenID Foundation published the core specification in November of 2014 [7]. With an ID token, issued by the AS, the client can authenticate the user and establish a login session or check for an existing session. With OIDC, SSO scenarios can be realized.

The process of how a client obtains an ID token and an access token is called a flow. OIDC technically is an authentication extension for OAuth that can be "activated" by the client through sending the `openid` scope value in the authentication request. The AS issues an ID token, after a successful authentication of the user, who grants authorization. The client can then send the access token to the protected *UserInfo Endpoint* at the AS to obtain additional identity information about the user, who authorized the access. The client receives information about the AS, as the Endpoint locations, through OIDC Discovery [8], while the AS can register the client through Dynamic Client Registration [9].

ID tokens differ much from access tokens since they are not exchanged for access to protected resources but provide information about the end-user themselves. An ID token is a signed JSON Web Token (JWT) [37] that contains claims identifying the user. Additionally, the AS can first sign and then encrypt the ID tokens using JSON Web Signatures (JWSs) [35] and JSON Web Encryption (JWE) [36], resulting in a nested JWT. The OIDC core specification defines several required and optional claims, but ID token can also include other, self-defined claims. The specification requires the presence of the `iss` claim, which defines the issuer of the response (AS). It also requires the `sub` claim, that identifies the end-user and the `aud` claim, specifying the audience (client) the ID token is issued for. Further, the `exp` and the `iat` claims are required, which specify the expiration time and the time the ID token was issued in the first place. The `nonce` security claim is required, if the client sends the `nonce` parameter in the authentication request. The AS includes the parameter value as the claim value without changing it. In that, it functions as mitigation against replay attacks since the client can check whether the nonce included in the ID token is the same as the one it sent prior.

In OIDC, the authentication process is determined through different flows. As in OAuth, a client requests a certain flow by defining a `response_type` in the authorization request. OIDC supports three different flows, namely the *authorization code flow*, the *implicit flow* and the newly introduced *hybrid flow*. The latter is a combination of the first and the second flow.

2.2.1 Code Flow

The authorization code flow (response type `code`) works similarly as its OAuth version. The OIDC authorization code flow is also supported in the FAPI. The difference is that the client includes the `openid` scope in the authorization request and the AS returns a ID token in the token response. The OIDC version of the authorization code grant is marked red in Figure 2.1. Further, a client can include the `nonce` parameter in the authentication request to mitigate replay attacks. Optionally, the AS can include the `at_hash` claim, the access token hash value, in the ID token.

2.2.2 Hybrid Flow

Another option to obtain access tokens and ID tokens is the hybrid flow, displayed in Figure 2.3. The hybrid flow is also supported in the FAPI. OIDC defines three different types of hybrid flows that can be requested using the `response_type` parameter (and the `openid` parameter) in the authorization request. In the `code id_token` type, the AS replies with an authorization code and an ID token. After redeeming the code, the AS issues an access token and an additional ID token. In the `code token` type, the AS responds with an authorization code, but also with a

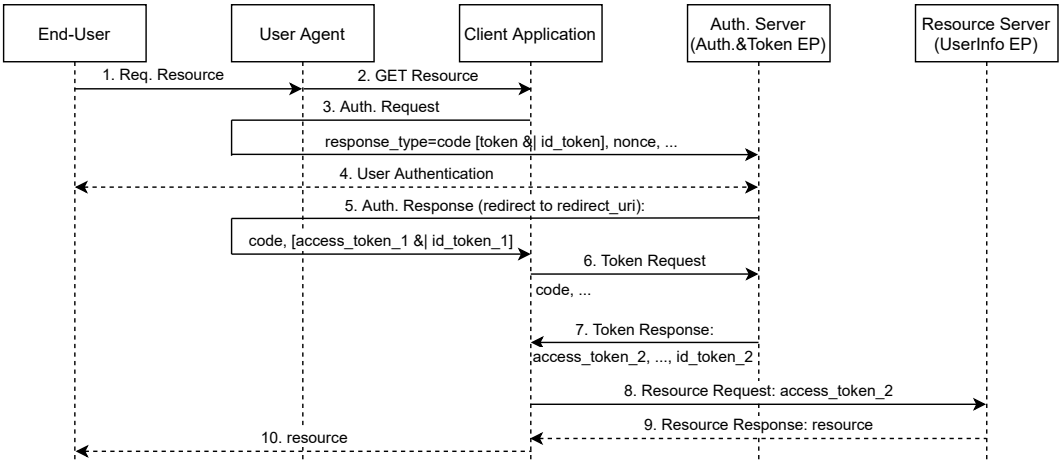
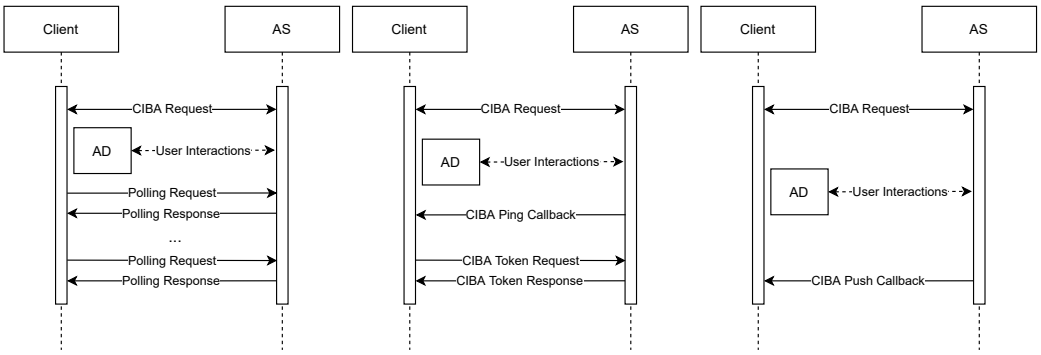


Figure 2.3: OpenID Connect Hybrid Flow.

token. The code can be exchanged in the same way as before. In the `code id_token token` type, the response includes all three values. If the AS issues two ID tokens during the flow, both tokens must have identical `iss` und `sub` claim values. The ID token sent through the UA can hold less identity information to protect the RO's privacy. However, claims being present in both tokens should have the same value. The `nonce` claim is generally required in the ID token when using the hybrid flow, while the `at_hash` is only required with response type `code id_token token`, to bind the access token to the code. Similarly, the `c_hash` claim, specifying the hash value of the authorization code, is required for the response types `code id_token` and `code id_token token`, to bind the issued code to the ID token.

2.2.3 Client-Initiated Backchannel Authentication (CIBA) Flow

In 2021, the OIDC CIBA Flow was published [54]. This authentication flow differs from the initial flows in that the client can start it without user interaction. The client is able to directly communicate with the AS, without redirection through the UA. Direct communication is realized through the newly introduced *Backchannel Authentication Endpoint*. With that the CIBA Flow does not change the general functionality of OIDC. In a CIBA scenario, end-user authentication happens on a so-called *Authentication Device (AD)*, which often is a smartphone owned by the user. Moreover, a *Consumption Device (CD)* is introduced, with which the end-user can consume the service offered by the client. The client can own the consumption device. The client can start the flow by sending a request to the Backchannel Authentication Endpoint of the AS, which answers with a unique identifier of the authentication. After the AS authenticated the end-user, it issues the tokens.



(a) Poll Mode. (b) Ping Mode. (c) Push Mode.

Figure 2.4: CIBA Flows in different Modes. [54]

There are three different ways to transmit the tokens: *Poll Mode*, *Ping Mode* and *Push Mode*. In the Poll mode, displayed in Figure 2.4a, the client sends requests to the token endpoint, asking if the tokens are available yet. The AS either responds with a token response or with a polling response, which does not contain the requested tokens. In the Ping mode, displayed in Figure 2.4b, the AS sends a ping callback to the client, stating that the tokens are available. The client then sends a token request to the token endpoint to obtain the tokens. In the Push mode, displayed in Figure 2.4c, the AS directly sends the tokens to the registered callback URI of the client. It is also possible to use sender-constrained access token (e.g., MTLS) with the Push mode, binding them to key material which is presented at the Backchannel Authentication Endpoint.
To show the support of the CIBA flow, ASs must publish the `backchannel_token_delivery_modes_supported` and the `backchannel_authentication_endpoint` AS metadata parameters. The first one holds a list of the supported modes to obtain the tokens (poll, ping, push), while the other parameter defines the URL of the Backchannel Authentication Endpoint at the AS. The client has to set one of the defined modes to obtain the tokens, either poll, ping, or the push mode with the `backchannel_token_delivery_mode` parameter. If they want to use the ping or push mode, clients also have to define the `backchannel_notification_endpoint` parameter. Another security feature introduced in the CIBA specification is the *user code* mechanism. In this mechanism, an end-user holds a secret user code, which a client needs to start a flow at the AS. The user code prevents the unsolicited starting of CIBA flows by malicious clients or users. To show support of this feature, the AS defines the `backchannel_user_code_parameter_supported` metadata parameter.

2.2.4 JWT Client Authentication

In OIDC [7], there are different ways defined to perform client authentication at the token endpoint of the AS. The default method is the `client_secret_basic`, where the client authenticates using HTTP Basic authentication with their received `client_secret`. Other ways (mandatory in the FAPI profiles) include the `client_secret_jwt` and the `private_key_jwt`, in which the client proves possession of a key, instead of directly sending the secret. Firstly, the client uses an HMAC SHA algorithm with the `client_secret` as the shared key, creating a signed JSON Web Token (JWT), which is then included in the token request. The AS can verify the signature using the shared key. Latterly, the client has registered a public key and uses the matching private key to sign a JWT, which is then included in the token request. The AS verifies the signature using the preregistered public key. Both signed JWTs have to include the following claims: `iss`, `sub`, `aud`, `jti`, and `exp`. The claims provide security, since both `iss` and `sub` hold the `client_id` and the `aud` claim specifies the audience, which is the token endpoint URI of the AS. Using these claims prevents attacks where an attacker forces the client to authenticate to a malicious token endpoint. The claims ensure that the signed JWT can not be reused to authenticate at the honest endpoint. The `jti` prevents reuse of the JWT, while the `exp` claim defines a time after which the JWT looses validity. The AS ties the authorization code to the `client_id`, so that only the correct client can redeem it at the token endpoint, by authenticating to the AS.

2.3 Extensions

Understanding the security measures mandated by the FAPI requires knowledge of several extensions for OAuth 2.0 and OIDC. In the following, we will explain common security extensions for OAuth 2.0 and OIDC through a standard authentication flow. In Table 2.5, the explained security extensions and the thereby protected messages are displayed. Additional information for each extension can be found in Appendix A.

Authorization Request After the end-user initiates the authorization flow, the client sends an authorization request. Since it is sent through the end-user's UA, the request can be manipulated. **JWT-Secured Authorization Request (JAR)** [32] extends the OAuth 2.0 authorization framework, so that authorization request parameters can be encoded in a JWT. Thus, authorization requests can be integrity protected by signing the JWT with JWS and encrypted by using JWE. A more detailed description of the JWT-Secured Authorization Request (JAR) extension is given in section A.1. Another solution to protect the authorization request is **Pushed Authorization Request (PAR)** [33]. With Pushed Authorization Request (PAR), clients can push the authorization request content to the AS. The

Table 2.5: Extensions securing different requests and responses.

Message	Extension
Authorization Request	JAR, PAR, RAR
Authorization Response	JARM, ID token as detached signature
Token Request	PKCE, MTLS for client authentication
Token Response	MTLS for token binding, DPoP
Resource Request	MTLS for token binding, DPoP
Introspection Response	Signed JWT introspection response

request is thereby not sent through the end-user's UA. Pushing the authorization request has various advantages, such as integrity and authenticity protection as well as confidentiality. With PAR, more request data can be pushed than in a conventional way since URLs have a length restriction. PAR also implements client authentication at the AS, before any user interaction happens. A more detailed description of the PAR extension is given in section A.2. Another extension for the transmission of more data is **Rich Authorization Request (RAR)** [27]. It introduces the new authorization request parameter `authorization_details`. This parameter helps to translate a high demand for authorization data present in the financial context. A more detailed description of the Rich Authorization Request (RAR) extension is given in section A.3.

Authorization Response After the end-user AS receives the authorization request and after successful user authentication, it answers with the authorization response. This response is sent through the end-user's UA to the client. As with the authorization request, the response is not protected from manipulation. By design, OIDC offers the possibility to protect authorization response values such as `state`. If the hybrid flow is used with the response type `code id_token`, the ID token can function as a **detached signature**. Response parameters or their hash values can be included in the ID token sent from the authorization endpoint. Alternatively, the **JWT Secured Authorization Response Mode (JARM)** [2] extension introduces a JWT-based mode, where the response parameters are encoded as a JWT. JWTs enable the possibilities of signing and encrypting the authorization response parameters. Further, it enables sender authentication and audience restriction. A more detailed description of the JWT Secured Authorization Response Mode (JARM) extension is given in section A.4.

Token Request After the client receives the authorization response containing an authorization code, it sends the token request. The token request includes the

code to exchange it for tokens. If the authorization code leaks to an attacker, it can use it to obtain tokens. In case of a confidential client, client authentication is used to prohibit this, since the code is bound to the client ID. Thereby, Client authentication performed on the transport layer is more secure than sharing secrets, as specified in OAuth. **Mutual TLS (MTLS)** [31] provides this kind of client authentication. A more detailed description of the Mutual TLS (MTLS) extension is given in section A.6. If a public client is used, client authentication is not an option. Therefore, **Proof Key for Code Exchange (PKCE)** [17] was created. The measure binds the authorization request to the token request. Through the PKCE parameters in the token request, the public client can prove that it has sent the authorization request. A more detailed description of the PKCE extension is given in section A.5.

Token Response If the authorization code is valid, the AS sends a token response, that includes an access token and might include a refresh token and an ID token. An attacker that obtains these tokens can redeem them at the RS since the access token and the refresh token are bearer tokens. A secure alternative to bearer tokens are sender-constrained tokens. **Mutual TLS (MTLS)** [31] can be used for binding the token to the client it is issued for. Thereby, the client uses MTLS for connecting to the token endpoint of the AS, which is then able to bind the issued token to the client's certificate. A more detailed description of the MTLS extension is given in section A.6. Alternatively, **Demonstration of Proof of Possession (DPoP)** [34] can be used. This extension describes token binding on the application layer. With Demonstration of Proof of Possession (DPoP), public and confidential clients can prove possession of a private key to redeem sender-constrained access tokens or refresh tokens. The standardization introduces the DPoP proof, which is a client-signed JWT, attached to the token request as the `DPoP` HTTP header. A more detailed description of the DPoP extension is given in section A.7.

Resource Request After receiving the token, the client redeems it at the RS using the resource request. Attackers that obtain the access token can redeem it themselves if the token is not sender-constrained. For token binding, **MTLS** (see section A.6) or **DPoP** (see section A.7) can be used.

Introspection Response With token introspection, the validity of a received token can be verified, or additional information regarding the token can be obtained at the respective token introspection endpoint at the AS. Token introspection responses are not integrity protected and can be manipulated. Through **signed JWT introspection responses** [55] , they can not be manipulated anymore. A more detailed description of the JWT introspection response extension is given in section A.8.

3 The Financial-Grade API (FAPI)

The financial-grade API (FAPI) is a security profile for OAuth, designed as an interoperable solution for Open Banking scenarios. It aims to fit the high need for security and regulatory aspects that core OAuth can not offer due to it being a framework. It was developed as a reaction to the European Payment Service Directive 2 (PSD2), published in 2015 [67]. The profile offers a secure and efficient alternative to screenscraping. Despite its origin in the financial context the FAPI profiles are also useful in other scenarios with a high demand for security. For example, in the e-health environment, the FAPI can enable secure access to and interaction with user accounts.

3.1 FAPI 1.0

FAPI 1.0 consists of two profiles: the *baseline* and the *advanced* profile. The final Financial-grade API 1.0 security profiles [22] [23] were published in March 2021 by the FAPI working group. Prior, the FAPI second implementer's drafts [3] [4] were published in October 2018. Conformance test suites are available to ensure the correct implementation of the standard. Implementations successfully passing the tests are listed as certified FAPI ASs and clients. The specification combines existing OIDC security features with best practices and other security recommendations to define two individual profiles, which are designed for different security levels.

The baseline profile [22] is developed for read-only access to APIs with a moderate risk, holding sensitive data. Therefore, security measures such as PKCE [17] (described in section A.5) or client authentication are mandatory. Further, the enforcement of security best practices such as the use of a short token lifetime or preregistered redirect URIs are enforced by the FAPI regulations.

On the other hand, the advanced profile [23] is intended to protect read and write access to APIs, bearing a high risk. Additional measures such as signing authorization requests and responses and using sender-constrained access tokens are applied on top of the baseline requirements to ensure high security.

Further specifications, developed in the context of FAPI include the "FAPI 1.0 — JWT Secured Authorization Response Mode for OAuth 2.0" (JARM) [2] (described

in section A.4), and the "Financial-grade API: Client Initiated Backchannel Authentication Profile" (CIBA) [53]. Since the latter is based on the specifications given in the FAPI second implementer's drafts, the differences between the drafts and the final profile versions are explained as well.

3.1.1 Part 1: Baseline

The baseline profile aims to protect (financial) APIs bearing a moderate risk [22]. It requires the implementation of the following standards "The OAuth 2.0 Authorization Framework" [13] (described in section 2.1), "The OAuth 2.0 Authorization Framework: Bearer Token Usage" [14], the "Proof Key for Code Exchange by OAuth Public Clients" [17] (described in section A.5) and the "OpenID Connect Core 1.0 incorporating errata set 1" [7] (described in section 2.2).

Use of Strong Cryptography The profile requires the use of strong cryptographic parameters to mitigate cryptographic attacks. Thereby, security can be ensured for a longer time, even with growing computation power. To fulfill this requirement, ASs must issue a `client_secret` to enable cryptographically strong keys due to sufficiently high entropy when using symmetric cryptography. Further, confidential clients must verify that their received `client_secret` is at least 128 bits long, in a symmetric cryptography context. Further, the AS and confidential clients shall only deal with > 2048 bit long keys for RSA algorithms and > 160 bit long keys in an elliptic curve context. ASs must mitigate credentials-guessing attacks by only issuing access tokens, (optional refresh tokens,) and authorization codes with sufficient entropy. Therefore, the attackers success probability shall be $\leq 2^{(-128)}$ and ought to be $\leq 2^{(-160)}$.

Safe Redirect URIs Redirect URIs are a popular attack vector. An attack on redirect URIs is described in subsection 6.2.11. Hence, the AS must enforce the pre-registration of redirect URIs and the use of the https scheme. Furthermore, the AS must reject authorization requests without the `redirect_uri` parameter or with `redirect_uri` values that do not match any of the pre-registered redirect URIs.

Public and confidential clients must utilize individual redirect URIs for each authorization server they communicate with. Further, clients must store the redirect URI value in the end-user's session to compare it with the receiving address of the authorization response. If they differ, the client must stop the flow.

Client Authentication ASs must always perform client authentication when dealing with confidential clients to bind the authorization code to them. This technique ensures that only the client bound to the code can exchange it for the respective tokens at the AS. For this purpose, they shall either use MTLS [31] (described in section A.6) for OAuth Client Authentication or one of two authentication methods specified in the OIDC standard (described in subsection 2.2.4). In both the `client_secret_jwt` and the `private_key_jwt` method, the client creates a JWT. In the first method, the client calculates a MAC using its client secret (symmetric key), while in the latter, the client signs the JWT utilizing its private key (which matches the previously registered public key). Each way, the client proves possession of a key instead of directly sending their client credentials to the AS. Moreover, the AS must return an `invalid_client` error, in case the `iss` and `sub` claim values in the JWT differ, since they both need to contain the same `client_id`.

Similary to the AS, a confidential client must also either support MTLS [31] or one of the named JWT-based authentication methods to authenticate at the token endpoint.

Secure Ways to Deal with End-Users One goal of the Open Banking movement and the PSD2 is to enable the end-user to control their privacy and security settings. The FAPI profile also aims to provide a certain level of transparency and clarity to the user. Consequently, the AS requires a reliable and appropriate level of user authentication. Specific user consent to the requested scopes is mandatory if they have not been approved before. Further, the AS should clarify authorization details to the user, especially whether they agree to a long-term grant, meaning long-term access to their resources. The AS should also offer a revocation mechanism enabling the end-user to revoke their consent. Consent revocation should result in the issued refresh tokens and access tokens becoming invalid.

Enforcement of Security Best Practices The FAPI baseline profile also requires the adherence to particular security best practices. These basic configurations already provide a higher level of security without additional extensions. FAPI-compliant ASs have to support confidential clients and should support public clients as well. The AS must not accept previously used authorization codes and should only issue bearer tokens with a lifetime of under 10 minutes or sender-constrained ones. It is recommended to use refresh tokens instead of long-lived access tokens. Both measures reduce the attack surface for replay attacks. Generally, access tokens should have a shorter lifetime than refresh tokens. Further, the AS has to return token responses, conforming to the OAuth Authorization Framework [13]. The AS must return a list of authorized scopes together with the access token if an authorization request was sent through the front channel. Since the request is not integrity protected when sent through the front channel, the AS clarifies to the client whether the scopes were manipulated. Moreover, the AS has to require

the use of PKCE, configured to use `S256` as the `code_challenge_method`. The server also has to support OIDC discovery [8] and can support OAuth authorization server metadata [18]. The AS is not allowed to distribute discovery metadata differently.

Both confidential and public client must support PKCE with the `S256` code challenge method. Clients also have to implement a working Cross-Site Request Forgery (CSRF) protection and ensure that the (in the token response) received scope matches the scope or a subset of it that is specified in the authorization request. Additionally, clients are only allowed to make use of the AS metadata published via the metadata document at the AS's well-known endpoints. Therefore, they should act compliantly with either OIDC Discovery [8] or OAuth Authorization Metadata (RFC8414) [18].

Optional Support of OpenID Connect Further, in case the client sends the `openid` scope in the authorization request, the AS must require the presence of the `nonce` parameter in the authentication request. This parameter aims to mitigate replay attacks. If the `openid` scope is not requested, the AS must require the `state` parameter instead, which is used to protect against CSRF attacks.

The client has to include the `openid` scope and the `nonce` parameter in the authorization request, if user authentication is wanted. If not, the client must include the `state` parameter instead.

Ensurance of Secure Transmissions Every interaction between any of the communication participants must be encrypted with TLS. Thereby, participants have to be compliant with the recommendations given in the BCP195 [69]. They must only use TLS version 1.2 or later versions and perform TLS server certificate checks. Furthermore, it is recommended that all endpoints employ DNSSEC and that endpoints used by web browsers can assure that connections cannot be downgraded.

Communication Between Client and RS RSs must support the HTTP GET method and should also support Cross Origin Source Sharing (CORS) or different methods that enable access to the endpoints for JavaScript clients, if the RS decides to. In this context, RFC6819 (OAuth 2.0 Threat Model and Security Considerations) [13] should be considered before allowing JavaScript clients.

The RS must only accept access tokens sent in an HTTP header and must not accept transmission via query parameters. Hence, clients must send the access token via an HTTP header.

The RS must only return the respective resources, which are precisely defined by the associated entity and the granted scopes combined. The resource response must be UTF-8 encoded. Regarding the HTTP headers included in the response, the RS must

use the Content-Type Header `Content-Type: application/json` (if applicable) and the HTTP `Date` header to transmit the server date. Additionally, it is required to include the `x-fapi-interaction-id` response header. This header's value must be the same as the one received in the client request, or a UUID if the value was not provided. The `x-fapi-interaction-id` aims to track the interaction between client and RS, therefore the value must be logged by the RS. Furthermore, the RS must not dismiss requests containing a `x-fapi-customer-ip-address` header that holds a valid IPv4 or IPv6 address.

Clients are allowed to include the `x-fapi-auth-date` header in the resource request to specify the last time when a customer logged in at the client. They are also permitted to transmit the IP address of the customer via the `x-fapi-customer-ip-address` header. Further, clients may send a UUID as the `x-fapi-interaction-id` request header to support the RS's logging process.

Validation of the access token The RS must ensure that the received access token neither expires, nor has been revoked. The RS also needs to check whether the associated scopes grant access to the resources the server is protecting. Further, it needs to identify the entity linked to the access token.

Miscellaneous Requirements and Recommendations Generally, it is recommended to use certified implementations of the FAPI profile or check compliance of own implementations by using the provided certification test suite [1]. Additionally, strict access control to the logs is recommended as well. If a higher security level is needed, especially against the failure of integrity protection, it should be considered to apply the FAPI Security Profile 1.0-Part 2: Advanced.

Further, native apps must follow the best security practices for native apps [57] in BCP212 and must neither support *Private-Use URI Scheme Redirection*, nor *Loopback Interface Redirection*. Native apps are only allowed to support *Claimed HTTPS Scheme URI Redirection*. In case one AS deployment has to provide individual authorization endpoints for a variety of "brands", an individual issuer must be used for each of them.

3.1.2 Part 2: Advanced

The advanced profile aims to protect (financial) APIs bearing a high risk, e.g., initiation of transactions [23]. Therefore, the profiles aim to provide non-repudiation and sender-constrained access tokens. Non-repudiation can be realized through signing the authorization request and response. It requires the implementation of the following standards "The OAuth 2.0 Authorization Framework" [13] (described in section 2.1), "The OAuth 2.0 Authorization Framework: Bearer Token Usage" [14], the "Proof Key for Code Exchange by OAuth Public Clients" [17] (described in

section A.5) and the "OpenID Connect Core 1.0 incorporating errata set 1" [7] (described in section 2.2).

The advanced profile generally builds upon the baseline profile but adds additional security measures. Further, PKCE is not required, and some specified provisions override given ones. In the advanced profile, only confidential clients are supported. Additionally, either the Code Flow (described in subsection 2.1.1) with the JARM or the Hybrid Flow (described in subsection 2.2.2) with ID token as detached signatures can be used.

Client Authentication The AS must authenticate the client either using MTLS [31] (described in section A.6) with the `tls_client_auth` or the `self_signed_tls_client_auth`, or using the `private_key_jwt` method, specified in the OIDC standard. In contrast to the baseline profile the symmetric `client_secret_jwt` authentication method is not supported.

Sender-Constrained access token ASs must only issue sender-constrained access tokens, supporting MTLS (described in section A.6). Confidential clients must also support MTLS as a mechanism for sender-constraining. RSs must not accept bearer tokens, but only support MTLS for sender-constrained access token. Thereby, they must be compliant with the requirements in RFC8705 [31].

Secure Ways to Deal with End-User The confidential client must assure that the AS provides an appropriate level of user authentication.

Protecting the Authorization Request - JWT-Secured Authorization Request (JAR) The advanced profile requires securing the authorization request by employing JAR [32] (described in section A.1) to provide non-repudiation. Therefore, the AS requires a JWT request object, which is signed with a JWS. The request object must be either passed by value (`request` parameter) or by reference (`request_object` parameter). Further, the request object must contain the `exp` claim, specifying a point in time $\leq$ 60 minutes after the `nbf` claim, and the `aud` claim, holding (an array of) the AS issuer identifier URL. The request object must also contain an `nbf` claim specifying a point in time $\leq$ 60 minutes in the past.

The confidential client must send either the `request` or the `request_uri` parameter in the authorization request. Additionally, confidential clients must include all parameters in the request object signed for the authorization request. If PAR is not used, the confidential client has to include the `response_type`, the `client_id` and the `scope` parameter/value pairs outside of the request object. That is necessary due to the OAuth 2.0 syntax. These values have to match the ones stated in the request object. If PAR is used, the client only has to send the `client_id` to the

authorization endpoint. In the request object, the confidential client has to integrate the `aud`, the `exp`, and the `nbf` claim, similarly to the requirements given for the AS.

Protecting the Authorization Response Another measure to provide non-repudiation is integrity protecting the authorization response. For this purpose, the advanced profile specifies two different ways: either using JARM [2] (described in section A.4) or utilizing the ID token as a detached signature.

In the case of using JARM [2], the response type has to be `code` with the response mode being `jwt`. ASs have to use JWT-secured authorization responses as stated in the response encoding paragraph of the standard [2]. Further, confidential clients have to validate the authorization response according to the processing rules given in the specification [2].

Another possibility is to use the ID token as a detached signature. The ID token can be utilized in a way that is not related to the identity of the end-user. ID tokens include the issuer identifier of the AS and are signed by it. The ID token can include an ephemeral subject identifier. Hashes of unprotected response parameters, such as `code` or `state` are included in the token and therefore signed by the AS. The hash of the code is called `c_hash`, while the hash value of the state is named `s_hash`. When using the ID token as a detached signature, the response type `code id_token` has to be used. The AS further must support OIDC and signed ID tokens. It is also recommended, that it supports signed and encrypted ID tokens. ID tokens have to be returned as detached signatures and must include the `s_hash` value. On the other hand, it is not ought to return any sensitive PII in the token. If that cannot be avoided, the ID token should be additionally encrypted.

The confidential client has to behave similarly: It must include the `openid` scope, require a JWS signed ID token and support signed and encrypted ID token. Further, confidential clients have to check that the authorization response was not manipulated by validating the ID token as the detached signature. They also must verify the `s_hash` by calculating the hash of the `state` value send in the authorization request and comparing both.

Pushed Authorization Endpoint (PAR) The AS is permitted to support the *pushed authroization request* endpoint as stated in RFC9126 [33] (described in section A.2). If the AS decides to support pushed authorization requests, it must employ PKCE with the `S256` code challenge method.

Confidential clients also have to employ PKCE [17] with the `S256` code challenge method when utilizing pushed authorization requests. Furthermore, they must transmit the `client_id` parameter-value pair to the authorization endpoint.

Enforcement of Security Best Practices The AS must exclusively process the parameters given in the signed request object, either transmitted via the `request` or the `request_uri` parameter. It is recommended that the JWKS URI endpoint is used by the AS to publish public keys.

The confidential client also has to utilize the JWKS URI endpoint or the `jwks` parameter as specified in the Dynamic Client Registration Protocol [16].

Further, protected actions must only be executed if a valid access token is present. Another recommendation for the advanced profile is not to use the `x5u` and `jku` JOSE headers.

Ensurance of Secure Transmissions TLS versions below 1.3 must be treated carefully, only four cipher suites are allowed: `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`, `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256` and `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`. The latter ones shall only be used with key lengths $\geq$ 2048 bits. Additional cipher suites, which are approved by the best current practices for the use of TLS and DNS [69], can be used for the authorization endpoint to ensure a higher level of interoperability. Further, all endpoints ought to use DNSSEC, and the `jwks_uri` endpoint must be served over TLS.

Strong Cryptography ASs and clients must use `PS256` or `ES256` algorithms for JWS. Algorithms using `RSASSA-PKCS1-v1_5` are not recommended and using `none` is forbidden. It is also forbidden to use the `RSA1_5` algorithm for clients and ASs performing JWE. Avoiding the same `kid` for more than one key in a JSON Web Key (JWK) set is recommended. If this is not possible, other JWK attributes, as `kty`, `use` or `alg`, must be used as additional identification to select the suitable verification key.

3.1.3 FAPI Second Implementer's Draft

Both FAPI profiles originate in the second implementer's drafts [3] [4]. These drafts are referenced by the CIBA FAPI profile (subsection 3.1.4). Further, the client and AS setup in chapter 5 are certified for the draft of the advanced FAPI profile. Therefore, it is essential to understand the differences between the drafts and their final version.

The read-only profile [3] is renamed to the baseline profile, while the read-write profile is now called the advanced profile. In the old draft version, the AS was only recommended to reject previously used authorization codes, in the read-only profile. Additionally, there was no recommendation for the access token lifetime present in the draft. Similarly, ASs did not need to support OpenID discovery or OAuth

authorization server metadata for metadata distribution. Clients did not need to use metadata distributed through these explicit mechanisms. In the draft, public clients had alternatives to using PKCE. Further, they were not obliged to check the received scopes present in the token response. In Contrast to the final version, the RS was required to include the charset in the `Content-type` header. Moreover, there were less thorough/comprehensive TLS recommendations, meaning that the draft did not recommend TLS stripping attack prevention measures or the use of DNSSEC. Minor changes primarily include clarifications, such as rejecting requests lacking the state or nonce values, and recommendations [38].

In the read-write profile [4], request objects which do not include a `nbf` claim were accepted. An `exp` claim that states a lifetime of more than 60 minutes was also allowed. Further, using PAR without PKCE parameters was accepted, as well as JWKs including keys with similar `kids`. Additionally, public clients were not explicitly forbidden, as well as the hybrid flow with the response type `code id_token token`. In the draft, JARM was only an addition to the use of OIDC while it is an alternative in the final version. Furthermore, the use of OAuth Token Binding [5] was possible for sender-constraining access tokens. Other minor changes include clarifications and recommendations regarding wording and cryptographic considerations [38].

3.1.4 Client Initiated Backchannel Authentication Profile

The second draft of the FAPI CIBA profile [53] was published in 2019. This secure profile for CIBA can be used in scenarios in which the client needs to initiate the flow. An example of such a scenario is the user authorization at a "point of sale" terminal in a shop. In that case, user consent is granted through an authentication device while the client initiates the flow from the consumption device. The CIBA profile builds upon the FAPI 1.0 baseline and advanced profile as well as the OIDC CIBA flow (described in subsection 2.2.3). All communication participants (ASs, only confidential clients and RSs) have to follow the requirements stated in both baseline and advanced profiles. Furthermore, the provisions regarding the assurance of secure transmissions and strong cryptography given in the advanced profile apply. Additional requirements are specified in the following paragraphs.

CIBA-Specific Security Requirements Since the CIBA flow specifies a new endpoint and three different methods for obtaining access tokens, additional requirements are specified. First, the AS must not support CIBA push mode, since the access token would received from the token endpoint. Instead, it must support the poll mode and can support the ping mode if intended. Second, the AS has to support both unsigned and signed authentication requests to the Backchannel Authentication Endpoint, compliant with the CIBA specification [54]. A new extension

to the authentication request, presented in this specification, is the `request_context` claim. This optional claim is a JSON object and can be included in the authentication request to share additional information with threat detection systems. The JSON object can hold information such as the geolocation of the consumption device. The AS can require the presence of the claim in the authentication request.

Confidential clients are advised to only send signed authentication requests to the *Backchannel Authentication Endpoint*.

Secure Ways to Deal with the End-User The specification further states that the AS must ensure an appropriate level of user authentication, matching the granted operations. Confidential clients have to verify that the AS has performed the appropriate level of user authentication.

Enforcement of Security Best Practices The AS must require the existence of a unique authorization context in the authorization request or a `binding_message` in the authentication request. Additionally, it is recommended for the AS that it does not make use of the `login_hint` or the `login_hint_token` for transmitting "intent ids" or other authorization metadata. Further provisions are that the AS must return the `acr` claim in the ID token, if supported and if requested by the client. Moreover, the AS must only accept signed authentication requests with `nbf` and `exp` claims limiting the request lifetime to ≤ 60 minutes.

Confidential clients must further require appropriate authorization context in an authorization request or include a `binding_message` in the authentication request. Furthermore, the confidential client must not send `x-fapi-customer-ip-address` and `x-fapi-auth-date` headers. However, it is recommended to send metadata about the consumption device, when the device is not in control of the client.

Several additional security measures are specified to ensure that authentication sessions cannot be started without the user's knowledge. One of these measures recommends that the `login_hint` has nonce properties and that ASs can accept the `id_token_hint` alternatively. In the latter case, the client needs to store the received ID token for later use. Hence, the identification mechanism for obtaining the token should be appropriate for the used channel. In addition, the specified `user_code` mechanism can be used.

3.2 FAPI 2.0

The first ideas for the financial-grade API security profile 2.0 [30] [28] were published in February of 2020. The second version of the FAPI standardization aims

to provide even more interoperability. Interoperability is achieved by limiting the options. Other goals of FAPI 2.0 include providing better non-repudiation while simplifying the development. Therefore, the FAPI working group works on an underlying attacker model [29] and takes learnings from practice into consideration. Two profiles belong to the second FAPI specification: the baseline profile [30], which is even more secure than the FAPI 1.0 advanced profile, and the new advanced profile [28], which improves non-repudiation through signing all relevant requests and responses. However, the FAPI 2.0 profiles do not build upon FAPI 1.0, but rather define an independent FAPI version.

3.2.1 Attacker Model

The financial-grade API 2.0 specifies an attacker model [29]. Since the FAPI promises a high level of security, it is necessary to define against which threats the profiles protect. Hence, several attackers, security goals, and non-repudiation requirements are modeled. The model forms a basis for the selection of suitable security measures. The goal of the attacker model is to facilitate future security analysis. It can not provide security guarantees. An analysis that provides the basis for this model was conducted in 2019 by Fett et al. [59]. The paper describes a "reverse engineered" attacker model of FAPI 1.0. The model defines three security goals: authorization, authentication, and session integrity.

Authorization No attacker has any chance to access the end-user's resources and obtain or use the RO access token for this purpose.

Authentication There is no possibility for an attacker to log in as the user at a client or obtain and use an ID token to do so.

Session integrity Session integrity means that a user logs in as themselves and accesses their own resources. Through CSRF or session swapping attacks, session integrity can be broken. Additionally, an attacker can force the user to log in under the attacker's identity or cause the user to access the attacker's resources. The latter case describes the session integrity goal regarding authorization. The first one represents the authentication session integrity goal since attackers should not be able to log in users under a different identity.

The specification also models eight different attacker types that it aims to protect against. The model assumes that TLS, JWKS, browsers, and endpoints all function correctly since their security is out of scope.

A1 - Web Attacker The Web attacker can send and receive messages. They can also participate in flows. They can use different standard tools and tamper messages at their own endpoints. Further, they can send links to honest users who visit them.

A1a - Web Attacker (participating as AS) A1a is a web attacker who additionally participates as a malicious AS. Hence, they can reuse messages from honest AS and redirect users to honest ASs endpoints.

A2 - Network Attacker Network attackers can control the entire network, meaning they can intercept, block and tamper with messages addressed to other parties in the network.

A3a - Read Authorization Request A3a is one of two attackers at the authorization endpoint. This type of attacker is a web attacker. The A3a attacker can read authorization requests, which are sent in the front channel. These can be leaked by the browser history, for example.

A3b - Read Authorization Response A3b is the second attacker at the authorization endpoint. They are a web attacker, able to read the authorization response, which can be leaked by web browser logs, for example.

A5 - Read and Tamper with Token Requests and Responses This type of attacker sits at the token endpoint and can read and tamper with token requests and responses since it can impersonate the token endpoint of the honest AS to the client.

A7 - Read Resource Requests and Responses A7 is one of the attackers sitting at the RS. This type has the same abilities as the web attacker but reads resource requests and responses, e.g., through reading TLS interception proxy logs.

A8 - Tamper with Resource Responses A8 is the other attacker at the RS. It has the same abilities as A7 with the additional capability of manipulating responses, e.g., through a compromised reverse proxy.

In addition to the different attacker types defined in this model, the additional non-repudiation requirements NR1 - NR9 are given. These requirements are necessary to ensure that a communication party cannot deny it has sent a specific request. That is of special interest in a financial context since payment initiations must be traceable

to one specific person or party. Messages that need protection, such as application-level signatures to provide non-repudiation, include pushed authorization requests (NR1) and responses to pushed authorization requests (NR2). The requirements also include authorization requests (NR3) and responses (NR4), sent through the front channel, as well as ID token contents (NR5). Furthermore, introspection responses (NR6), user info responses (NR7), resource requests (NR8), and responses (NR9) are affected.

3.2.2 Baseline Profile

The baseline profile [30] aims to protect (financial) APIs bearing a moderate risk. It requires the implementation of the following standards "The OAuth 2.0 Authorization Framework" [13] (described in section 2.1), "The OAuth 2.0 Authorization Framework: Bearer Token Usage" [14], the "Proof Key for Code Exchange by OAuth Public Clients"[17] (described in section A.5) and the "OpenID Connect Core 1.0 incorporating errata set 1" (described in section 2.2) [7].

Ensurance of Secure Transmission All endpoints provided by any of the protocol participants must be TLS protected and should use DNSSEC. For this reason, TLS 1.2 or later versions must be used. Additionally, communication participants have to adhere to the recommendations stated in [69]. When TLS 1.2 is used, only four defined cipher suites are allowed, and two of them must have specified key lengths. TLS certificate checks are mandatory. Furthermore, endpoints that are used by web browsers must prevent the downgrading of TLS connections, e.g., by utilizing preloaded HTTP strict transport security policies.

Strong Cryptography All protocol participants must behave compliant with the JWT best current practices [39] when dealing with JWTs. The best practices recommend avoiding `RSA-PKCS1 v1.5` algorithms used for encryption and implementing the Elliptic Curve Digital Signature Algorithm (ECDSA) in a deterministic way, as specified in RFC6979 [40]. These recommendations must be followed. Especially, the `none` algorithm must not be used. Further, the second version of the FAPI enforces, the use of either `PS256` or `ES256` algorithms. All RSA keys must be at least 2048 bits long, while elliptic curve keys must be at least 160 bits long. To prevent an attacker from guessing credentials, access tokens, refresh tokens, and authorization codes must have at least 128 bits of entropy.

Pushed Authorization Endpoint (PAR) In contrast to FAPI 1.0, Pushed Authorization Requests [33] (described in section A.2) are no longer optional, but required. Thus, ASs must only support client-authenticated PARs and reject common authorization requests or PARs sent without client authentication. Further,

ASs must only accept PARs that include the `redirect_uri` parameter. When issuing the `request_uri`, the servers must include `expires_in` values that are between 5 and 600 seconds. Clients have to use PAR to meet the AS's requirements.

Rich Authorization Requests (RAR) New to the FAPI cosmos is the Rich Authorization Request [27] (described in section A.3), which introduces the `authorization_details` parameter in the authorization request. This parameter allows communicating more details about the requested resource access. Especially in a financial context, this can be very helpful to transmit payment initiation details, such as the names of sender and receiver, their IBANs, the amount of money, the time, and currency, for example. Hence, ASs must support RAR and with that the `authorization_details` parameter.

Sender-Constrained access token In the second version of the FAPI, there are two ways to realize sender-constrained access tokens: MTLS [31] (described in section A.6) and DPoP [34] (described in section A.7). The AS must only issue access tokens using one of the two sender-constraining methods. Clients and RSs also have to support at least one of the methods.

Client Authentication As in FAPI 1.0, MTLS [31] and `private_key_jwt` are supported methods for client authentication. ASs have to authenticate clients, either using MTLS or using `private_key_jwt`. Clients need to authenticate using one of these methods. When the latter method is used, the client must include the AS's issuer identifier as a string in the `aud` claim sent in the client authentication assertions. The AS then must accept the received claim value.

Redirection The AS must require the `redirect_uri` parameter in the pushed authorization request. Redirect URIs must not use the "http" scheme since the AS must only transmit authorization responses via encrypted connections. One exception exists for native clients using Loopback Interface Redirection. Further, when the AS redirects requests, including user credentials, it must not use the HTTP status code 307 (temporary redirect). When redirecting the user agent, it is recommended to use the HTTP status code 303. ASs and clients must not expose open redirectors.

Enforcement of Security Best Practices Additional security measures that the AS must fulfill include the distribution of discovery metadata with the metadata document, as specified in [18]. Further, ASs must support the authorization code grant but must not accept requests sent in the resource owner password credentials

grant, the implicit grant, or the hybrid flow. ASs have to support confidential clients and enforce PKCE with the `S265` code challenge method. Moreover, the AS must include an `iss` parameter in the authorization response to prevent mixup attacks. Authorization codes that have been used before must be rejected by the AS.

Clients also must use the authorization code grant and PKCE with the `S256` code challenge method. The `iss` parameter sent in the authorization response must be checked by the client as stated in [42]. In cases, where MTLS is either used as a authentication method or a sender-constraining mechanism, clients must utilize the `mtls_endpoint_aliases` metadata, as specified in [31].

Further, ASs must not use refresh token rotation when the client has not received the new refresh token response. However, it can be used in cases where resending the request, which includes the last valid refresh token, succeeds.

Communication Between RS and Client RSs must only accept access tokens sent in the HTTP header, but not in query parameters. Therefore, clients must transmit access tokens in the HTTP header. Further, it must solely return resources identified by the combination of entity and scope, stated with the access token. Additionally, the RS must identify the respective entity.

Validation of access token When receiving resource requests, RSs must verify the access tokens, focussing on validity, integrity, expiration, revocation, and the scopes. Regarding the scopes, the RS must check that they and the included `authorization_details` [27] grant access to the resources the RS represents.

3.2.3 Advanced Profile

The advanced profile [28] aims to protect (financial) APIs bearing a higher risk by providing additional non-repudiation and other means of protection against the modeled attacker. The Advanced profile builds upon and extends the baseline profile, meaning that all baseline specifications apply to the advanced specification. The high level of non-repudiation can be realized through signing the authorization request and response by using JAR [32] (described in section A.1) and JARM [2] (described in section A.4). Additionally, introspection responses can be signed.

The advanced profile requires the implementation of the following standards “The OAuth 2.0 Authorization Framework” [13] (described in section 2.1), “The OAuth 2.0 Authorization Framework: Bearer Token Usage” [14], the “Proof Key for Code Exchange by OAuth Public Clients” [17] (described in section A.5), the “OpenID Connect Core 1.0 incorporating errata set 1” [7] (described in section 2.2) and the

"OAuth 2.0 Token Introspection" [41]. Since the draft is in an early stage of development, it is subject to change, and open questions remain.

Protection of the Authorization Request - JWT-Secured Authorization Request (JAR) A JWT-Secured Authorization Request [32] is used to ensure non-repudiation of the authorization request. For this purpose, the AS has to support JAR and with that accept signed request objects at the PAR endpoint. The client has to sign request objects, as specified in the JAR specification [32].

Protecting the Authorization Response The JWT Secured Authorization Response Mode (JARM) [2] is used to prevent tampering with the authorization response. In this context, the AS must support JARM. Clients have to check that authorization responses are protected with either JARM or ID tokens as detached signatures and verify the respective signatures. Since the FAPI 2.0 advanced profile is still a draft, this requirement might change to solely supporting JARM.

Securing Token Introspection Responses JWT secured token introspection responses [55] are used to provide a higher assurance that the responses were issued by a respective AS. ASs can but are not obliged to provide token introspection. If they provide token introspection, they have to sign the introspection responses, conforming to RFC7662 [41]. Clients that make use of the mechanism must request signed and conforming token introspection responses.

Open Questions At this point, it remains an open question how to protect the communication involving the RS, meaning which mechanisms can/should be used to sign resource requests and responses.

4 (Tabular) Comparison of Security Features

The FAPI working group claims that the FAPI is a "highly secured OAuth profile" [22]. The four FAPI profiles are compared to OIDC [7], OAuth 2.0 [13] and 2.1 [11] to understand how this statement originated. The following (tabular) comparison presents an overview of the applied configurations. All examined standards differ in their attacker models (see subsection 2.1.3, 2.1.4, and 3.2.1) and in the (security) features provided. Further, it is noticeable that the security demand rises over time, resulting in the creation and application of a variety of security extensions covering new security vulnerabilities. The comparison shows a broad standardization scope in the FAPI profiles, as well as many similarities between OAuth 2.1 and the FAPI. Lastly, the FAPI CIBA profile [53] is compared to the OIDC CIBA specification [54] and the OAuth 2.0 device authorization grant [19].

4.1 Methodology

We performed a comparison of the supported and required features to answer the question of whether the FAPI profiles provide security improvements compared to the classic OAuth and OIDC flows. The tabular comparison provides a detailed overview of the specified security features of the different profiles. While this might be helpful as a reference, it offers little information about the overall security of the profile on hand. Multiple features in one profile can be replaced by a single feature in a newer profile while providing a similar level of security. A security evaluation of the FAPI compared to other protocol specifications can be found in chapter 7.
The tables do not claim to be complete and are subject to change since FAPI 2.0 is still under development. The tables were assembled by mostly incorporating security features and configurations specified in the four FAPI profiles. Then, we verified whether and how these aspects were referenced in the other three protocols, namely OAuth 2.0, OAuth 2.1, and OIDC. Thereby, we added suitable security measures to the table, e.g., additional client authentication methods, stated in the none-FAPI profiles. Finally, the features were sorted into different categories, providing a better overview. Furthermore, one has to consider that certain features are not supported in some protocols as they were not yet invented. An example for this is the term "credentialed clients" which is introduced in the freshly published OAuth 2.1 draft [11]. Moreover, the protocols were designed for different use cases: OAuth

is designed for authorization, while OIDC is created for authentication, resulting in OAuth not supporting ID tokens. OAuth and the FAPI profiles also differ in their attacker models, resulting in "missing" security measures due to different attacker capabilities obsoleting these.

4.2 Comparison of Attacker Models

In this section, we present the differences between the three introduced attacker models. A tabular overview is provided in Table 4.1.

Table 4.1: Comparison of the three different attacker models. Key: x = explicitly defined; (x) = implied.

Attacker Models	**OAuth**	**Updated OAuth**	**FAPI 2.0**
Web Attacker	x	x	x
Web Attacker as AS	x	(x)	x
Network Attacker	x	x	(x)
Read Authorization Request		x	x
Read Authorization Response		x	x
Read and Tamper with Token Requests and Responses			x
Read Resource Requests and Responses			x
Tamper with Resource Responses			x
Obtain Access Token		x	(x)

Compared to the FAPI attacker model [29] specified in subsection 3.2.1, the OAuth attacker model [15] (subsection 2.1.3) is significantly less complex. The FAPI-attackers can be web attackers, participating as AS. They can also control the entire network, enabling the attackers to intercept, block and tamper with messages. FAPI attackers can further sit at the authorization endpoint. This means that they can read authorization requests and responses. Alternatively they can sit at the token endpoint, enabling them to read and tamper with token requests and responses. Additionally, the attacker can also sit at the RS so that they can read resource requests and read and manipulate the responses.

The main difference between these two attacker models is that the FAPI attacker can read and tamper with various requests and responses due to leaks through browser histories or logs. Although the OAuth attacker has access to the network, using TLS will mitigate most attacks. This results in the OAuth attacker

not having "real" access to the authorization, token, and resource requests and responses.

Compared to the original OAuth 2.0 Attacker model, the updated version, presented in [10] (see subsection 2.1.4), includes more types of attackers. While the original model describes the capabilities of a web attacker and a network attacker, the updated one adds attackers that are capable of reading authorization requests and responses. In the newer model, an attacker can also obtain access tokens. The FAPI attacker model includes the attacker types defined in the updated attacker model and additional ones. It adds a web attacker that participates as an AS and gives an attacker the capabilities of reading and tampering with token requests and responses. Moreover, the FAPI 2.0 attacker model also defines attackers that can read resource requests and responses and tamper with the latter. Thereby, the FAPI presents an attacker model that is way stronger than the original OAuth attacker model. The updated OAuth version can be seen as a subset of the FAPI attacker model, meaning that it only aims to protect against a subset of the attacks the FAPI aims to protect against. This has to be kept in mind when comparing the different standards.

4.3 Comparison with the FAPI

In this section, we present the most important results of our comparison. The complete tabular comparison of 138 (security) features is presented in Appendix B. In Table 4.2, a tabular overview is provided. It summarizes the significant differences between the examined specifications.

General Configuration The general configurations of the profiles differ in their acceptance of various client types and flows. While both OAuth versions and OIDC support public clients, the FAPI strongly recommends the use of confidential clients, since they are able to authenticate at the AS. Another noticeable difference is the acceptance of different flows. OAuth 2.0 allows the authorization code flow, the implicit flow, and the resource owner password credential grant. In contrast, newer standards such as the FAPI or OAuth 2.1 forbid the use of the latter two due to severe security problems. Another reason why the second FAPI version only supports the authorization code flow might be interoperability and a simplification of the profiles.

Client Authentication Regarding client authentication, one can notice a shift towards asymmetric cryptography, namely `private_key_jwt` and MTLS. The FAPI 1.0 baseline profile still accepts a symmetric method, but the newer FAPI profiles only accept asymmetric methods. Even OAuth 2.1 recommends the use of

Table 4.2: Selection of security feature comparison, presented in the tabular comparison the Appendix. Key: 0 = not mentioned; # = explicitly not permitted; [x] = may/can support // named; (x) = should support; x = must support.

	FAPI 1.0		**FAPI 2.0**		**OAuth**		**OIDC**
Features/Configuration	**Bas.**	**Adv.**	**Bas.**	**Adv.**	**2.0**	**2.1**	**1.0**
Public Clients	(x)	#	0	0	x	x	x
Confidential Clients	x	x	x	x	x	x	x
Authorization Code Flow	x	x	x	x	x	x	x
Hybrid Flow	0	x	#	#	0	0	x
Other Flows	0	#	#	#	x	#	x
Symmetric Client Auth.	x	#	#	#	[x]	[x]	x
`priv_key`	x	x	x	x	[x]	(x)	x
MTLS (Auth.)	x	x	x	x	[x]	(x)	0
PKCE	x	(x)	x	x	0	x/(x)	0
`state`	x	x	#	#	(x)	[x]	(x)
`nonce`	x	x	0	0	0	[x]/x	[x]/x
MTLS (Sen. Const. AT)	0	x	x	x	0	(x)	0
DPoP	0	0	x	x	0	(x)	0
JAR	#	x	0	x	0	0	[x]
JARM	#	x	0	x	0	0	0
ID token as det. sig.	#	x	0	#	0	0	[x]
PAR	0	[x]	x	x	0	[x]	0
RAR	0	0	x	x	0	[x]	0
Pre-registered redirect URIs	x	x	#	#	x/(x)	x	x

asymmetric authentication, while OAuth 2.0 and OIDC do not give any restrictions.

General Security Features The development over time also shows the replacement of `state` and `nonce` by PKCE, manifesting in OAuth 2.1's enforcement of PKCE and the optionality of the state parameter. Furthermore, the encryption of ID tokens becomes redundant since they are only exchanged in the backchannel due to the forbiddance of the hybrid flow. We observed that the FAPI version became more secure from version to version.

Sender-constrained Token Back when the OAuth 2.0 framework was published, sender-constrained access tokens were not discussed. The first FAPI profile dictates the use of MTLS for the purpose of sender-constraining since this was the only mechanism available at the time. The second FAPI profile also supports the use of DPoP, a proof-of-possession mechanism that is still in a draft state. OAuth 2.1 recommends sender-constrained tokens and with that also names DPoP.

Protecting Authorization Requests and Responses Many non-repudiation features, such as JAR or JARM, are neither specified in both OAuth protocols nor OIDC. This may be due to the fact that the FAPI also protects against attackers with access to requests and responses beyond the transport layer. The more powerful attacker model makes additional protection on other layers mandatory. One difference in the usage of JAR by the two FAPI profiles is the use of JAR at the PAR endpoint.

Regarding the protection of the authorization response, support of ID tokens as detached signatures differs in the two FAPI versions. The first one supports it as a temporary solution, while the second version explicitly forbids it. OIDC supports this mechanism by design, according to [23].

Securing the Token Introspection Response and UserInfo response FAPI 2.0 also dictates the use of JWT-secured introspection responses as an additional non-repudiation feature, which is not necessary to ensure security against the OAuth 2.0 attacker.

PAR and RAR Another temporal development can be seen in the use of PAR. While it was optional in the first FAPI version, it becomes mandatory in the second one, most likely due to it becoming an official standard. This further results in additional requirements for the use of PAR being specified. The new standard is an optional feature in OAuth 2.1, as RAR is, which also becomes mandatory in the second FAPI version.

Secure Redirection Regarding secure redirection, the FAPI specifies way more (precise) requirements than the non-FAPI profiles. Especially striking is that the preregistration of redirect URIs is not mandatory in the new FAPI version anymore since the redirect URIs are transmitted and secured in the Pushed Authorization Request. Moreover, OAuth 2.1 also defines more recommendations for redirecting.

Communication between Client and Resource Server Another remarkable feature is that both FAPI versions clearly define requirements for secure communication between client and RS. In contrast, none of the remaining protocols cover this at all. One main difference between the two FAPI versions is the integration of the `x-fapi-*` headers, which were removed in the newer specification.

User Contact Similarly, requirements for the user context were specified in the first FAPI profile. Except for OIDC, this was not done in the remaining specifications.

Strong Cryptography Cryptographic and algorithm recommendations are also mainly specified in the FAPI profiles and OIDC. Noticeable is that OIDC supports algorithms that are not secure. The reason for this is the publication date of the standard. OAuth might not specify many cryptographic requirements since there are no ID tokens and no signatures present.

Assurance of secure transmission Regarding TLS recommendations, the second FAPI specification dictates the most, even concrete cipher suites. Both OAuth versions and OIDC also recommend some of the recommendations, stated in the FAPI specifications but in a less specific way. At the publishing date of OAuth 2.0, TLS 1.1 was the most used version, and TLS 1.2 was still new, so enforcement of TLS 1.2 was not feasible. On the other hand, the protocol clearly states that appropriate TLS versions will vary over time

Enforcement of Security Best Practices Security best practices partly being recommended in both OAuth versions and OIDC become mandatory in the FAPI profiles. Some of the first FAPI features are not defined in the second FAPI profile. Especially refresh token rotation is not recommended in the second FAPI profiles since it does not bear a more significant security benefit but can come with operational issues.

Additionally, only FAPI 1.0 profiles recommend the use of certified FAPI-implementations, since there is not yet a certification process for FAPI 2.0 available.

4.4 Comparison with the FAPI CIBA Profile

In order to answer the question, whether the FAPI CIBA profile [53] (subsection 3.1.4) provides security improvements compared to the OIDC CIBA extension [54] (subsection 2.2.3), a comparison of the supported and required features is performed. The tabular comparison provides an overview of the specified security features of both specifications. A summarized table of the most significant differences is displayed in Table 4.3. The tables presented do not claim to be complete and are subject to change since the FAPI CIBA profile is still in a draft state. We assembled the tables by incorporating security features and configurations specified in the FAPI CIBA profile. Then, we verified the presence of these measures in the OIDC CIBA specification. Finally, the features were sorted into different categories, providing a better overview.

A tabular comparison to the OAuth 2.0 Device Authorization Grant [19] (subsection 2.1.2) can be found in the Appendix B. Comparing the CIBA profile to the Device Grant is not in focus since the flows may seemingly have similarities, such as polling the AS or employing the `user_code` parameter. However, the `user_code` functions completely different in the both specifications. In the Device Grant, the user code is a required parameter and an essential part of the authorization flow. In contrast, the user code is an optional parameter in the CIBA flow that functions as spam protection. Another striking difference is the different scenarios both flows are used in, depending on the capabilities of the client. In the Device Grant, the client has no access to a browser, while it does in a CIBA scenario, resulting in RO-client interaction, which is not possible in the Device grant. Since both flows have different application purposes and the FAPI CIBA profile cannot directly be used for the Device grant, a tabular comparison might not be reasonable.

Regarding the general configurations, a huge security improvement is the prohibition of public client in the FAPI CIBA profile. Another CIBA-specific limitation is that only ping and poll mode are accepted in the FAPI profile. The poll mode must be supported and is the preferred method for obtaining tokens. Additionally, the FAPI profile specifies accepted methods for client authentication, which are not limited in the OIDC CIBA standardization. Moreover, the FAPI enforces several CIBA specific security features, such as the mandatory use of signed backchannel authentication requests, the `binding_message` parameter or the inclusion of different claims, which are optional in the original specification. Another visible difference is the support of all FAPI-related measures, enforced through two statements included in the FAPI CIBA profile. The AS must support all regulations stated in “clause 5.2.2 of Financial-grade API - Part 1 and clause 5.2.2 of Financial-grade API - Part 2” [53] of both implementer’s drafts [3] [4]. Further, confidential clients must support the regulations given in “ clause 5.2.4 of Financial-grade API - Part 1 [FAPI1] and clause 5.2.4 of Financial-grade API - Part 2 [FAPI2]” [53]. Thereby, the profile

Table 4.3: Comparison of the FAPI CIBA Profile with the OIDC CIBA Flow. Key: 0 = not mentioned; # = explicitly not permitted; [x] = may/can support // named; (x) = should support; x = must support.

Features/Configuration	**FAPI 1.0 CIBA Draft**	**OIDC CIBA**
Public Clients	#	x
Confidential Clients	x	x
Push Mode	#	x
Poll Mode	x	x
Ping Mode	[x]	x
Signed Backchannel Auth. Requests	x	[x]
`binding_message` / unique Auth. Context	x	[x]
`user_code` mechanism	[x]	[x]

references the second implementer's draft versions, described in subsection 3.1.3. The FAPI CIBA profiles also includes the same TLS and algorithm considerations that are given in the main FAPI profiles. The OIDC CIBA version does not specify TLS considerations to this extent.

5 Exemplary Setup of a FAPI Client and Provider

The OpenID Foundation offers certification mechanisms for FAPI ASs and clients. We set up a certified client and AS to gain practical experience and impressions from working with real-life FAPI implementations. First, we describe our reasons for selecting the *Gluu oxd Client API 4.2* [48] and the *Gluu Server 4.2* [46]. Then we describe the implementation of our testing tool and the setup process. Finally, we will analyze the two applications to document the security features they are providing.

5.1 Selection of a FAPI Client and Provider

The OpenID Foundation offers a certification program for AS and client applications [1]. Apart from the standard OpenID certification, FAPI certifications are available for ASs and clients. Further, a certification is available for the FAPI CIBA profile. Both the FAPI 1.0 baseline and advanced profiles are covered in different development stages. Implementations are certified for the FAPI 1.0 Second Implementer's draft and the final version. Additionally, different configurations for each version can be certified, e.g. AS using MTLS or private key authentication. Further, different adaptions of the FAPI, such as *Brazil Open Banking*, can be certified as well.

Compared to the number of the certified FAPI ASs, less FAPI clients are certified. There are only four different certified clients, apart from the clients belonging to Brazil Open Banking. Due to these limitations, we decided to first choose a well-documented, open-source client implementation before choosing a matching open-source AS. The *Gluu oxd Client API 4.2* [48] is a well-documented client implementation, certified for the second implementer's draft [1]. The oxd Client API is certified for two configurations: MTLS and private key client authentication. The *Gluu Server 4.2* [46] is a well-documented AS implementation, which is also certified for the both configurations of the second implementer's draft [1].

Gluu oxd Client API 4.2 The oxd Client API [48] is a REST application that is an optional component of the Gluu Server. For this thesis, the oxd Client API is examined in its standalone version. The oxd Client API provides different endpoints that a client application can call. The client API then interacts with the AS on the client application's behalf, as shown in Figure 5.1. This concept enables different client applications to use the same authorization and authentication backend, which can be an advantage. The oxd Client API is available on Github [61]. Further, the oxd APIs are also documented [43].

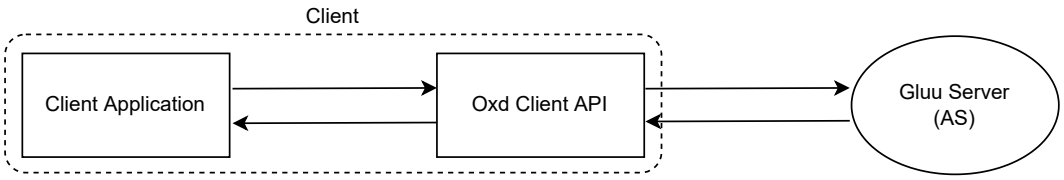


Figure 5.1: Gluu communication scenario.

Gluu Server 4.2 The Gluu Server [46] is an AS implementation that is used for identity and access management (IAM). The application supports the use of OAuth and OIDC, as well as other standards used for federated identity. The Gluu Server consists of several software components, partly available on Gluu's Github pages [63]. Documentation for the Gluu Server APIs is also available. The *Gluu Server API* offers endpoints used for token introspection and revocation. The *OIDC API* exposes basic OIDC endpoints. Both are part of Gluu's AS, called *oxAuth* [60]. The *oxTrust API* offers configuration endpoints for the *oxTrust Admin GUI* [49].

5.2 Description of the Environment

For further security analysis, we used Docker to containerize the oxd Client API and the Gluu Server. This enables reproducibility and an enclosed environment.

5.2.1 Gluu oxd Client API 4.2

For the oxd Client API [61], we assembled a Dockerfile ourselves, which is attached in the Appendix C in Listing C.1. If it is desired to have both the Gluu Server and the oxd Client API in one network, there are two possibilities. One is to activate the `oxd_server` during the deployment of the Gluu Server by adding `OXD_SERVER = True` in the `settings.py` file. Another way that provides a standalone version is to load the oxd Client API Dockerfile in a `docker-compose.override.yml`. However,

in the environment created for testing, both FAPI implementations are independently set up. We implemented a testing tool written in python for the oxd Client API to ensure its functionality and to explore its features.

Starting Script Our tool builds and starts the Docker container with the chosen configuration file and launches mitmproxy with our addon `oxd_addon.py`. The use of the Docker API enables an easy way of integration [6].

Configuration Currently, three configuration files are available one for client authentication with *MTLS* and one for authentication using *Private Key*. Apart from these FAPI variants, we also provide the third file, which enables a standard code flow, without any FAPI configuration. The three options are differently configured versions of the `oxd-server.yml`. This main configuration file enables a complete configuration of the oxd Client API, such as configuring the server or the logging [44]. Additionally, a specific file for the registration of the client application is used, depending on the selected option.

Mitmproxy We chose *mitmproxy* [26] as an easy way to communicate with the oxd Client API. The proxy enables sending requests and responding to requests. This is necessary to test the oxd Client API, since mitmproxy impersonates the client application and the AS.

Additionally, mitmproxy [26] offers several helpful features such as intercepting HTTP(S) requests and responses, saving conversations, and the generation of TLS certificates [25]. Another advantage of mitmproxy is the addon framework [24]. Every addon is essentially a python class that can expose commands for the mitmproxy console tool. Further, addons can use event hooks to change mitmproxy's behavior. For example, by hooking the `request` event, the addon can print something to the event log if a defined request was sent. With the `ctx` module important standard objects are exposed. It enables, e. g., writing data to the event log.

We created the addon `oxd_addon.py` that defines the `Oxd` class. In this class we define all requests a client application would send in a FAPI flow. Every request is bound to a command, which adds the respective request to the mitmproxy master flow view. There it can be modified and sent by the user. The oxd Client API is thereby triggered to send requests to the AS, which is impersonated by the mitmproxy in this case. The mitmproxy can respond to these requests since we hooked the request event. We defined responses for each API endpoint, the oxd Client API requests. The responses are automatically sent once the endpoint is called. Additionally, we hooked the response event to save values sent by the oxd Client API, such as the `oxd_id`, the `state` and `nonce` values. This is necessary, as the oxd Client API expects these values in the following responses of the AS.

The implementation we created is easily expandable regarding configuration files and endpoints. It provides a simple way of displaying the flow, ensuring the functionality of the setup, and helps to understand how the oxd Client API works.

5.2.2 Gluu Server 4.2

The Gluu Server provides a Docker installation [45]. The executable python zip archive builds the Docker Compose environment. It is up to the developer to decide which services should run in the environment. Each service runs in its own container. The Gluu Server consists of 4 to 19 services. The most important services are the *oxAuth* [60] services and the *oxTrust* [62] service. The first service functions as the authorization server and supports e.g., OIDC and dynamic client registration. The latter enables the administration of the Gluu Server via a web interface.

Due to a lack of time, we were not able to configure the Gluu Server and to implement a testing tool for the Gluu Server. However, with the concepts demonstrated in the oxd Client API testing tool, further researchers can easily extend the current functionality to test the Gluu Server.

5.3 Documentation of Security Features

In this section, we give an overview of the security features supported by the two examined FAPI implementations. We further explain how they can be configured.

5.3.1 Gluu oxd Client API 4.2

The oxd Client API supports several security features. They can be configured in the `oxd-server.yml`. An excerpt of the configuration file can be seen in Listing 5.2. It displays security-related configuration details.

The oxd Client API supports the use of the `nonce` and `state` values. Their expiration time can be specified, and they can be encoded. Further, the oxd Client API can enforce the validation of the `s_hash`, the `at_hash` and the `c_hash` in the ID token.

The oxd Client API accepts several client authentication methods: `none`, `client_secret_basic`, `client_secret_post`, `client_secret_jwt`, `private_key_jwt`, `access_token`, `tls_client_auth` and `self_signed_tls_client_auth` [44]. The oxd Client API is FAPI certified for MTLS and `private_key_jwt`.

The oxd Client API supports passing request objects by value and by reference. The request objects can be signed and encrypted. The support of the hybrid flow enables the use of ID tokens as detached signatures. Access tokens can be configured to be (signed) JWTs, and their lifetime can be limited. ID tokens can be signed and encrypted as well. Additionally, the *UserInfo* response can be signed and encrypted.

The oxd Client API provides different FAPI-related validations. They can be enabled by setting the key `fapi_enabled` to true. This setting makes the audience, the issuer, and the nonce parameter mandatory in the ID token. Further, the `iat` value in the ID token is verified. It is examined, whether the ID token includes an `azp` claim, if it also includes multiple audiences [51]. The oxd Client API enables the pre-registration of redirect URIs.

```
#server configuration
    fapi_enabled: true
    mtls_enabled: false
    mtls_client_key_store_path: ''
    mtls_client_key_store_password: ''
    tls_version: ['TLSv1.2']
    tls_secure_cipher: ['TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384']
    state_expiration_in_minutes: 5
    nonce_expiration_in_minutes: 5
    protect_commands_with_access_token: true
    protect_commands_with_oxd_id: []
    [...]
defaultSiteConfig:
    response_types: ['code', 'code id_token']
    request_uris: []
    client_jwks_uri: ''
    token_endpoint_auth_method: ''
    token_endpoint_auth_signing_alg: ''
    access_token_as_jwt: false
    access_token_signing_alg: ''
    access_token_lifetime: null
    id_token_signed_response_alg: ''
    id_token_encrypted_response_alg: ''
    id_token_encrypted_response_enc: ''
    user_info_signed_response_alg: ''
    user_info_encrypted_response_alg: ''
    user_info_encrypted_response_enc: ''
    request_object_signing_alg: ''
    request_object_encryption_alg: ''
    request_object_encryption_enc: ''
    [...]
```

Listing 5.2: oxd-server.yml

The oxd-server can be configured to use a specified TLS version with a distinct TLS cipher suite. Access token and ID token signing algorithms can be defined. Algorithms for signing and encrypting the user info response and request objects can further be specified. Accepted signing algorithms are: `HS256`, `HS384`, `HS512`, `RS256`, `RS384`, `RS512`, `ES256`, `ES384`, `ES512`, `PS256`, `PS384` and `PS512`.

Due to the design of splitting the client components, the oxd Client API provides an additional security mechanism to secure the communication between the client application and the oxd Client API. The connection between these two can be secure by an access token and/or the `oxd_id`.

5.3.2 Gluu Server 4.2

The Gluu Server supports several security features. They are configured in the `oxauth-config.xml` [64]. It holds the configuration of the Gluu AS. It can be edited through the admin UI or the oxTrust API by calling the `/configuration/oxauth/settings` endpoint [50].

An excerpt of the configuration file can be seen in Listing 5.3 [47]. It displays security-related configuration details.

```
"fapiCompatibility":true,
"userInfoSigningAlgValuesSupported":[],
"userInfoEncryptionAlgValuesSupported": [],
"userInfoEncryptionEncValuesSupported":[],
"idTokenSigningAlgValuesSupported":[],
"idTokenEncryptionAlgValuesSupported":[],
"idTokenEncryptionEncValuesSupported":[],
"requestObjectSigningAlgValuesSupported":[],
"requestObjectEncryptionAlgValuesSupported":[],
"requestObjectEncryptionEncValuesSupported":[],
"tokenEndpointAuthMethodsSupported":[],
"authorizationCodeLifetime":'',
"refreshTokenLifetime":'',
"idTokenLifetime":'',
"accessTokenLifetime":'',
"dynamicRegistrationEnabled":true,
"clientWhiteList": ["*"],
"clientBlackList": ["*.attacker.com/*"],
[...]
```

Listing 5.3: oxauth-config.json

The Gluu Server supports similar features as the oxd Client API. It enables client authentication offering the same options as the oxd Client API. The Gluu Server is also only FAPI-certified for MTLS and `private_key_jwt`. The Gluu Server supports the manual and the dynamic registration of clients. It supports white- and blacklisting of client redirection URIs. The Gluu Server enables a stricter behavior of the AS through the `fapiCompatibility` parameter. Further, it allows configuration of the access token, authorization code, refresh token, and ID token lifetime. It also supports the use of MTLS for token binding. The Gluu Server also supports using signed request objects and specifies allowed algorithms for signing and encryption. Moreover, the Gluu Server supports signed and encrypted ID tokens and UserInfo responses. Thereby the supported algorithms are similar to the ones the oxd Client API supports.

The Gluu Server further functions as a FAPI CIBA provider. It thereby supports the poll and the ping mode with MTLS or private key client authentication [1]. It further supports the authorization device grant.

6 Security Catalog

We will evaluate the security of the FAPI by describing different attacks and their countermeasures. Throughout this security catalog, we will discuss which well-known attacks on OIDC [7] and OAuth [13] are theoretically applicable on FAPI implementations. We describe attacks and possible countermeasures, as well as countermeasures described in FAPI 1.0 [22] [23], FAPI 2.0 [30] [28] and the FAPI CIBA profile [53]. Additional attacks on the system, on RO and on RS are attached in the Appendix (seeAppendix D).

6.1 Attacks on Clients

In this section, we describe a collection of attacks on clients. When testing a FAPI-certified client, this catalog can be taken as a reference. Testers should verify that the examined client is protected by the named countermeasures.

6.1.1 Cross-Site Request Forgery (CSRF)

CSRF attacks are widely known and described in [13]. The attacker tricks the victim into following a malicious URI. In an OAuth scenario, the redirect URI of the client is a target of interest in the authorization code and implicit grant. With it, the attacker can inject their own access token or authorization code, which might then be used to access the resources of the attacker. The impact could cause the victim to upload sensitive data to the attacker's resources. This breaks session integrity.

Therefore, the `state` parameter can be used in requests sent to the redirection endpoint. This parameter binds the respective request to the session (UA) and is included in authorization requests and responses. With that, it also links the authorization request to the redirect URI. Alternatively, PKCE or the OIDC nonce can be used to achieve the same CSRF protection. Before utilizing PKCE, the client has to check whether the AS supports it as well. If it is not supported, the `state` or `nonce` parameter must be used instead.

Countermeasures by FAPI 1.0 In FAPI 1.0, clients must implement CSRF protection measures. Further, when the `openid` scope is requested, a nonce value must be included in the authentication request. Otherwise, the `state` parameter has to be included. Additionally, the use of PKCE is mandatory.

Countermeasures by FAPI 2.0 In FAPI 2.0, PKCE is mandatory and has to be used with the `S256` code challenge method.

6.1.2 Client Impersonation

Client impersonation is described in [13] and in [15]. Attackers can impersonate clients to get authorization, if the client's credentials are leaked. An attacker can, for example, obtain credentials through Client Credential Phishing at the Token Endpoint, as described in [23]. Authorization server endpoints are not bound to each other. Clients that do not use different `redirect_uri`s for different ASs open up a broader attack surface. This can lead to client credential and authorization code phishing at the token endpoint. In this case, an attacker social engineers the victim (client developer) into sending the token request to an attacker-controlled "token endpoint". This way, the attacker gains knowledge of the secrets and replays the token request to the "real" token endpoint.

An attacker could also launch phishing attacks when in possession of client credentials. This can happen when credentials are transmitted without protective measures, such as TLS. Further, automatically processing authorization requests if user consent was granted previously is dangerous. Attacker clients can exploit being redirected without renewed user consent to get an authorization code. The victim does not have a chance to check the client again, wherefore they do not notice the absence of the honest, public client.

As a countermeasure, clear communication with the end-user about the consent they are about to grant is always necessary. For example, informing ROs about the respective client, lifetime, and scope of the authorization process can be helpful as stated in [11]. Further, the AS can prevent this attack by authenticating the client. Therefore, different authentication methods are recommended that do not directly transmit the credentials. Instead, they use MACs or proof of possession mechanisms. Another recommendation is to not automatically accept identical authorization requests when received multiple times until it is clear who the real sender is. In the case of public clients, `redirection_uri`s must be pre-registered to mitigate the attack. Further, the scope of automatically issued access tokens can be reduced by ASs.

Countermeasures by FAPI 1.0 FAPI 1.0 enforces the use of TLS for all transactions. Further, explicit user approval is necessary, and grant details must be displayed to the RO. For client authentication, MTLS, `client_secret` or `private_key_jwt` have to be used. These authentication methods do not require directly sending client credentials to the AS. With transport layer and asymmetric authentication, the `client_secret` is never exposed to attackers. Additionally, for public clients, it is necessary to pre-register redirect URIs.

Countermeasures by FAPI 2.0 FAPI 2.0 also enforces TLS and, since only confidential clients are allowed, client authentication is mandatory. For the authentication, only asymmetric methods and MTLS are allowed and with PAR, client authentication happens early on. With transport layer and asymmetric authentication, the `client_secret` is never exposed to attackers.

6.1.3 User Session Impersonation

User session impersonation is described in [15]. Attackers can impersonate the client and the user session. With DNS or ARP spoofing, an attacker can steal the code from the transmission between the browser and callback endpoint. Then they can send the code to the client themselves and obtain access to protected resources. A countermeasure is using a redirect URI with an HTTPS scheme. Another countermeasure is using server authentication to validate the redirect URI.

Countermeasures by FAPI 1.0 To protect against DNS spoofing, the FAPI 1.0 recommends the use of DNSSEC. Further, redirect URIs must use the HTTPS scheme.

Countermeasures by FAPI 2.0 To protect against DNS spoofing, the FAPI 2.0 recommends the use of DNSSEC. Further, redirect URIs must use the HTTPS scheme.

6.1.4 Manipulation of Scripts

The manipulation of scripts is defined in [15]. Clients written in a scripting language can be vulnerable to an attacker manipulating or exchanging the respective client scripts. The attacker may modify the scripts by acting as the client webserver and using DNS and ARP spoofing. This can lead to an attacker gaining knowledge of user credentials.

To prevent this kind of attack, ASs must authenticate the server, presenting the scripts. Clients must check that the respective scripts were not modified during

transportation. Further, single-use secrets, such as the `client_secret` values, reduce the attack's effectiveness.

Countermeasures by FAPI 1.0 In the first FAPI specification, no explicit protection against the manipulation of scripts is defined. However, it is recommended to employ DNSSEC on all endpoints to mitigate DNS spoofing and perform TLS server certificate checks.

Countermeasures by FAPI 2.0 The newer standardization does not explicitly define protection measures but recommends using DNSSEC and enforcing TLS server certificate checks.

6.1.5 Compromising Multiple Clients Sharing the Same Key

The threat of multiple clients sharing the same key is described in [23]. For the use of MTLS, certificates are needed. These are often issued by certificate authorities. Sometimes, this happens on an organizational level, where one organization shares the same certificates with more than one client. This is a disadvantage since the compromise of one client leads to the compromise of all clients using the same key material. As a countermeasure, issuing certificates on a client level is recommended so that no two clients share the same key.

Countermeasures by FAPI 1.0 The advanced profile names this risk and makes developers aware of it and the possible compromisation of all clients.

Countermeasures by FAPI 2.0 The current FAPI 2.0 draft does not yet specify this risk.

6.1.6 Request and Response Disclosure and Modification

Disclosure and modification of requests and responses are described in [7] and [23]. Requests can be disclosed to attackers through a compromised UA, for example. In plain OAuth, authorization requests and responses are not signed and are thereby not integrity protected. In this case, attackers can manipulate authorization requests in the browser since TLS is stopped there.

Attacks against authorization responses happen in scenarios where the attacker AS and the honest AS make use of the same client. Unprotected authorization responses are vulnerable to parameter injection.

Signed request objects can be used to prevent attacks from inserting any parameter wanted. To further protect authorization responses from parameter injections, ID tokens as detached signatures, including the `c_hash`, `at_hash` and `s_hash` or JARM can be used for integrity protection.

Countermeasures by FAPI 1.0 Application-level confidentiality is provided in the advanced profile through the use of JAR and JARM or ID tokens as detached signatures, including the `c_hash`, and `s_hash`.

Countermeasures by FAPI 2.0 In FAPI 2.0, application-level confidentiality is enforced through the mandatory use of PAR, which enables encryption of the authorization request. In the advanced profile, JAR and JARM are used to provide additional security.

6.1.7 Mixup Attack

The mixup attack is described in [10] and [23]. In a mix-up attack the client confuses a compromised AS with the honest AS. Thereby, the client sends an authorization code or access token directly to the attacker. Prerequisites for this kind of attack is a client that supports multiple ASs. It needs to support a compromised AS, controlled by the attacker. If a client only supports one AS, mix-up attacks are not applicable.

In this attack, the malicious AS returns the same `client_id`, as an honest, registered one. As soon as the client is social-engineered into, for example, clicking on a malicious link, an authorization request is sent to the malicious AS. The AS then redirects the request to the respective honest AS, which returns an authorization code to the client if it is already logged in. The client then automatically sends a token request to the malicious AS. The attacker thereby obtains a valid authorization code, which they can redeem for an access token at the honest AS.

To further mitigate the risk of mix-up attacks, the client stores the issuer identifier, which is the intended receiver of the authorization request. The issuer is then bound to the UA and identifies the connection between the used authorization and token endpoint. The issuer is included in the authorization response by the AS. The client then compares it with the saved issuer it sends the authorization request to and stops the flow if they do not match. A less effective alternative to the use of an issuer, which does not require new features, is utilizing different redirect URIs for each AS. The client checks whether the URI at which the authorization response was received matches the one used for the intended AS. If they do not match, the client stops the flow. However, this mitigation can be circumvented, which disqualifies it.

Countermeasures by FAPI 1.0 The advanced profile of FAPI 1.0 names the threat of AS mix-up attacks and possible mitigations. The FAPI can prohibit this attack from being successful by including the honest ASs issuer identifier in the secured authorization response. It can be secured via ID token as a detached signature or by using JARM.

Countermeasures by FAPI 2.0 FAPI 2.0 enforces the validation of the `iss` parameter by the client in the authorization response.

6.1.8 Obtaining Refresh Token and Access Token

Obtaining refresh tokens and access tokens is described in [15]. Refresh tokens have to be protected against being obtained from a web or a native application. Attackers can also steal access tokens from an accessible storage device. Since access tokens and refresh tokens are normally bearer tokens, attackers can redeem them for access to protected resources.

To mitigate the risk of stolen refresh tokens, the `client_id` related to the refresh token must be validated each refresh. The token scope can be limited and refresh tokens and client secrets must be revocable. Client authentication and common web server protection measures help protect a web application from being stolen from. Further, a physical device lock and the secure storage of secrets prevent reading the token from the native app's local file system. The token scope and lifetime should be limited to reduce the impact of a stolen access token. To mitigate the stealing of access tokens, they should be kept in transient or private memory. The same measures that help protect refresh tokens, protect access token.

Countermeasures by FAPI 1.0 FAPI 1.0 enforces client authentication for confidential clients. Further, token revocation mechanisms should be provided. FAPI 1.0 limits the token lifetime to a maximum of ten minutes. Further, access tokens are sender-constrained in the advanced profile.

Countermeasures by FAPI 2.0 FAPI 2.0 also enforces client authentication for confidential clients, which are the only clients supported. The baseline profile discourages the use of refresh token rotation since it "doesn't bring any security benefits for confidential clients and can cause significant operational issues" [30]. In FAPI 2.0, the AS must support sender-constraining access tokens, using either MTLS or DPoP, in both profiles.

6.2 Attacks on Authorization Servers

In this section, we describe a collection of attacks on ASs. When testing a FAPI-certified AS, this catalog can be taken as a reference. Testers should verify that the examined AS is protected by the listed countermeasures.

6.2.1 PKCE Downgrade Attack

The PKCE downgrade attack is described in [10]. Targets for this kind of a common CSRF attack are an AS that is supporting, but not enforcing, the use of PKCE, and a client that relies on PKCE for CSRF protection. The attacker acts as an end-user, which can toggle the use of PKCE through the inclusion of the `code_challenge` parameter. The attacker can then send an authorization request without a `code_challenge` parameter, resulting in the AS not enforcing PKCE. Thereby, the AS issues an authorization code that is not bound to a `code_challenge`. CSRF attacks on the client (see subsection 6.1.1) are possible since it solely relies on PKCE for CSRF protection.

Countermeasures include the correct use of the `state` parameter or a stricter use of PKCE. Latter means, that ASs supporting PKCE have to check that a `code_challenge` is included in the authorization response. If so, they have to bind it to the issued code. Further, it must be checked that if a code was received at the token endpoint and a code challenge was included in the authorization request, the matching code verifier is included in the token request. Additionally, if no code challenge was included in the authorization request, token requests including a code verifier must be rejected. Mandatory use of PKCE includes the already described measures.

Countermeasures by FAPI 1.0 The AS and the client must enforce the use of PKCE, according to RFC7636 [17].

Countermeasures by FAPI 2.0 The AS and the client must enforce the use of PKCE, according to RFC7636 [17].

6.2.2 Server Masquerading

Server Masquerading is described in [7]. Attackers can disguise their malicious ASs as honest ASs in a variety of ways. Authenticating the server mitigates this. Therefore, the client can employ signed and/or encrypted JWTs, as defined in OIDC core.

Countermeasures by FAPI 1.0 FAPI 1.0, supports the use of signed and encrypted JWTs for server authentication. Therefore, JARM or ID tokens as detached signatures are employed in the advanced profile.

Countermeasures by FAPI 2.0 FAPI 2.0 also supports the use of signed JWTs through the use of JARM in the advanced profile.

6.2.3 Authorization Code Phishing

Authorization code phishing is described in [15]. Authorization codes can be phished by an attacker imitating the web app client and using DNS or ARP spoofing. Countermeasures helping against authorization code phishing include redirect URIs with an HTTPS scheme, which are authenticated by the end-users browser through server authentication. Further, a mandatory client authentication of confidential client mitigate phishing attempts.

Countermeasures by FAPI 1.0 In the FAPI 1.0, redirect URIs must have an HTTPS scheme. Further, the use of DNSSEC is recommended to mitigate DNS spoofing. Additionally, client authentication is mandatory, if confidential clients are used.

Countermeasures by FAPI 2.0 In FAPI 2.0, redirect URI also must have an HTTPS scheme, and DNSSEC is recommended. In this case, only confidential clients are supported and client authentication is mandatory.

6.2.4 Refresh Token Phishing by Counterfeit Authorization Server

Refresh token phishing by a counterfeit authorization server is described in [15]. Attackers can impersonate an AS and proxy the requests to the AS since no verification of the AS is specified in OAuth. Therefore, they can intercept requests sent by the client by using DNS or ARP. Users must verify the authenticity of the AS before continuing. As a countermeasure, ASs can use TLS and inform the end-user about potential phishing.

Countermeasures by FAPI 1.0 FAPI 1.0 recommends the use of DNSSEC and enforces the use of TLS.

Countermeasures by FAPI 2.0 FAPI 2.0 also recommends the use of DNSSEC and enforces the use of TLS.

6.2.5 Authorization Code Leakage through Counterfeit Client / Injection

Authorization code leakage through counterfeit clients is described in [15]. Attackers can abuse the authorization code grant to first trick a victim to grant access to their counterfeit client sites. Then, attackers can use the code with their own user accounts to associate their accounts with the victim's resources on a client site.

This is also called authorization code injection, as described in [10]. For this attack to work, the authorization code has to be obtained. However, it is not specified how the attacker gets the authorization code. If it is not obtained through leakage by a counterfeit client, single-use authorization codes are recommended to complicate the attack. In this kind of attack, the attacker injects an authorization code into their session with a client. They then abuse the client to redeem the code for them.

As a countermeasure, the best current practices recommend binding the authorization code to the respective client instance the code was issued for. To bind the code to a client, an AS can associate the redirect URI with the authorization code so that the attack becomes visible at the token endpoint if both redirect URIs do not match. Further, pre-registered redirect URIs help, as well as deployment-specific `client_id` and `client_secret` pairs for native apps. Binding the `client_id`s to the authorization code further complicates the attack.
Newer approaches include the use of PKCE or a nonce. In the first, preferred solution, the attack fails since the code is associated with a specific `code_challenge`, not matching the attackers `code_verifier`. In the latter solution, the nonce is created and associated with the UA session by the client and is included in the initial request to the OpenID Provider, where it is associated with the authorization code and included in the ID token.

Countermeasures by FAPI 1.0 In FAPI 1.0, pre-registration of redirect URIs are mandatory. The FAPI 1.0 only accepts authorization codes that are used once. Further, PKCE is enforced, and a nonce is included when the `openid` scope is present. If this is not the case, the state parameter is included.

Countermeasures by FAPI 2.0 In FAPI 2.0, the pre-registration of redirect URIs is not mandatory since they have to be included in the PAR. This ensures client authentication with the authentication request. FAPI 2.0 also only accepts authorization codes that are used once, and PKCE is required.

6.2.6 Eavesdropping Access/Refresh Token

Eavesdropping access tokens and refresh tokens is described in [15]. Access tokens and refresh tokens can be eavesdropped between the connection of AS and client, if no confidentiality protection is in place.

Access tokens should be treated as credentials, and therefore, the protection via TLS is essential. If not possible, the impact of a stolen access token should at least be reduced. Therefore, the scope and the token lifetime should be limited. The binding of the token to the respective `client_id` can further complicate the attack.

Countermeasures by FAPI 1.0 FAPI 1.0 enforces the use of TLS for all communication. It also reduces the impact of a stolen access token by limiting the token lifetime to a maximum of ten minutes. The advanced profile further complicates the attack by only issuing sender-constrained access tokens.

Countermeasures by FAPI 2.0 In FAPI 2.0, TLS must be used for all communication and ASs must support sender-constrained access tokens in both profiles.

6.2.7 Obtaining Access Tokens, Authorization Codes and Refresh Tokens from Authorization Server Database

Obtaining access tokens, authorization codes and refresh tokens from the authorization server database is described in [15]. Access tokens, refresh tokens and authorization codes can be disclosed, if stored as handles in a database by the AS. If an attacker has access to the database or can successfully perform a SQL injection attack, access tokens, refresh tokens and codes can be stolen.

Therefore, protection of the database system through system security measures, credential storage protection best practices, and SQL injection mitigations are necessary. Further, access tokens should only be stored as their hash values, and tokens should be bound to `client_id`s.

Countermeasures by FAPI 1.0 FAPI 1.0 does not specify system security measures or regulations for the storage of tokens. Anyhow, access tokens must be sender-constrained when using the advanced profile.

Countermeasures by FAPI 2.0 FAPI 2.0 also does not specify security measures regarding token storage or SQL injections. A sender-constraining mechanism for access tokens must be supported in both the baseline and the advanced profile.

6.2.8 Credential-Guessing Attacks

Credential-guessing attacks are described in [13]. An attacker might be able to guess credentials, such as access tokens, refresh tokens, authorization codes, and other credentials when they are chosen badly, e.g., too short. For example, an attacker can guess pairs of `client_id` and `client_secret`, if the entropy of secrets was chosen too low [15]. This is especially crucial for the entropy of symmetric keys, as described in [7]. The secret must have a high enough entropy to enable the generation of strong key material. This ensures an adequate level of security provided by keys derived from the `client_secret`. In the same way, authorization codes can be guessed by an attacker, enabling them to exchange them for a valid access token and a refresh token [15]. Further, refresh tokens values can be guessed to get valid access tokens.

To protect against the guessing of access tokens, refresh tokens, authorization codes, and other credentials, an attacker's success probability must be reduced as much as possible. Therefore, tokens and other secrets should have a high entropy or should be signed [15]. The token lifetime can be limited. Further, the use of strong client authentication can prevent this form of online guessing. Additionally, accounts can be locked after a certain number of unsuccessful login attempts. Binding the code to the redirect URI and the refresh token to the `client_id` further helps against guessing the authorization code or refresh token, as it increases the guessing challenge.

Countermeasures by FAPI 1.0 FAPI 1.0 dictates the AS to offer non-guessable access tokens, refresh tokens and authorization codes with a high entropy. When using symmetric keys, the client secret used must be at least 128 bits long and thereby adhere to the requirements stated in [7]. In the advanced profile, the use of symmetric key cryptography is not an option. Further, access tokens should only be valid for ten minutes. If the time is longer, they must be sender-constrained, which is mandatory in the advanced profile. Both measures reduce the attack impact of guessing a token. In the advanced profile, both authorization requests and responses are signed, which further mitigates the success of guessing attacks.

Countermeasures by FAPI 2.0 In the second version of the FAPI, sender-constrained access tokens are mandatory in both profiles. It also states to only issue credentials with an entropy of at least 128 bits. In FAPI 2.0 the use of symmetric key cryptography is not an option. With JAR and JARM, both authorization request and response are signed in the advanced profile.

6.2.9 Credential Leakage via Browser History / Log Files

Credential leakage via the browser history or via log files are described in [10] and in [15]. Credentials, such as authorization codes and access tokens can leak through the browser history if an attacker has physical access to the respective device. This results in authorization code replay attacks as well as in unauthorized access to protected resources. Authorization codes can leak via the browser history when the redirect to the redirection endpoint contains the code value. Access tokens can leak via the browser history or log files when sent in query parameters.

Mitigations include employing authorization code replay prevention and using the form post response mode as an alternative to redirecting. Additionally, access tokens should not be transmitted as query parameters. Therefore, using the authorization code grant and the form post response mode is recommended. Countermeasures include employing authorization headers and post parameters. Additionally, the logging must be sufficiently configured and authenticated requests can be enforced. Generally, limiting the access token scope and lifetime and making them single-use lowers the attack's impact.

Countermeasures by FAPI 1.0 In FAPI 1.0, access tokens must not be sent in query parameters, but as HTTP headers. Further, strict access control to logs is recommended. Access tokens are limited to ten minute lifetime and are sender-constrained in the advanced profile. Authorization codes are single-use.

Countermeasures by FAPI 2.0 Authorization codes are single-use. If this is not possible, their lifetime should be restricted to one minute. FAPI 2.0 does not allow sending access tokens in query parameters. Sender-constraining access tokens must be supported in both profiles.

6.2.10 Credential Leakage via Referrer Headers

Credential leakage via referrer headers is described in [10]. Credentials, such as authorization codes, state values and access tokens (transmitted in URI fragments), can be leaked through referrer headers from the client or AS websites. Attackers knowing these values can use those for the following attacks.
The client leaks credentials when rendering a specific page after a successful authorization response. This page either includes links to attacker pages on which a user clicks, or third-party content, e.g., advertisement. Then, attackers obtain the authorization response URL to get the code and state values. The AS also leaks credentials, in cases where the authorization endpoint includes either links or third-party contents.

To mitigate credentials leakage via referrer headers, both the authorization endpoint and the clients landing page after a successful authorization response should not contain external links or third-party resources. Further, methods that complicate the attack are suppressing the referrer header through a referrer policy and using the authorization code grant. Moreover, the binding of the authorization code to either a confidential client or a respective PKCE challenge will force the attacker to guess an additional value. Another mitigation is to make the authorization code single-use and revoke all tokens issued by that code if it is used twice. Similarly, the state value should be single-use and made invalid after being used at the redirection endpoint. Alternatively, the form post response mode can be used to avoid the risks of redirection for the authorization response.

Countermeasures by FAPI 1.0 FAPI 1.0 mandates the use of the authorization code grant or the hybrid flow with the response type `code id_token`. Further, the use of PKCE is mandatory, and authorization codes are single-use.

Countermeasures by FAPI 2.0 FAPI 2.0 only supports the authorization code grant and enforces the use of PKCE. Authorization codes are single-use and with PAR, client authentication can happen early on.

6.2.11 Insufficient Redirect URI Validation / Redirect URI Manipulation

Insufficient redirect URI validation is described in [10]. When ASs accept the registration of redirect URI patterns, they might be vulnerable to insufficient redirect URI validation. The validation logic is complex to implement correctly. Patterns can allow the use of wildcards, even allowing characters that are normally not valid for domain names. If the pattern validation logic is not implemented correctly, attackers can obtain authorization codes and access tokens. An example is the authorization code redirection URI manipulation, described in the OAuth Framework [13]. In the attack, an attacker has the ability to change the `redirect_uri` to their desire. They can lead the AS to redirect the UA to their chosen URI. Since they can choose an URI in their control, they can obtain the authorization code sent in the redirected authorization response. Therefore, they can send the UA to a URI in their control.

Countermeasures include simplifying the validation logic to the extent that only exact string matching of preregistered redirect URIs is allowed. Further, open redirectors and the reattaching of fragments to redirect URIs should be prevented. Additionally, issuing of access tokens at the token endpoint is recommended. An alternative to limit the danger of insufficient redirect URI validation is the use of JAR or PAR since both provide integrity and origin verification mechanisms. Therefore, the

AS can trust the known clients and the redirect URI included in the authorization request.

Countermeasures by FAPI 1.0 In the FAPI 1.0, the use of preregistered `redirect_uris` is mandatory. ASs and clients must validate and compare them to the used URIs. Further recommendations regarding the prevention of open redirectors are mandatory. PAR can be supported, and JAR is mandatory when using the advanced profile. Additionally, access tokens are only issued at the token endpoint.

Countermeasures by FAPI 2.0 In the second FAPI profile, the use of PAR is mandatory. Further, JAR is also essential when using the advanced profile. Redirect URIs are included in the PAR, the exposure of open redirectors is forbidden, and only the authorization code grant is allowed.

6.2.12 307 Redirect

The threat of the 307 redirect is described in [10]. After the AS successfully authenticated the end-user, they are redirected back to the redirection endpoint of the client. According to the OAuth 2.0 Framework, HTTP status code 307 is also allowed, resulting in the transmission of the user credentials from the AS to the client. In case of a malicious client, it can impersonate the end-user by utilizing the received credentials.

Therefore, HTTP status code 307 should not be used for redirection but instead 303. With this code, HTTP POST requests are always rewritten to HTTP GET requests resulting in the user credentials not being transmitted in the body.

Countermeasures by FAPI 1.0 FAPI 1.0 does not specify any rule regarding the redirection HTTP status codes.

Countermeasures by FAPI 2.0 Contrary, FAPI 2.0 forbids the use of the 307 HTTP status code, while the use of 303 status code is highly recommended.

6.2.13 Open Redirection

The threat of open redirectors is described in [13], [15], [10] and [11]. Open Redirectors are endpoints that automatically redirect an UA to a location specified in a query parameter. The problem is that the location is not validated, resulting

in open redirectors that can be utilized for further attacks, such as phishing attacks. Open redirectors combined with only partly registered redirect URIs allow the attacker to bypass the AS validation. Thereby, the attacker transmits tokens or the authorization code to themselves. It further might lead to the abuse of URLs that point to the client but lead to a phishing site. ASs can be abused as open redirectors by changing the redirect URI at the authorization endpoint. Open redirectors can also be used for phishing attacks in this case since the victim trusts the AS.

To prohibit open redirectors, ASs and clients should validate the query parameter's value or register the complete redirect URI. Alternatively, clients can follow redirects if they can prove its origin and integrity of it. ASs should only accept fully registered redirect URIs and not automatically redirect without verification of the redirect URI and the respective client id. In the case of an untrusted URI, an AS can delegate the trust decision to the RO.

Countermeasures by FAPI 1.0 The first FAPI specification prevents this attack by enforcing the pre-registration of redirect URIs and the exact matching of the `redirect_uri` at the AS. Further, both public and confidential clients have to store and compare the `redirect_uri` of the end-user's session with the URI the authorization response was received at. If they are not similar, the flow is stopped.

Countermeasures by FAPI 2.0 The newer FAPI specification prohibits exposing open redirectors at ASs and clients. Compared to FAPI 1.0, preregistered redirect URIs are not necessary, since they are included in the PAR, which provides client authentication.

6.2.14 DoS attacks

DoS attacks are described in [15]. ASs can be attacked by requesting a great number of authorization codes and access tokens. Further, DoS attacks that exhaust resources are only possible if there is no user intervention.
Attackers in control of a botnet can find clients redirect URIs. Then they can send manufactured, random authorization codes and cause a great number of connections on a single AS. The clients redeeming the codes, "protect" the attackers identity.

To prohibit the first kind of attack, the number of valid tokens per user can be limited, and authorization codes should have sufficient entropy. Countermeasures against manufactured authorization codes include using the `state` parameter. ASs can stop the attack, by blocking requests from clients that send too many wrong authorization codes.

Countermeasures by FAPI 1.0 Since the first FAPI specification enforces sufficient entropy for authorization codes and access tokens, DoS attacks aiming to cover all possibilities are mitigated. Further, the specification dictates using the state parameter (when `openid` is not requested).

Countermeasures by FAPI 2.0 The second FAPI sepcification also ensures a high entropy of authorization codes and access tokens. State protection in FAPI 2.0 is replaced by PKCE.

6.2.15 Clickjacking

Clickjacking is one of the attacks, explained in [13]. Clickjacking describes the attack of tricking a user into clicking on visible buttons, which are positioned under an invisible iframe. The iframe includes the authorization endpoint webpage. Dummy buttons visible for the user are directly under important buttons such as "Authorize!". The victim believes they clicked on a cute cat video, but in reality, they clicked on the invisible button, granting access to the attacker's client. Further, clickjacking can be used to obtain the ROs credentials, change the scope or access the protected resources.

A mitigation for this attack is the use of external browsers when native apps request user authentication. ASs can forbid iframes by setting the `x-frame-options` header, either to `deny` or to `sameorigin`. While the first prohibits any iframing and the latter only allows framing by sites with the same origin. For older browsers, JavaScript frame-busting methods can be used instead. Additionally, ASs are recommended to use Content Security Policies (level 2) on the authorization endpoint and other endpoints responsible for authentication and authorization.

Countermeasures by FAPI 1.0 FAPI 1.0 does not define explicit countermeasures against clickjacking.

Countermeasures by FAPI 2.0 FAPI 2.0 does not define explicit countermeasures against clickjacking.

6.3 FAPI-specific Attacks

The attacks in this section were presented by Fett et al. [59]. They are known to be successful on the FAPI 1.0. The authors name countermeasures for most of them. However, these countermeasures have not all been added to the FAPI specifications yet.

6.3.1 PKCE Chosen Challenge Attack

The PKCE chosen challenge attack is described in [59]. The "PKCE Chosen Challenge Attack" circumvents the authorization code protection that should normally be provided by PKCE. The attack scenario includes the installation of an honest and a malicious client app on the end-users device. Thereby, the honest app is a public client, registered at the honest AS, while the malicious one is not. The attack starts with the malicious app requesting the user to log in. It then sends an authorization request to the honest AS, containing both `client_id` and `redirect_uri` of the honest app, and the calculated `code_challenge` of a chosen `code_verifer`. The AS redirects the user to the honest app after processing the request and after successful end-user authentication. The authorization response might leak to the attacker, including the authorization code. As the attacker knows the `code_verifier` and does not need to authenticate at the server, they can exchange the obtained code at the honest AS against a valid access token to the end-users protected resources. The AS does not require client authentication since the honest app is a public client.

The only possibility to mitigate this attack is, binding the `code_challenge` to the `client_id` of the honest client, that created the `code_verifier`. They can be bound together by signing the authorization request object, as specified in JAR. This solution expects public clients to have the ability to store secrets.

Countermeasures by FAPI 1.0 The FAPI baseline 1.0 profile is vulnerable to this attack. The advanced profile mitigates this attack by only allowing confidential clients that make use of JAR to protect the authorization response.

Countermeasures by FAPI 2.0 FAPI 2.0 is not vulnerable to this attack, since only confidential clients are allowed and the use of PAR is mandatory in the baseline and advanced profile. PAR allows client authentication in an early state.

6.3.2 Cuckoo's Token Attack

In the formal security analysis of the FAPI 1.0, published by Fett et al., the so-called "Cuckoo's Token Attack" is introduced [59]. This attack is directed against the binding of access tokens in the FAPI advanced profile. Sender-constraining access tokens might prevent an attacker from redeeming the token themselves, but it is still possible to abuse a client for redeeming the token.
The attack scenario can either be a hybrid flow with an ID token as a detached signature or an authorization code flow with JARM. In the scenario, the attacker takes on the role of an end-user and of a malicious AS. The attack assumes that the attacker already phished a certificate bound access token, bound to a client (victim).

The client victim has to support multiple ASs, including the malicious one. Further, the client must be configured to use the same honest RS with both the malicious and the honest AS. The token is issued by an honest AS for accessing the resources of an honest end-user. First, the attacker starts the flow in the role of an end-user, who pretends to authorize the client (victim) at the attacker AS. The client thereupon sends the authorization request to the AS. The AS directly responds with the authorization response since user authentication is not needed with the attacker controlling both user and AS. After the token request, the malicious AS returns the phished access token in the token response. This token is bound to correct client, which does not notice a difference between a flow with an honest AS. For that reason, the client normally redeems the access token at the RS and receives the protected resources of the honest user the access token was "stolen" from. The client returns the information to the attacker (in the role of the end-user) without knowing that the users do not match. The attacker now has access to the protected resource, even in the presence of certificate-bound access token.

A protection measure against this attack is sending the AS's identity, which issued the token, in the resource request. This enables the RS to check for the correct AS/sender before returning protected resources.

Countermeasures by FAPI 1.0 FAPI 1.0 does not name explicit countermeasures for this kind of attack.

Countermeasures by FAPI 2.0 FAPI 2.0 does not name explicit countermeasures for this kind of attack.

6.3.3 Access Token Injection with ID Token Replay

This attack is another attack on the binding of access tokens in the FAPI 1.0 advanced profile. It is also described in [59]. In this scenario, the attacker has already obtained an access token, bound to a certain client. The access token is issued by an honest AS for access to an honest user's resources. Further, the token endpoint of the target AS is misconfigured to redirect to an attacker-controlled URI. The attacker starts the flow as the end-user at the honest but misconfigured AS. After the authorization request and successful user authentication, an ID token is returned in the authorization response in case of the hybrid flow. Now the client sends the token request to the misconfigured token endpoint in control of the attacker. The attacker then directly responds with the previously phished access token and an ID token since client authentication of further proof is not necessary for the attacker. Thereby the ID token is the one that was transmitted through their user-agent in the authorization response. The client checks according to the requirements of the profile whether certain ID token values are the same, which is the case since both

tokens are the same. Finally, the client redeems the sender-constrained access token at the resource server, which responds with the protected resources of the honest user (victim). This way, the client allows the attacker access to the protected resources.

This attack can be mitigated by including the hash of the access token in the ID token returned at the token endpoint. This mitigation works for both the hybrid flow, using ID token as detached signatures, and the authorization code flow, using JARM.

Countermeasures by FAPI 1.0 The advanced profile names the possible inclusion of a `c_hash` and `at_hash`, when OIDC is used.

Countermeasures by FAPI 2.0 FAPI 2.0 does not specify direct countermeasures.

6.3.4 Authorization Request Leak Attacks

The leakage of authorization requests is an assumption of the FAPI attacker model, made by Fett et al. in [59]. This leads to the exposure of the `code_challenge` and the `state` values. Attackers, in possession of the `state` value, can circumvent the CSRF protection and with that break session integrity. Session integrity is broken by logging in the end-user under the attacker's identity. Authorization request leak attacks are already known, but the FAPI and with that PKCE fail to protect against it.

With the strong FAPI attacker model, CSRF attacks have to be mitigated to prevent this attack. Session integrity is only proven by Fett et al. for clients using `OAUTHB` [5], a deprecated attempt on token binding. For every other client, session integrity has not been proven with an attacker model of this strength.

Countermeasures by FAPI 1.0 Although FAPI 1.0 enforces the implementation of an effective CSRF protection, Fett et al. were not able to prove session integrity. FAPI 1.0 with MTLS for token-binding is vulnerable to authorization request leak attacks.

Countermeasures by FAPI 2.0 For FAPI 2.0, session integrity was not proven yet.

6.4 Attacks on CIBA

The presented attacks on the Client-Initiated Backchannel Authentication flow originate in the security considerations of the OIDC CIBA specification [54] and the FAPI CIBA profile draft [53]. Attacks named in the sections above might be applicable on the CIBA flow as well.

6.4.1 CIBA Injection Attack

Injection attacks on CIBA are described in [54]. If the `login_hint_token` parameter is not signed in the authentication request, the sender of the hint cannot be identified. This may lead to injection attacks and the collection of user identifiers.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profile recommends not to use the `login_hint_token` for the transmission of authorization metadata.

6.4.2 Backchannel Client Notification Endpoint Confusion

This threat is described in [54]. If the `backchannel_client_notification_endpoint`, defined during registration, is in control of an attacker, the AS might send the authentication results directly to them.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profile strongly recommends the use of the `poll` mode, instead of the `ping` or `push` mode, for which this endpoint is needed. The `push` mode is strongly forbidden, while the `ping` mode is tolerated.

6.4.3 Obtaining Access Token and Manipulating Values in Push Callback

This threat is explained in [54]. Since the CIBA flow in the `push` mode is not based on redirecting, tokens can be obtained and manipulated when directly sent to the client's notification endpoint. Hashes of access tokens and refresh tokens are included in the signed ID token to mitigate this attack. Sender-constraining access tokens also reduces the attack's impact.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profile additionally requires the use of sender-constrained access token and prohibits the use of the `push` mode.

6.4.4 Authentication Sessions Started Without a Users Knowledge or Consent

This threat is described in [53]. In the Client-Initiated Backchannel Authentication flow, the client starts the authentication process. Since the end-user is not involved in the starting process, an attacker can start sessions without the user's knowledge. The threat is even more prominent in cases where the `login_hint` is a known identifier, for example a phone number.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profiles recommends the use of a nonce as a `login_hint` or the alternative acceptance of `id_token_hints`. In that case, the ID token must be obtained appropriately. Additionally, the `user_code` mechanism can be used as spam protection.

6.4.5 Reliance on User to Confirm Binding Messages

This threat is described in [53]. Attackers might initiate a malicious flow with an honest RO's identifier. At the same time, the honest end-user participates in an honest flow. In cases where both requests have the same scope, the only difference exists in the `binding_message` which has to be checked by the RO. This makes the validation of the `binding_message` a single point of failure.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profile discusses this issue and recommends alternative measures to facilitate the manual verification of this message. For example, scanning a QR code instead of manual checking.

6.4.6 Loss of Fraud Markers to OpenID Provider

This threat is explained in [54]. The CIBA flow is not redirect based. This decreases data that may help detect fraud.

Countermeasures by the FAPI 1.0 CIBA profile The FAPI CIBA profile names this threat and states the possible lack of fraud detection. Thereby it raises awareness.

7 Evaluation

To evaluate the security of the financial-grade API, we will sum up the results of the comparison, performed in chapter 4, the practical experience and impressions gained through the set up described in chapter 5 and the outcomes of the security catalog in chapter 6. Thereby, we aim to answer the research questions: *which concepts enable the high-security of the FAPI*, *which known attacks on OAuth and OIDC are applicable on the FAPI* and *whether the FAPI provides security improvements to classic OAuth and OIDC flows.*

7.1 Evaluation of the Comparison Results

Comparing the FAPI to "conventional" OAuth and OIDC implementations shows that the FAPI specifications offer more comprehensive profiles. The profiles thereby also include recommendations regarding cryptography, redirection, and communication between client and RS. These topics are not discussed in the compared frameworks. One benefit of creating profiles is interoperability. All four specified profiles do not leave a lot of choices.

When comparing the FAPI to "conventional" OAuth and OIDC implementations, one has to consider the different underlying attacker models. The FAPI has an attacker model that is stronger than both the original and the updated OAuth 2.0 attacker model. If one only considers the similar capabilities of the updated attacker model and the FAPI attacker model, OAuth 2.1 and the FAPI specifications have many commonalities. However, the FAPI aims to protect against a stronger attacker by employing non-repudiation features, such as JAR, JARM and PAR. Further, the FAPI specifications cover a broader scope of security measures as specified above.

Additionally, the FAPI defines a security level that is secure at the current time. It includes many up-to-date security features that were not available at the time of the publication of the original OAuth framework. This leads to the question of how long the FAPI will remain secure or whether such a profile creation only lasts a few years until newer specifications are needed. Older profiles may then give a false sense of security. The development of the FAPI 2.0 so quickly after the publishing of the first one does not promise a sustainable permanent solution.

Regarding the FAPI CIBA profile, one can say that it ensures a secure configuration of the OIDC CIBA specification. The FAPI CIBA profile results not only in a more secured communication but also in better interoperability. The FAPI also provides extensive regulations regarding TLS and algorithm considerations. However, the FAPI CIBA profile is still in a draft state, meaning that it might change in the future. Further, it should be discussed whether supporting all regulations of the main FAPI profiles is necessary and appropriate in the CIBA context.

7.2 Impressions of Real-World implementations

Working with the oxd Client API gave us an insight in setting up and configuring a FAPI-certified client. First, it is noticeable that far less FAPI-clients, than FAPI-ASs are certified. Moreover, the certification test suite does not cover FAPI system regulations, such as strong access control to the logs. There is no certification for RSs or for the FAPI 2.0 drafts. Configuring the oxd Client API to be FAPI-compliant was not trivial. Gluu does not provide explicit documentation on how to make the oxd Client API FAPI/compliant. Further, we noticed that the oxd Client API cannot be FAPI-certified on its own. Due to it's design, the oxd Client API cannot function as a standalone client. An additional client application is needed, which has to validate the authorization response. The oxd Client API does not offer a mechanism for validating the authorization response. Thereby, the ID token as a detached signature is not validated by the FAPI-certified client implementation. Furthermore, the design of the oxd Client API is not standard-conform. Dividing the client into two parts might increase the potential attack surface. The communication between the oxd Client API and one or more client application(s) can be further protected by an additional token and an ID. However, the design of the oxd Client API is not explicitly prohibited by the FAPI.

7.3 Evaluation of the Security Catalog

The security catalog shows that the FAPI enforces protective measures against most of the known attacks on OAuth and OIDC. The attacks are described in security considerations of the protocol specification, the best current practices [10], and the OAuth 2.0 threat model [15]. Several threats such as clients obtaining scopes without RO authorization (see subsection D.4.8) are only applicable on flows, that are strictly forbidden in the FAPI. Generally, typical FAPI countermeasures, such as sender-constrained access token, the enforcement of client authentication, and the presence of PKCE parameters and state of nonce values complicate many attacks and limit their effectiveness. For example, stolen access tokens that are sender-constrained can not be redeemed that easily as bearer tokens. Signing authorization requests and

response is also an effective measure. Especially the enforcement of PAR seems to be a very effective addition to FAPI 2.0. PAR offers many advantages but especially enables authentication of the client early on.

However, FAPI 2.0 does not yet implement protective measures against the attacks, described by Fett et al. [59] in their analysis of FAPI 1.0. These attacks include "Cuckoo's Token Attack" (subsection 6.3.2), "Access Token Injection with ID Token Replay" (subsection 6.3.3) and the "PKCE Chosen Challenge Attack" (subsection 6.3.1), as well as "Authorization Request Leak Attacks" (subsection 6.3.4). All these attacks are only successful in the presence of several prerequisites and a strong attacker. Regarding "Authorization Request Leak Attacks", no evaluation for DPoP was performed at the time.

Moreover, in some cases such as timing attacks (see subsection D.1.4), or manipulation of scripts (see subsection 6.1.4), the FAPI only enforces a part of the recommended countermeasures. In other cases, the FAPI does not enforce any mitigations, for example in the phishing category (see subsection D.2.3). The specifications do not state to inform the user about the potential threat of phishing attacks. However, the FAPI could define more countermeasures regarding the complete system, apart from the authorization flows. FAPI 1.0 seems to provide more of these system security measures, including logging recommendations. On the other hand, FAPI 2.0 does not yet offer general protection measures for the entire system. Recommendations for this might follow, since FAPI 2.0 is currently under active development.

8 Conclusion and Future Work

In this thesis, the financial-grade API was systematically analyzed. First, we gave an introduction to the underlying fundamentals: OAuth and OpenID Connect. Then, we outlined security extensions employed by the FAPI. We explained the FAPI profiles and highlighted their differences. Additionally, we compared the security features used by the FAPI to features used by OAuth and OIDC. The resulting tables provide a helpful overview for future analysis. Further, we described the different attacker models and put them into perspective. Then, we set up a FAPI-certified client and AS and created a testing tool. The tool ensures the functionality of the setup and enables further manual testing in the environment. Moreover, we created a security catalog, including known attacks on OAuth and OIDC and the FAPI's countermeasures. The catalog provides a reference for further practical analysis. Finally, we evaluated the results of the comparison, the setup, and the security catalog. The core results of our analysis are summed up below, answering our research questions.

Which concepts enable the, according to the FAPI Working group, high-security level of the FAPI? The FAPI aims to realize a high level of security through the enforcement of several techniques. First, the FAPI forbids configurations and flows that are known to be insecure, such as the use of the RO password credentials grant or insecure cryptographic algorithms. Additionally, the profiles ensure adherence to security best practices, such as the mandatory use of PKCE, issuing access tokens with a short lifetime, using client authentication for confidential clients, or including state or nonce values. To further protect high-risk APIs, the FAPI mandates the use of sender-constrained access tokens, and non-repudiation features, such as JAR, JARM and PAR.

Do the FAPI profiles provide security improvements compared to the classic OAuth and OIDC flows? In conclusion, FAPI 1.0 offers two security profiles for different needs. Both profiles ensure the use of known security measures and offer interoperability. The correct use of the FAPI is supported through the certification offered by the OpenID Foundation. The security level provided by the FAPI can be achieved by employing several OAuth and OIDC extensions. Nevertheless, the FAPI limits decision making, which mostly mitigates poor decisions and provides interoperability. Moreover, it serves as a collection of well-working security mechanisms that facilitate the protection of OAuth and OIDC implementations. Both

FAPI versions and their profiles do not introduce new security mechanisms in their specifications. However, the development of the FAPI has led to the development of JARM, PAR and RAR [20]. Furthermore, the FAPI includes many up-to-date security features that were not available at the time of the publication of the original OAuth framework. However, with OAuth 2.1, the new "classic" OAuth flows are closer to the security provided by the FAPI.

Which attacks are applicable on FAPI implementations - clients and providers? The FAPI is proven to be theoretically secure, shown by Fett et al. [59]. For the remaining successful attacks, the authors recommend possible countermeasures. These are not yet implemented. These successful attacks do not have a significant impact on the security the FAPI provides since strong attackers are needed, and many prerequisites have to be fulfilled. However, the current drafts of FAPI 2.0 indicate even more secure profiles yet lack specific system-related security recommendations. These might follow in the future since the drafts are in an early stage of development.

Future Work may focus on the practical analysis of FAPI implementations. The testing tool we have created can easily be extended to test other configurations and implementations. The mitmproxy addon framework enables simple adaptation to the particular implementation. Thereby, the FAPI-certified implementations can be the starting point.

For testing the oxd Client API, the configuration of MTLS for client authentication and token binding must be pursued. At this point in time, we were not able to verify that the oxd Client API uses MTLS, even though the MTLS flags are set. We configured mitmproxy to request client authentication by patching the current release. The patched mitmproxy now sends a certificate request to the oxd Client API, which can be observed by using *Wireshark* [52]. However, we were not able to observe the expected responses.

The certification implies certified security, but the confirmation test suites might not cover everything a manual analysis might find. Thus, the certification test suites should be evaluated. They can be misleading and implementers might rely on their promises.

Another focus may lie on the security extensions employed by the FAPI. Several of the extensions are relatively new and might not have been analyzed properly yet, for example RAR, PAR or DPoP.

Additionally, the FAPI drafts that are currently under development should be both theoretically and practically analyzed once finalized. FAPI drafts include the FAPI 2.0 drafts and the FAPI CIBA draft.

A Security Extensions

A.1 JWT-Secured Authorization Request (JAR)

The JWT-Secured Authorization Request (JAR) was introduced with RFC9101 [32] in August 2021. JAR extends the OAuth 2.0 authorization framework, so that authorization request parameters can be encoded in a JWT. Thus, authorization requests can be integrity protected by signing the JWT with JWS and encrypted by using JWE. The original OAuth 2.0 framework does not provide these additional integrity and confidentiality measures since the authorization request parameters were directly sent through the UA, using query parameter serialization. Encoding the request parameters in the URI also leads to source authentication failure, and application layer security can not be used. The standardization introduces two new parameters: the `request` parameter, holding the JWT-encoded request parameters, also called the "request object", and the `request_uri`, also called the "request object URI", which replaces the `request` parameter, when passing the request by reference.

A.2 Pushed Authorization Request (PAR)

RFC9126 was published in September 2021 and specifies "OAuth 2.0 Pushed Authorization Requests" [33]. With the PAR specification, clients can push the authorization request content to the AS. Then the AS responds with a `request_uri` parameter referring to the pushed data, which is then used by the client for the subsequent authorization request. Pushing the authorization request has various advantages since authorization requests can become quite large, and URLs have a length restriction. Further, the PAR extension can ensure integrity and authenticity protection as well as confidentiality. The Pushed Authorization Request completes the JAR standardization, since it realizes the "pass by reference" mechanism, which uses the `request_uri` parameter. PAR also implements client authentication at the AS, before any user interaction happens. The AS must validate the pushed authorization request as if it would be a regular authorization request and authenticate the client as at the token endpoint. A successful response to a pushed request contains not only the `request_uri` but also a `expires_in` parameter, specifying the lifetime of the URI.

A.3 Rich Authorization Request (RAR)

The 10th version of the "OAuth 2.0 Rich Authorization Requests" draft was published in January 2022 and defines the new authorization request parameter `authorization_details` [27]. This parameter helps to translate a high demand for authorization data. This high demand is mainly present in the financial context since transaction requests may include numerous dynamic details. For example, transaction details such as the IBAN, the names of sender and receiver, the amount of money, and the currency. This scenario can hardly be realized only using the `scope` parameter. Therefore, the `authorization_details` parameter was introduced. Fine-grained authorization details are transmitted using an array of JSON objects. Every object contains a `type` field, holding a string, which defines the type of the details and, with that, the allowed object contents. Examples for types can be `account_information` or `payment_initiation`. It is necessary to obtain user consent over the specified authorization details.

A.4 JWT-Secured Authorization Response Mode (JARM)

The second draft of the "Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)" [2] was published in October 2018 by the OpenID Foundation. The standardization protects the authorization response by introducing a JWT-based mode, where the response parameters are encoded as a JWT. JWTs enable the possibilities of signing and encrypting the authorization response parameters. Further it enables sender authentication and audience restriction. Besides the authorization request parameters, the JWT response document also includes the `iss`, `aud` and `exp` claims. The claims further secure the transmission by specifying the respective AS, the `client_id` and the expiration of the token. The authorization response JWT contains response parameters according to the respective, requested `response_type`. Further, the standard differentiates between three different ways of response encoding modes: `query.jwt`, `fragment.jwt`, `form_post.jwt`. They all send the JWT in the defined `request` parameter of the response. A shortcut to select the default response mode belonging to the respective response type is called "jwt".

A.5 Proof Key for Code Exchange (PKCE)

The "Proof Key for Code Exchange by OAuth Public Clients" was published in September 2015 [17]. It was created for securing public clients, which are not able to authenticate at the AS. The security measure aims to protect against the authorization code interception attack. In the attack, the authorization code leaks to an attacker, which then tries to exchange it against an access token. PKCE with

`S256` as the code challenge method provides protection even in cases where both authorization request and authorization response leak to the attacker. The measure binds the authorization request to the token request.

First the client creates a `code_verifier`, which is a random string. Then the `code_challenge` is calculated by hashing the verifier using a defined `code_challenge_method`. This method can either be `S256`, using SHA256 or `plain`, where both verifier and challenge are the same. Third, the client includes the verifier and the calculation method in the authorization request. Then, the AS associates the issued code with the received `code_challenge`. After receiving the authorization response, the client send the token request, including not only the authorization code but also the `code_verifier`. The AS can then apply the stated `code_challenge_method` on the received `code_verifier` to compare the result with the `code_challenge` associated to the respective code. If the values are the same, the AS can issue an access token, knowing that it deals with the same party that sends the authorization request. An attacker that knows both authorization request and response can not redeem the code since they do not know the `code_verifier`.

A.6 Mutual TLS (MTLS)

In RFC8705 [31], both Mutual TLS (MTLS) for client authentication and certificate-bound access token is defined.

Client authentication performed on the transport layer is more secure than sharing secrets, as specified in OAuth. MTLS can be used for client authentication at all endpoints, where it needs to be performed: at the token, introspection, revocation, and the backchannel authentication endpoints. In the standardization, there are two ways of client authentication using MTLS specified. First, the "PKI Mutual-TLS Method", and second the "Self-Signed Certificate Mutual-TLS Method". In both ways, the client proves possession of the key to the certificate. For every request sent with MTLS, the client has to include its `client_id`, so that the AS can easily locate the client configuration and check the presented certificate against the saved client credentials. If the client does not provide a certificate or the correct one, the AS returns an `invalid_client` error.

Token binding describes a technique in which the access token is bound to the client, meaning that it can only be redeemed at the resource server by the associated client. One way to achieve token binding is by using MTLS. Thereby, the client uses MTLS for connecting to the token endpoint of the AS, which is then able to bind the issued token to the client's certificate. The association is done by including the hash of the certificate in the access token. This way, the RS can verify the binding. An alternative is the use of token introspection. When requesting protected resources at the RS, the client has to use MTLS. The RS can then compare the hash of the certificate with the one received through the access token or token

introspection. In case of mismatching certificate, the RS returns an `invalid_token` error.

A.7 Demonstration of Proof of Possession (DPoP)

The sixth draft of the "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)" [34] standardization was published in March 2022 and describes a proof of possession way of token binding on the application layer. With DPoP, public and confidential clients can prove possession of a private key to redeem sender-constrained access tokens or refresh tokens. In contrast to MTLS, DPoP is not used for client authentication. Further, DPoP does not provide message integrity. For this purpose it relies on TLS.

The standardization introduces the DPoP proof, which is a signed JWT, including individual data regarding the HTTP request (e.g., a timestamp and a unique identifier) and the used signature algorithm as well as the public key of the client. The DPoP proof is created by the client and then attached to the token request as the `DPoP` HTTP header. Then the AS binds the access token (or a refresh token) to the client's public key, provided in the `DPoP` header. The token can only be used by the client, attaching the particular DPoP proof to the resource request to the RS. Since the token is bound to the client's public key, the RS verifies the DPoP proof by comparing the public key provided by the DPoP header to the one the token is bound to. This information can be included in the respective token or can be obtained at the token introspection endpoint. It further validates the signature and checks whether the access token hash, which must be included in the DPoP proof sent to the RS, matches the access token sent in the request. If one of the checks fails, the RS does not respond with the requested resources.

A.8 Signed JWT Introspection Response

The 12th version of the "JWT Response for OAuth Token Introspection" [55] draft was published in September 2021. It specifies a secure way of transmitting a token introspection response by encoding it as a JWT. With token introspection, a RS can verify the validity of a received token or obtain additional information regarding the token at the respective token introspection endpoint at the AS. The standardization enables the token introspection endpoint to return JWT-encoded responses. Therefore, ASs shall have the capability of identifying, authenticating, and authorizing RSs by holding credentials and configuration data for the individual RSs. This can be realized through treating RSs as clients and with that using dynamic client registration. The JWT response can be requested by the RS through sending the `Accept`

HTTP Header with the value "application/token-introspection+jwt". After authenticating the RS as a client, the AS responds with a JWT including the `iss`, `aud`, `iat` and the `token_introspection` claims. The latter includes the token introspection response members.

B Complete Tabular Comparison

General Configuration	FAPI 1.0 **CIBA-Draft**	OIDC **CIBA**	OAuth 2.0 **Device Grant**
Public Clients	#	x	x
Confidential Clients	x	x	x
Push mode	#	x	0
Poll mode	x	x	x
Ping mode	[x]	x	0
Client Authentication	**CIBA-Draft**	**CIBA**	**Device Grant**
client_secret_basic	#	x	x
client_secret_post	#	x	0
client_secret_jwt	x	x	0
private_key_jwt	x	x	0
none	#	#	#
MTLS	x	x	0
CIBA specific	**CIBA-Draft**	**CIBA**	**Device Grant**
Signed Backchannel Auth. Requests	x	[x]	0
Unsigned Backchannel Auth. Requests	#	x	x
return acr claim in IDT, if requested & supported	x	[x]	0
nbf claim / exp claim 60 min in Sign. Auth. Req.	x	0	0
request_context claim	[x]	0	0
login_hint; login_hint_token	(#)	[x]	0
user_code mechanism	[x]	[x]	x
binding_message / unique Auth. Context	x	[x]	0
not send x-fapi-customer-ip-address/x-fapi-auth-date headers	x	0	0
send metadata about consumption device	(x)	[x]	0
FAPI specific	**CIBA-Draft**	**CIBA**	**Device Grant**
PKCE	(x)	0	0
JAR	x	0	0
JARM	x	0	0
Sender-constrained AT (MTLS)	x	(x)	0
PAR	(x)	0	0
nonce	x	0	0

Figure B.1: Complete Comparison of the FAPI CIBA profile Draft, the OIDC CIBA extension and the OAuth 2.0 Authorization Device Grant. Key: 0 = not mentioned; # = explicitly not permitted; [x] = may/can support // named; (x) = should support; x = must support.

	FAPI 1.0		FAPI 2.0		OAuth		OIDC
General Configuration	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
Public Clients	(x)	#	0	0	x	x	x
Confidential Clients	x	x	x	x	x	x	x
Credentialed Clients	0	0	0	0	0	x	0
Authorization Code Flow	x	x	x	x	x	x	x
Hybrid Flow	0	x	#	#	0	0	x
Implicit Flow	0	#	#	#	x	#	x
Resource Owner Password Credential Grant	0	#	#	#	x	#	0
Client Credentials Grant	0	#	#	#	x	x	x
Refresh Token	x	x	x	x	x	x	x
Client Authentication	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
always client authentication of confidential clients	x	x	x	x	x	x	x
client_secret_basic	#	#	#	#	[x]	[x]	x
client_secret_post	#	#	#	#	[x]	[x]	x
client_secret_jwt	x	#	#	#	[x]	[x]	x
private_key_jwt	x	x	x	x	[x]	(x)	x
with private_key_jwt: include AS's issuer in aud claim	0	0	x	x	[x]	0	(x)
none	#	#	#	#	0	0	x
MTLS	x	x	x	x	[x]	(x)	0
with MTLS: support the mtls_endpoint_aliases metadata (RFC8705)	0	0	x	x	0	0	0
General Security features	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
PKCE	x	(x)	x	x	0	x/(x)	0
state	x	x	#	#	(x)	[x]	(x)
nonce	x	x	0	0	0	[x]	[x]/x
Signed ID Token	x	x	x	x	0	0	x
Encrypted ID Token	[x]	(x)	#	#	0	0	[x]
Sender-constrained Access & Refresh Token	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
only issue sender-constrained Access Token	[x]	x	x	x	0	(x)	0
MTLS for Access Token	0	x	x	x	0	(x)	0
with MTLS: compliancy to requirements in RFC8705	0	x	x	x	0	(x)	0
with MTLS: support the mtls_endpoint_aliases metadata (RFC8705)	0	0	x	x	0	0	0
DPoP for Access Token	0	0	x	x	0	0	0
DPoP for Refresh Token	0	0	x	x	0	(x)	0
with DPoP: compliancy to requirements in DPoP draft	0	0	x	x	0	(x)	0
Using JAR	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
support JAR	#	x	0	x	0	0	[x]
support signed JWT request objects at the PAR endpoint	#	0	0	x	0	0	0
pass signed JWT request object by value with request parameter	#	x	0	[x]	0	0	[x]
pass signed JWT request object by reference with request_uri parameter	#	x	0	#	0	0	[x]
request object with exp claim with lifetime <= 60 min after nbf claim	#	x	#	#	0	0	0
request object with nbf claim that is not longer than 60 min in the past	#	x	#	#	0	0	0
include all parameters in the signed request object	#	x	0	x	0	0	x
only process parameters transmitted in the signed request object	#	x	0	x	0	0	0
include response_type, client_id & scope in auth. request outside of request object (if not PAR)	#	x	0	#	0	0	0
Protecting the Auth Response	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
support JARM	#	x	0	x	0	0	0
using response type code and response mode jwt, when using JARM	#	x	0	0	0	0	0
creating jwt-secured auth. response compliant to JARM spec., 4.3	#	x	0	0	0	0	0
support ID tokens as detached signatures	#	x	0	[x]	0	0	[x]
using response type code ID token when using ID tokens as detached signatures	#	x	0	0	0	0	x
OpenID Connect is supported	#	x	[x]	[x]	0	0	x
support signed and encrypted ID Token	#	(x)	#	#	0	0	x/[x]
when using ID token as detached signature, include a s_hash	#	x	#	#	0	0	0
verify s_hash when ID token is used as detached signature	#	x	#	#	0	0	0
not return sensitive PII in ID token when sending it as detached signature in the front channel	#	(x)	0	0	0	0	0
if returning sensitive PII in ID token in front channel, encrypt ID token	#	(x)	0	0	0	0	0
Securing the token introspection response/userinfo response	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
support JWT introspection response	0	0	0	x	0	0	0
be compliant to RFC7662 - OAuth Token Introspection	[x]	[x]	0	x	0	[x]	0
sign token introspection responses, if token introspection is supported	0	0	0	x	0	0	0
request signed token introspection responses, if token introspection is supported	0	0	0	x	0	0	0
Signed/(&)Encrypted User Info Response	0	0	0	0	0	0	[x]
PAR	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Advance**	**2.0**	**2.1**	**1.0**
support PAR (endpoint)	0	[x]	x	x	0	[x]	0
reject common auth. requests	0	#	x	x	#	#	#
if PAR, use PKCE with s256 code challenge method	0	x	x	x	0	0	0
transmit client_id to auth. endpoint when using PAR	0	x	x	x	0	0	0
only accept client-authenticated PARs	0	#	x	x	#	#	#
require redirect_uri parameter in pushed auth. Requests	0	0	x	x	0	0	0
issue par-request_uri with expires_in values between 5 & 600 seconds	0	0	x	x	0	0	0
RAR	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
support RAR (authorization_details)	0	0	x	x	0	[x]	0
Secure Redirection	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
Pre-registration of redirect URIs	x	x	#	#	x/(x)	x	x
redirect_uris with https	x	x	x	x	0	x/[x]	(x)
individual redirect URIs for each AS	x	x	0	0	0	(x)	0
reject auth requests without redirect_uri	x	x	x	x	#	[x]	x
reject auth requests with redirect_uris that don't match the registered ones	x	x	#	#	x	x	x
store redirect uri In UA session (client) & compare with receiving address of auth response	x	x	0	0	0	[x]/x	0
not use HTTP status code 307 for redirects including user credentials	0	0	x	x	0	x	0

Figure B.2: First half of the complete Comparison of the FAPI 1.0, 2.0, OAuth 2.0, 2.1 and OIDC. Key: 0 = not mentioned; # = explicitly not permitted; [x] = may/can support // named; (x) = should support; x = must support.

use HTTP status code 303 for redirecting the UA	0	0	(x)	(x)	0	(x)	0
not expose open redirectors	0	0	x	x	[x]	x	0
Communication between Client and Resource Server	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
support HTTP GET method (RS)	x	x	0	0	0	0	0/(x)
support CORS (RS)	(x)	(x)	0	0	0	0	0/(x)
consider RFC6719 before allowing JavaScript Clients	(x)	(x)	0	0	0	0	0
send AT in the HTTP header	x	x	x	x	[x] typic.	(x)	0/(x)
don't sent AT via query parameters	x	x	x	x	0	x	0
ensure that AT is not expired or revoked, but valid	x	x	x	x	x	x	x
only execute protected actions in presence of a valid AT	x	x	x	x	x	x	x
identify entity linked to AT (RS)	x	x	x	x	0	0	0
check if authorization_details fit to protected resources of RS	0	0	0	0	0	[x]	0
only return explicitly granted resources	x	x	x	x	x	x	0
UTF-8-encode the resource response	x	x	0	0	0	0	0/(x)
use Content-type: application/json (RS)	x	x	0	0	0	0	0/x
include x-fapi-interactio-id response header	x	x	#	#	0	0	0
not dismiss requests with valid x-fapi-customer-ip-address header (RS)	x	x	#	#	0	0	0
include x-fapi-auth-dater header in resource request	[x]	[x]	#	#	0	0	0
include x-fapi-customer-ip-address header in resource request	[x]	[x]	#	#	0	0	0
include x-fapi-interaction-id header in resource request	[x]	[x]	#	#	0	0	0
User Contact	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
appropriate level of user authentication	x	x	0	0	0	0	x
specific user content to requested scope	x	x	0	0	0	0	[x]
clarify granting long-term access to end-user	(x)	(x)	0	0	(x)	0	(x)
offer consent revocation mechanism for access and refresh token	(x)	(x)	0	0	0	0	(x)
Strong Crypto	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
strong client secret	x	x	x	x	x	x	x
strong AT, RT and auth. codes	x	x	x	x	x	x	0
use PS256 or ES256 algorithms for JWS	0	x	x	x	0	0	#
not using algorithms using RSASSA-PKCS1-v1_5	0	(x)	(x)	(x)	0	0	#
not using none for jws	0	x	x	x	0	0	x
not using RSA1_5 algorithm for jwe	0	x	x	x	0	0	#
not using same kid for more than one key in a jwk	0	(x)	0	0	0	0	0
but if use kty, use or alg as additional identification	0	x	0	0	0	0	0
adhere to JWT BCP	0	0	x	x	0	0	0
implement ECDSA in a deterministic way	0	0	(x)	(x)	0	0	[x]
use rs256 (representing RSASSA-PKCS-v1_5)	0	#	0	0	0	0	(x)
use ES256	0	x	x	x	0	0	x
use rsa1_5	0	#	0	0	0	0	x
rsa keys with min. length of 2048 bit	x	x	x	x	0	0	0
ecc keys with min. length of 160 bit	x	x	x	x	0	0	0
Ensurance of secure transmission	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
encrypt all communication with TLS	x	x	x	x	(x)	x/(x)	x
when using TLS 1.2: TLS-compliance with BCP195/RFC7525	x	x	x	x	[x]	[x]	0
use TLS version 1.2 or later	x	x	x	x	[x]	[x]	[x]
perform TLS server certificate checks	x	x	x	x	[x]	x	x
assure that connections cannot be downgraded (endpoints used by web browsers)	[x]	[x]	x	x	0	0	0
for TLS versions below 1.3 (meaning TLS 1.2) only use four specified cipher suites	0	x	x	x	0	0	0
use the dhe cipher suites with key lengths longer than 2048 bits	0	x	x	x	0	0	0
use DNSSEC on all endpoints	[x]	[x]	(x)	(x)	0	0	0
serve jwks uri endpoints over tls	0	x	x	x	0	0	0
Enforcement of Security Best Practices	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
not accept previously used auth. Codes	x	x	x	x	x	x	(x)
issue bearer tokens with lifetime of < 10min	(x)	(x)	0	0	[x]	[x]	(x)
use refresh tokens for longer access, instead of long-langsting access token	(x)	(x)	0	0	[x]	[x]	[x]
return standard-conforming token responses	x	x	0	0	x	x	x
return list of granted scopes in token response, when token request through frontchannel	x	x	0	0	[x]	(x)	[x]
support OIDC Discovery	x	x	x	x	0	0	x
support OAuth authorization metadata	[x]	[x]	x	x	[x]	[x]	0
no other distribution of discovery metadata	x	x	0	0	[x]	#	0
implemetation of CSRF protection	x	x	0	0	x	x	x
use jwks uri endpoints for publishing public keys	0	(x)	0	0	0	0	x
not using x5u and jku JOSE headers	0	(x)	0	0	0	0	x
use PKCE with S256 code challenge method	x	x	x	x	0	x/(x)	0
include and check iss parameter in auth response to prevent mixup attacks	0	[x]	x	x	0	0	x
use refresh token rotation	0	0	[x]	[x]	[x]	(x)	0
support refresh tokens rotation	0	0	x	x	[x]	(x)	0
Misc	**Bas. 1.0**	**Adv. 1.0**	**Bas. 2.0**	**Adv. 2.0**	**2.0**	**2.1**	**1.0**
use certified FAPI-implementations	(x)	(x)	0	0	0	0	0
strong access control to logs	(x)	(x)	0	0	0	0	0
native apps follow BCP212/RFC8252	x	x	0	0	0	x	0
not support Private-use URI Scheme Redirection (native Apps)	x	x	0	0	0	#	0
not supprot Loopback Interface Redirection (native Apps)	x	x	#	#	0	#	0
support Claimed https Scheme URI Redirection (native Apps)	x	x	0	0	0	(x)	0

Figure B.3: Second half of the complete Comparison of the FAPI 1.0, 2.0, OAuth 2.0, 2.1 and OIDC. Key: 0 = not mentioned; # = explicitly not permitted; [x] = may/can support // named; (x) = should support; x = must support.

C Setup Code of the oxd Client API

```
FROM ubuntu:18.04
ENV DEBIAN_FRONTEND noninteractive
EXPOSE 8443

# https://www.gluu.org/docs/oxd/install/
ENV OXD_SERVER_HOME=/opt/oxd-server

RUN apt-get update && apt-get upgrade -y
RUN apt-get install -y --no-install-recommends \
    wget\
    bash \
    unzip

# install java deps
RUN apt-get install openjdk-8-jre -y --no-install-recommends

# install oxd-server
WORKDIR $OXD_SERVER_HOME
RUN wget https://ox.gluu.org/maven/org/gluu/oxd-server/4.2.1.Final/oxd-server-4.2.1.Final-
     distribution.zip -O tmp.zip
RUN unzip tmp.zip
RUN rm tmp.zip
COPY oxd-server.yml $OXD_SERVER_HOME/conf


WORKDIR $OXD_SERVER_HOME/bin
RUN chmod +x oxd-start.sh
COPY docker-entrypoint.sh $OXD_SERVER_HOME/bin/
ENTRYPOINT ["./docker-entrypoint.sh"]
```

Listing C.1: Dockerfile for oxd Client API.

D Additional attacks (Security Catalog)

D.1 General Attacks

D.1.1 Code Injection and Input Validation

Code injection and input validation attacks are described in [13]. Code injection attacks take advantage of unsanitized input variables to change the application logic. Attackers can access data on the device or cause other problems, such as denial of service.

Sanitization and validation of every value send to AS and client prohibits code injection. A special focus should be the `state` parameter and `redirect_uri` parameter.

Countermeasure by FAPI 1.0 FAPI 1.0 does not define protection measures against code injection.

Countermeasure by FAPI 2.0 FAPI 2.0 does not define protection measures against code injection.

D.1.2 TLS Terminating Reverse Proxies

The threat of TLS terminating reverse proxies is described in [10]. Often, application servers are hidden behind reverse proxies. These terminate the TLS connection. Reverse Proxies that transmit every header coming from the outside are vulnerable to attackers sending fake header values. If these values reach the protected application server, the attacker can circumvent security protection measures, such as IP whitelisting.

To protect against the risks of TLS terminating reverse proxies, the proxies have to sanitize any receiving requests and check the security-relevant header values. Further, it is essential to verify the authenticity of the communication since attackers with access to the internal network might bypass further security measures. Additionally, there must be protection against eavesdropping, injection as well as the

replay of messages in place for any communication between proxies and applications servers.

Countermeasure by FAPI 1.0 FAPI 1.0 enforces the protection of all communication by using TLS.

Countermeasure by FAPI 1.0 FAPI 2.0 also enforces the protection of all communication by using TLS.

D.1.3 Obtaining Client Secrets

The threat of an attacker obtaining client secrets is described in [15]. An attacker can replay or obtain access tokens, refresh token and authorization codes to access the victim's resources when gaining knowledge of the client secret. Client secrets can be obtained from the source code of an open source project or reverse-engineered from a binary.

To mitigate this client secrets should only be issued for confidential clients with a sufficient security level. Further, client secrets should be revocable and deployment-specific. In the latter case, the secret can be stolen from a web server or a native application. Therefore, web server protection measures should be applied or respectively the client secrets can be secured in the local storage of the native app.

Countermeasure by FAPI 1.0 FAPI 1.0 does not specifiy explicit protection measures against this threat.

Countermeasure by FAPI 2.0 FAPI 2.0 does not specifiy explicit protection measures against this threat.

D.1.4 Timing Attack

Timing attacks are described in [7]. Timing attacks originate in successful and unsuccessful decryption and signature verification execution times differing in length. A successful timing attack can facilitate breaking a cipher due to a reduction of the effective key length.

Simple mitigation against timing attacks is to not stop the process, as soon as an error occurs, but to continue until everything is processed.

Countermeasure by FAPI 1.0 The FAPI does not specify an explicit countermeasure against timing attacks, but the use of long keys can harden the attack surface.

Countermeasure by FAPI 2.0 FAPI 2.0 does also not specify a countermeasure against timing attacks, but the use of keys with a long length can make timing attacks harder.

D.1.5 Other Cryptography Related Attacks

The threat of cryptography-related attacks is described in [7] and [23]. Using JWTs comes with the risk of different crypto attacks. Therefore, implementers must be informed about the risks of using certain encryption and signature algorithms and further weaknesses specified in the JWT specification. JWS and JWE can only fulfill their security intentions if implemented securely.

The FAPI specifies certain security measures, to ensure a secure use of JWTs, JWSs, JWE. Therefore, only algorithms specified by the FAPI shall be used. In this context, the distribution of public key material is important. This can be implemented through the use of TLS-protected `jwks_uri` endpoints by ASs and clients. Alternatively, client can make use of the `jwks` parameter. When using JWK sets, keys should not have the same `kid`. If they do have the same, other attributes should be used to differentiate between keys. Otherwise, selecting the correct key to verify a signature is not possible.

Countermeasure by FAPI 1.0 The advanced profile dictates serving `jwks_uri` over TLS if used as recommended. Further, JWK sets are not recommended to have keys with the same `kid`. The advanced profile also specifies secure algorithms that must be used for JWE and JWS.

Countermeasure by FAPI 2.0 In the second FAPI version, the baseline profile specifies algorithm considerations used for JWTs.

D.1.6 Abuse of poorly configured TLS (and DNS) deployment

The importance of a secure TLS and DNS deployment is described in both FAPI 1.0 profiles [22] [23].

To prevent any kind of information disclosure and manipulation TLS must be used. TLS is only able to ensure confidentiality and integrity if used correctly and securely at the time. Therefore, TLS server certificate checks and the use of an (at the

time) secure cipher suite are essential. To prevent DNS spoofing attacks, the use of DNSSEC on all endpoints is recommended.

Countermeasure by FAPI 1.0 The FAPI 1.0 baseline profile enforces the use of TLS 1.2 or later and the performance of TLS server certificate checks. The TLS BCP [10] must be followed. Preloaded HTTP strict transport security policies are recommended to protect against TLS STripping attacks. DNSSEC should be used on all endpoints. The advanced profile additionally dictates explicit cipher suites that must be used with TLS 1.2 and defines certain key lengths necessary.

Countermeasure by FAPI 2.0 FAPI 2.0 explicitly specifies network layer protection measures, such as the mandatory use of TLS, its BCP, and the performance of TLS server certificate checks, and it further permits certain cipher suites and key lengths. The use of DNSSEC is recommended and protection measures against TLS stripping attacks are essential.

D.2 Attacks on Resource Owners

D.2.1 Resource Owner Impersonation at the Authorization Server

Resource owner impersonation at the authorization server is described in [15]. Attackers can impersonate the ROs and gain unauthorized access to their protected resources by simulating the necessary requests with the AS if the user authentication mechanism is not interactive or the flow is split up on different pages. RO impersonation is only executable by clients on the victims device (UA or native app).

Countermeasures include the use of interactive user authentication, e.g., through captchas, one-time secrets, or the combination of password authentication and consent. Additionally, the AS can message the ROs when access is granted in their names.

Countermeasure by FAPI 1.0 FAPI 1.0 does not explicitly defines protection measures against RO impersonation at the AS.

Countermeasure by FAPI 2.0 FAPI 2.0 does not explicitly defines protection measures against RO impersonation at the AS.

D.2.2 Client Impersonating Resource Owner at the Resource Server

The threat of a client impersonating a RO at the RS is described in the [10]. Based on the ROs identity, RSs can make access control choices. The ROs identity can be transmitted in the `sub` claim by an ASs token introspection response. A client that registers is able to register a chosen `client_id` can probably choose the same `sub` value as an end-user. This can lead to confusion a token received through a successful client credentials grant can be misunderstood as one granted by the respective user since the ids are similar.

Countermeasures to this kind of attack include limiting the clients power for choosing their ids. Additionally, it must be checked by the AS which access token was granted by an RO and which by a client.

Countermeasure by FAPI 1.0 In FAPI 1.0, the AS explicitly gathers user content and issues ID tokens that include a respective `sub` value.

Countermeasure by FAPI 2.0 FAPI 2.0 does not specify explicit countermeasures against client impersonating RO at the RS.

D.2.3 End-user Credentials Phishing

End-user credential phishing attacks are described in [13] and [15]. Redirect-based protocols can make the user insensitive regarding the severeness of them being redirected and prompted to enter their credentials. Attackers can start phishing attacks to obtain the credentials of the end-user. Therefore, attacker clients can use embedded or compromised browsers to change the visual trust mechanisms of the authorization website. Thus, end-users might not be able to tell the difference between an original website and a malicious one and enter their credentials into the faked website.

Generally, OAuth clients should never deal with end-user credentials, and hence developers should not collect user authentication information. Additionally, end-users can be informed about this kind of phishing and teach them to look for authenticity clues that can be provided by the client. Further, client applications can be verified before being published on a controllable market. Additionally, ASs require TLS on all endpoints where user interaction happens.

Countermeasure by FAPI 1.0 The FAPI 1.0 does not specify explicit protection measures against end-user credentials phishing, except the use of TLS for all endpoints.

Countermeasure by FAPI 2.0 The FAPI 2.0 does not specify explicit protection measures against end-user credentials phishing, except the use of TLS for all endpoints.

D.3 Attacks on Resource Servers

D.3.1 Access Token Redirect

The threat of an access token redirect is described in [7], [15] and [14]. Tokens, issued for a certain RS can successfully be used at another RS by other RSs or clients. Access tokens issued for access to one resource can also be used by an attacker to access another resource.

This can be prevented by audience and scope restricting the access token, e.g., by adding the resource's identifier/ the correct token recipient as the audience. Access tokens should be limited to only one specified RS.

Countermeasure by FAPI 1.0 The advanced FAPI 1.0 profile mandates the use of sender-constrained access tokens.

Countermeasure by FAPI 2.0 In both FAPI 2.0 profiles, sender-constrained access tokens must be supported.

D.3.2 Token replay

Token replay attacks are described in [14]. Attackers can successfully reuse an already used single-use token at the RS.

To mitigate token replay, the token lifetime is limited, and confidentiality protection is implemented for communication between the client and RS or AS. Further, clients prove the RS identity and check the TLS certificate chain. A timestamp included in the token can also reduce the attack surface. Complete mitigation is achieved by recording whether the token was already redeemed or not.

Countermeasure by FAPI 1.0 FAPI 1.0 recommends access tokens that are only valid for ten minutes. Further, RSs must check the token lifetime and revocation state. All interactions must be protected via TLS, and certificate checks are mandatory.

Countermeasure by FAPI 2.0 FAPI 2.0 also enforces TLS usage and certificate checks. Further, access tokens are sender-constrained and must be validated regarding expiration and revocation.

D.3.3 Replay of Authorized Resource Server Requests

Replay attacks of authorized resource server requests are described in [15]. Authorized resource requests can be replayed at the RS to access user data.

To mitigate this attack, TLS must be used and the RS can enforce signed requests, including nonces and timestamps, to detect this form of a replay attack.

Countermeasure by FAPI 1.0 FAPI 1.0 enforces the usage of TLS.

Countermeasure by FAPI 2.0 FAPI 2.0 enforces the usage of TLS. Further, the draft names the consideration of signing resource requests and responses. To this date, no mechanism has replaced the open question.

D.3.4 Leak of Confidential Data in HTTP Proxies

The threat of leaking of confidential data in HTTP proxies is described in [15]. Caches and proxies can fail to protect confidential data in absence of HTTP authentication headers, making it publicly accessible.

To mitigate this, RSs and clients can use Cache-control headers, if not using HTTP authentication headers. Further, token scope and lifetime should be reduced to decrease the impact of stolen tokens.

Countermeasure by FAPI 1.0 In the FAPI 1.0 access token lifetime is restricted to ten minutes. In the advanced profile, sender-constrained access token are mandatory.

Countermeasure by FAPI 2.0 In FAPI 2.0, sender-constrained access tokens must be supported in both profiles.

D.3.5 Access Token Leakage through a compromised Resource Server

The threat of access tokens leakage through a compromised RS is described in [10]. Attackers successfully compromising a RS, have then access to not only all resources but also gain access to access tokens, that are valid at other RSs.

To limit the risk of stolen access token replay at other RSs, sender-constrained access token can be utilized, as well as audience restriction of them. Further, access tokens have to be treated and stored as other credentials, e.g., not logging them.

Countermeasure by FAPI 1.0 FAPI 1.0 recommends strict access control to logs. The advanced profile mandates the use of sender-constrained access token.

Countermeasure by FAPI 2.0 In FAPI 2.0, supporting sender-constrained access tokens is mandatory in both profiles.

D.3.6 Token manufacture/modification

The threat of modified tokens is described in [14] and [7]. Tokens can be manipulated or forged to, e.g., change their validity period or widen their access. When using token-based authentication, RSs could wrongly grant access.

To prohibit token modification tokens can be sent over a TLS-protected channel. Only protecting the token using TLS does not mitigate the attack performed by malicious clients. Mitigations for token modification and manufacture include providing a MAC or digitally signing the token (by the AS). In the latter case, the signature must be validated by the client, or it is effectless. Another mitigation is to include a reference to authorization information in the token. This requires additional interaction and protection of the reference.

Countermeasure by FAPI 1.0 In FAPI 1.0, all communication must happen over a TLS-protected channel. Further, token introspection can be used, and the specification also notes the possibility of employing structured access token, such as signed JWTs.

Countermeasure by FAPI 2.0 FAPI 2.0 also demands the use of TLS for all communications.

D.3.7 Session Fixation

Session fixation is described in [23]. Any measures protecting the obtaining and use of access tokens by attackers are obsolete if the API executes protected actions without checking if the necessary privilege is present.

Therefore, any protected action must be only executed in the presence of a valid access token.

Countermeasure by FAPI 1.0 FAPI 1.0 defines rules for RSs with FAPI endpoints. They must verify access tokens and only return the correct resources, matching the given scope and entity belonging to the access token.

Countermeasure by FAPI 2.0 FAPI 2.0 also defines RS regulations, such as the mandatory verification of access tokens and the return of resources that fit entity and the granted scope.

D.3.8 Access Token Phishing by Counterfeit Resource Server

Access token phishing attacks by counterfeit resource servers are described in [15] and [10]. Attackers can impersonate a RS, that accepts access token issued by a specific AS to phish valid access tokens. The victim client that sends the access token is not bound to only one RS, but instead receives the RS URL at runtime. When the fake RS receives a valid access token, the attacker can redeem that themselves at the honest RS.

Countermeasures for this kind of attack, include the client not sending requests with access token to unknown RS. Therefore, AS metadata can be used to provide a list of supported and trust-worthy RSs, or the AS can include known RS in the token response. This method is not primarily recommended since it relies on the client and many clients fail to implement security measures correctly. Another countermeasure is the association of the RSs endpoint URL with the respective token. Further, the RS might authenticate the client and only accept tokens bound to that client. Sender-constrained access token can, e.g., be implemented through MTLS, DPoP. Restricting the token scope to one defined RS can help reduce the impact of the attack. For the audience restriction of access token, the AS binds a the access token to a certain RS, which then validates the audience and stops the flow, if it does not match.

Countermeasure by FAPI 1.0 The advanced profile enforces the use of certificate-bound access tokens. Further, the distribution of AS metadata is mandatory.

Countermeasure by FAPI 2.0 Both profiles must support sender-constrained access tokens. Further, the distribution of discovery metadata is mandatory.

D.4 Attacks on Deprecated Flows

D.4.1 Token Substitution

Token substitution attacks are described in [7]. The swapping of any kind of token by attackers is called token substitution. A known attack of this class is called the "cut and paste" attack, in which the attacker swaps a token of one session and copies it into the HTTP message for another one. The implicit flow and the hybrid flow with the response types `code token` and `code id_token token` are vulnerable to this kind of attack without the use of respective security measures. Clients need the ability to check that tokens were issued for itself. One mitigation, provided by OIDC is the use of certain claims in the ID token. With an ID token, the issuer, subject, authorized party and the hashes of access token and authorization code are signed, enabling clients to check for token substitution attacks.

Another attack vector in this context is the reordering of messages. This is mitigated by the HTTP binding, specified by OIDC, in which the token response is bound to the respective token request by the message order since TLS is used. Alternatively, ID tokens including a hash of the authorization code can be included in both token request and response.

Countermeasure by FAPI 1.0 In FAPI 1.0, the implicit grant is not accepted. In the baseline profile, the use of the hybrid flow is also not intended. According to Fett et al. [59], the FAPI 1.0 profile is essentially an authorization code flow. In the advanced profile, the Hybrid flow with the response types `code token` and `code id_token token` is not allowed. The audience is defined in the ID token, and TLS is used for all communication.

Countermeasure by FAPI 2.0 In FAPI 2.0, the implicit grant and the hybrid flow are not accepted. The audience is defined in the ID token, and TLS is used for all communication.

D.4.2 Server Response Disclosure

The threat of server response disclosure is described in [7]. Authorization/authentication responses can contain sensitive information about the client and the authentication that should not be disclosed.

To mitigate server response disclosure, the authorization code flow can be used (response type `code`), in which only the code is returned in the authorization response. Further, TLS is used and the client is authenticated. In different flows, access tokens or ID tokens are transmitted in the server response, which is redirected through the UA exposing them. Another possible mitigation is the use of signed and encrypted responses, using the client's public key or a Symmetric secret as an encrypted JWT.

Countermeasure by FAPI 1.0 FAPI 1.0 uses the authorization code flow or the hybrid flow with response type `code id_token`. When the hybrid flow is used, the advanced profile recommends supporting signed and encrypted ID tokens. Generally, TLS and client authentication for confidential clients are mandatory.

Countermeasure by FAPI 2.0 FAPI 2.0 only accepts the authorization code flow and confidential clients that are authenticated. The use of TLS is mandatory.

D.4.3 (Accidental) Exposure of Passwords at Client Site

The threat of exposure of passwords at client site is described in [15]. Passwords can be exposed at client site, by employees or attackers.

To mitigate this, the use of other flows is recommended. If not possible, the use of digest authentication and the obfuscation of passwords in logs are recommended.

Countermeasure by FAPI 1.0 FAPI 1.0 implies the use of the hybrid flow or the authorization code grant.

Countermeasure by FAPI 2.0 FAPI 2.0 explicitly forbids the use of the RO password credentials grant.

D.4.4 Eavesdropping of User Passwords in Resource Owner Password Credentials Grant

The threat of eavesdropping on user passwords is described in [15]. User Password can be eavesdropped on by attackers when sent to the AS.

To prohibit the eavesdropping of an end-user's password, TLS must be used to ensure confidentiality and alternative authentication methods can be used, e.g., such as MACs.

Countermeasure by FAPI 1.0 FAPI 1.0 implies the use of the hybrid flow or the authorization code grant. Furthermore, TLS is always used.

Countermeasure by FAPI 2.0 FAPI 2.0 explicitly forbids the use of the resource owner password credentials grant and enforces TLS for all connections.

D.4.5 Misuse of Access Token to Impersonate Resource Owner in Implicit Flow

The threat of misusing an access token to impersonate a resource owner is described in [13]. OAuth does not specify a way how public clients that uses the implicit flow can detect whether the access token on hand was issued for them or another client. In an attack scenario, in which the attacker has already obtained an access token, e.g., through phishing, they can impersonate the user they stole the token from. Therefore, they switch the access token received in the authorization response with the stolen token and forwards the response to the honest client. In the implicit flow, the provider of the access token can not be assumed to be the actual RO. Every client who does, is vulnerable to RO impersonation, which leads to information exposure and enables the attacker to access the protected resources as if they are the end-user.

The implicit flow should not be used for end-user authentication to a client, at least not without additional security measures in place. Sender-constrained access token can help to gain knowledge for whom the token was issued.

Countermeasure by FAPI 1.0 FAPI 1.0 implies the use of the hybrid flow or the authorization code grant. Furthermore, sender-constrained access tokens have to be used in the advanced profile.

Countermeasure by FAPI 2.0 FAPI 2.0 explicitly forbids the use of the implicit grant. Furthermore, sender-constrained access tokens have to be supported in both profiles.

D.4.6 Credential Leakage via Browser History

Credential leakage via browser history is described in [10] Credentials, such as authorization codes and access tokens can leak through the browser history, if an attacker has physical access to the respective device. This results in authorization code replay attacks as well as in unauthorized access to protected resources. Authorization codes can leak via the browser history when the redirect to the redirection endpoint contains the code value. Access tokens can leak via the browser

history when sent in query parameters, either through directly trying to access a page or through the redirect to the redirection endpoint in case of the implicit grant.

Mitigations include the use of authorization code replay prevention and the use of the form post response mode, as an alternative to redirecting. Additionally, access tokens should not be transmitted as a query parameter. Therefore, the use of the authorization code grant and the form post response mode are recommended.

The impact of this attack can be limited through the reduction of the token lifetime. Further, responses should be made non-cacheable.

Countermeasure by FAPI 1.0 In FAPI 1.0, access tokens must not be sent in query parameters, but as HTTP headers. The authorization code grant or the hybrid flow with the response type `code id_token` are supported. Authorization codes are single-use and access token have a lifetime of at most 10 minutes.

Countermeasure by FAPI 2.0 Authorization codes are single-use, if this is not possible, their lifetime should be restricted to one minute. Only the authorization code grant is supported. Access tokens must not be sent in query parameters but as HTTP headers.

D.4.7 Obtaining User Passwords from Authorization Server Database

The threat of obtaining user passwords from authorization server databases is described in [15]. When the AS stores username-password combinations in a database, attackers can access them through successful SQL injection attacks.

To mitigate the leakage of all username-password combinations, following the credential storage protection best practices is mandatory.

Countermeasure by FAPI 1.0 Since, the FAPI 1.0 does not use the RO password credentials grant, this attack is not applicable.

Countermeasure by FAPI 2.0 FAPI 2.0 forbids the use of the RO password credentials grant, so this attack is not applicable on the authorization code flow.

D.4.8 Client Obtains Scopes without End-User Authorization

The threat of a client obtaining scopes without end-user authorization is described in [15]. In the Resource owner password Credentials grant, the end-user only communicates with the client. Therefore, the user has no capability of limiting the token scope, without the client being able to unnoticeably circumvent their wishes.

Countermeasures include choosing a different flow, the AS always restrict the access token scope and informs the end-user about the granted access.

Countermeasure by FAPI 1.0 Since the FAPI 1.0 does not use the RO password credentials grant, this attack is not applicable. Further, the end-users must be informed in detail about the grant they are accepting.

Countermeasure by FAPI 2.0 FAPI 2.0 forbids the use of the RO password credentials grant, so this attack is not applicable on the authorization code flow.

D.4.9 Client Obtains Refresh Token through Automatic Authorization

The threat of a client obtaining a refresh token through automatic authorization is described in [15]. Since end-users only communicate with clients, long-term refresh token can be obtained by the client without user consent.

It is recommended to utilize different flows. Otherwise, the AS can prohibit the issuing of refresh tokens, except for trusted clients and inform the ROs about issued refresh token.

Countermeasure by FAPI 1.0 Since the FAPI 1.0 does not use the RO password credentials grant, this attack is not applicable.

Countermeasure by FAPI 2.0 FAPI 2.0 forbids the use of the RO password credentials grant, so this attack is not applicable on the authorization code flow.

D.4.10 Access token phishing

The threat of a access token phishing is described in [23]. Access token phishing is another common attack that is more likely to succeed when flows are used which transport tokens in the front channel.

Countermeasure by FAPI 1.0 FAPI 1.0 describes a variety of mechanisms mitigating it. One of these mechanisms is the use of flows that do not exchange access tokens in the front channel. Further, comparing preregistered `redirect_uri`s can help prevent this kind of attack. Furthermore, certificate-bound access token, e.g. by using MTLS, make phished tokens unusable, since they are bound to the victim TLS certificate, which is not in possession of the attacker.

Countermeasure by FAPI 2.0 FAPI 2.0, also enforces the usage of the authorization code flow, in which access tokens are sent in the backchannel. Sender-constrained access tokens must be supported in both baseline and advanced profiles. However, pre-registered redirect URIs are not necessary, since they are included in the PAR.

List of Figures

List of Tables

List of Listings

Bibliography

[1] *Openid certification | openid*. https://openid.net/certification/, visited on 1/27/2022.

[2] *Draft-02: Financial-grade api: Jwt secured authorization response mode for oauth 2.0 (jarm)*, 10/18/2018. https://openid.net/specs/openid-financial-api-jarm.html, visited on 1/27/2022.

[3] *Draft-06: Financial-grade api - part 1: Read-only api security profile*, 10/18/2018. https://openid.net/specs/openid-financial-api-part-1-ID2.html, visited on 4/29/2022.

[4] *Draft-06: Financial-grade api - part 2: Read and write api security profile*, 10/18/2018. https://openid.net/specs/openid-financial-api-part-2-ID2.html, visited on 4/29/2022.

[5] *Oauth 2.0 token binding*, 10/19/2018. https://tools.ietf.org/id/draft-ietf-oauth-token-binding-08.html, visited on 5/6/2022.

[6] *Docker sdk for python — docker sdk for python 5.0.3 documentation*, 10/8/2021. https://docker-py.readthedocs.io/en/stable/, visited on 5/5/2022.

[7] *Final: Openid connect core 1.0 incorporating errata set 1*, 11/8/2014. https://openid.net/specs/openid-connect-core-1_0.html, visited on 1/27/2022.

[8] *Final: Openid connect discovery 1.0 incorporating errata set 1*, 11/8/2014. https://openid.net/specs/openid-connect-discovery-1_0.html, visited on 1/27/2022.

[9] *Final: Openid connect dynamic client registration 1.0 incorporating errata set 1*, 11/8/2014. https://openid.net/specs/openid-connect-registration-1_0.html, visited on 1/27/2022.

[10] *draft-ietf-oauth-security-topics-19*, 1/27/2022. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics, visited on 1/27/2022.

[11] *draft-ietf-oauth-v2-1-04*, 1/27/2022. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-04, visited on 1/27/2022.

[12] *Oauth 2.1*, 1/27/2022. https://oauth.net/2.1/, visited on 1/27/2022.

[13] *Rfc 6749 - the oauth 2.0 authorization framework*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc6749, visited on 1/27/2022.

[14] *Rfc 6750 - the oauth 2.0 authorization framework: Bearer token usage*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc6750, visited on 1/27/2022.

[15] *Rfc 6819 - oauth 2.0 threat model and security considerations*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc6819, visited on 1/27/2022.

[16] *Rfc 7591 - oauth 2.0 dynamic client registration protocol*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc7591, visited on 1/27/2022.

[17] *Rfc 7636 - proof key for code exchange by oauth public clients*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc7636, visited on 1/27/2022.

[18] *Rfc 8414 - oauth 2.0 authorization server metadata*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc8414, visited on 1/27/2022.

[19] *Rfc 8628 - oauth 2.0 device authorization grant*, 1/27/2022. https://datatracker.ietf.org/doc/html/rfc8628, visited on 1/27/2022.

[20] *Financial-grade api (fapi) wg | openid*, 2016. https://openid.net/wg/fapi/, visited on 1/27/2022.

[21] *Fapi - financial grade api*, 2022. https://fapi.openid.net/, visited on 1/27/2022.

[22] *Final: Financial-grade api security profile 1.0 - part 1: Baseline*, 3/12/2021. https://openid.net/specs/openid-financial-api-part-1-1_0.html, visited on 1/27/2022.

[23] *Final: Financial-grade api security profile 1.0 - part 2: Advanced*, 3/12/2021. https://openid.net/specs/openid-financial-api-part-2-1_0.html, visited on 1/27/2022.

[24] *Addons*, 3/19/2022. https://docs.mitmproxy.org/stable/addons-overview/, visited on 5/5/2022.

[25] *Certificates*, 3/19/2022. https://docs.mitmproxy.org/stable/concepts-certificates/, visited on 5/5/2022.

[26] *Introduction*, 3/19/2022. https://docs.mitmproxy.org/stable/, visited on 5/5/2022.

[27] *draft-ietf-oauth-rar-11*, 4/25/2022. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rar, visited on 4/25/2022.

[28] *openid / fapi / fapi_2_0_advanced_profile.md — bitbucket*, 4/25/2022. https://bitbucket.org/openid/fapi/src/596057dff73d039ffa9a6213256bb529eb7e2ab9/FAPI_2_0_Advanced_Profile.md?at=master, visited on 4/25/2022.

[29] *openid / fapi / fapi_2_0_attacker_model.md — bitbucket*, 4/25/2022. https://bitbucket.org/openid/fapi/src/c1095f10f13c17c93a425081ec5669796245ff9c/FAPI_2_0_Attacker_Model.md?at=master, visited on 4/25/2022.

[30] *openid / fapi / fapi_2_0_baseline_profile.md — bitbucket*, 4/25/2022. https://bitbucket.org/openid/fapi/src/47c71db3d6d535b805ac207f89119b7f32b1f74e/FAPI_2_0_Baseline_Profile.md?at=master, visited on 4/25/2022.

[31] *Rfc 8705 - oauth 2.0 mutual-tls client authentication and certificate-bound access tokens*, 4/25/2022. https://datatracker.ietf.org/doc/html/rfc8705, visited on 4/25/2022.

[32] *Rfc 9101 - the oauth 2.0 authorization framework: Jwt-secured authorization request (jar)*, 4/25/2022. https://datatracker.ietf.org/doc/html/rfc9101, visited on 4/25/2022.

[33] *Rfc 9126 - oauth 2.0 pushed authorization requests*, 4/25/2022. https://datatracker.ietf.org/doc/html/rfc9126, visited on 4/25/2022.

[34] *draft-ietf-oauth-dpop-07*, 4/26/2022. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop, visited on 4/26/2022.

[35] *Rfc 7515 - json web signature (jws)*, 4/27/2022. https://datatracker.ietf.org/doc/html/rfc7515, visited on 4/27/2022.

[36] *Rfc 7516 - json web encryption (jwe)*, 4/27/2022. https://datatracker.ietf.org/doc/html/rfc7516, visited on 4/27/2022.

[37] *Rfc 7519 - json web token (jwt)*, 4/27/2022. https://datatracker.ietf.org/doc/html/rfc7519, visited on 4/27/2022.

[38] *openid / fapi / fapi_1.0 / changes-between-id2-and-final.md — bitbucket*, 4/29/2022. https://bitbucket.org/openid/fapi/src/68bee86ae0c7a1787cc6ed1645292bf83eac6a8d/FAPI_1.0/changes-between-id2-and-final.md?at=master, visited on 4/29/2022.

[39] *Rfc 8725 - json web token best current practices*, 4/29/2022. https://datatracker.ietf.org/doc/html/rfc8725, visited on 4/29/2022.

[40] *Rfc 6979 - deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa)*, 4/30/2022. https://datatracker.ietf.org/doc/html/rfc6979, visited on 4/30/2022.

[41] *Rfc 7662 - oauth 2.0 token introspection*, 4/30/2022. https://datatracker.ietf.org/doc/html/rfc7662, visited on 4/30/2022.

[42] *Rfc 9207 - oauth 2.0 authorization server issuer identification*, 4/30/2022. https://datatracker.ietf.org/doc/html/rfc9207, visited on 4/30/2022.

[43] *Api - oxd 4.2 docs*, 5/5/2022. https://gluu.org/docs/oxd/4.2/api/, visited on 5/5/2022.

[44] *Configuration - oxd 4.2 docs*, 5/5/2022. https://gluu.org/docs/oxd/4.2/configuration/oxd-configuration/, visited on 5/5/2022.

[45] *Docker - gluu server 4.2 docs*, 5/5/2022. https://gluu.org/docs/gluu-server/4.2/installation-guide/install-docker/, visited on 5/5/2022.

[46] *Gluu server 4.2 docs*, 5/5/2022. https://gluu.org/docs/gluu-server/4.2/, visited on 5/5/2022.

[47] *oxauth json properties - gluu server 4.2 docs*, 5/5/2022. https://gluu.org/docs/gluu-server/4.2/reference/JSON-oxauth-prop/#general-configuration, visited on 5/5/2022.

[48] *oxd 4.2 docs*, 5/5/2022. https://gluu.org/docs/oxd/4.2/, visited on 5/5/2022.

[49] *oxtrust api - gluu server 4.2 docs*, 5/5/2022. https://gluu.org/docs/gluu-server/4.2/api-guide/oxtrust-api/, visited on 5/5/2022.

[50] *oxtrust api - gluu server 4.2 docs*, 5/5/2022. https://gluu.org/docs/gluu-server/4.2/api-guide/oxtrust-api/#updateoxauthjsonsetting, visited on 5/5/2022.

[51] *Release notes - oxd 4.2 docs*, 5/5/2022. https://gluu.org/docs/oxd/4.2/release-notes/, visited on 5/5/2022.

[52] *Wireshark*, 5/5/2022. https://www.wireshark.org/, visited on 5/8/2022.

[53] *Draft-02: Financial-grade api: Client initiated backchannel authentication profile*, 8/16/2019. https://openid.net/specs/openid-financial-api-ciba.html, visited on 1/27/2022.

[54] *Openid connect client-initiated backchannel authentication flow - core 1.0*, 9/1/2021. https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html, visited on 1/27/2022.

[55] *Jwt response for oauth token introspection*, 9/4/2021. https://www.ietf.org/id/draft-ietf-oauth-jwt-introspection-response-12.html, visited on 4/27/2022.

[56] V. Bertocci and T. Lodderstedt: *Exploring financial-grade api (fapi) with torsten*, 1/23/2021. https://identityunlocked.auth0.com/public/49/Identity%2C-Unlocked.--bed7fada/d847cfb7, visited on 1/27/2022.

[57] W. Denniss and J. Bradley: *Oauth 2.0 for native apps*. https://www.rfc-editor.org/rfc/rfc8252.html.

[58] D. Fett: *Fapi 2.0: A high-security profile for oauth and openid connect*. 1617-5468, 2021, ISSN 1617-5468. https://dl.gi.de/handle/20.500.12116/36503.

[59] D. Fett, P. Hosseyni, and R. Kuesters: *An extensive formal security analysis of the openid financial-grade api*. https://arxiv.org/pdf/1901.11520.

[60] GitHub: *Github - gluufederation/oxauth at version_4.2.0*, 5/5/2022. https://github.com/GluuFederation/oxAuth/tree/version_4.2.0, visited on 5/5/2022.

[61] GitHub: *Github - gluufederation/oxd: Client software to secure apps with oauth 2.0, openid connect, and uma*, 5/5/2022. https://github.com/GluuFederation/oxd, visited on 5/5/2022.

[62] GitHub: *Github - gluufederation/oxtrust: Gluu server ui for managing authentication, authorization and users*, 5/5/2022. https://github.com/GluuFederation/oxTrust, visited on 5/5/2022.

[63] GitHub: *Gluu*, 5/5/2022. https://github.com/GluuFederation, visited on 5/5/2022.

[64] GitHub: *oxauth/oxauth-config.json at 4.2.1 · gluufederation/oxauth*, 5/5/2022. https://github.com/GluuFederation/oxAuth/blob/4.2.1/Server/conf/oxauth-config.json, visited on 5/5/2022.

[65] S.P. Hosseyni Damabi: *Security analysis of the OpenID financial-grade API*. PhD thesis, 2018. https://elib.uni-stuttgart.de/handle/11682/10097.

[66] Mohamed, Aly Mohamed Abdalkarim Salheen: *A prototype implementation of the OpenID Financial-grade API*. PhD thesis, 2021. https://elib.uni-stuttgart.de/handle/11682/11641.

[67] T.E. Parliament and the Council of the European Parliament: *Directive (eu) 2015/ of the european parliament and of the council of 25 november 2015 on payment services in the internal market, amending directives 2002/65/ec, 2009/110/ec and 2013/36/eu and regulation (eu) no 1093/2010, and repealing directive 2007/64/ec*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32015L2366&from=DE, visited on 1/28/2022.

[68] N. Sakimura and A. Saxena: *Openid foundation fapi wg: June 2017 update*, 2017. https://www.slideshare.net/nat_sakimura/openid-foundation-fapi-wg-june-2017-update, visited on 1/27/2022.

[69] Y. Sheffer, R. Holz, and P. Saint-Andre: *Recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls)*. `https://www.rfc-editor.org/rfc/rfc7525.html`.