Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

A Prototype Implementation of the OpenID Financial-grade API

Aly Mohamed Abdalkarim Salheen Mohamed

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Pedram Hosseyni, M.Sc.
Commenced:	October 13, 2020
Completed:	May 04, 2021

Abstract

With the rise of the financial technology (FinTech) industry and the introduction of the Payment Services Directive 2 (PSD 2) [33], banks are moving towards digitization. With this comes the ability for third-party companies and service providers to provide bank account holders their services independently of the banks themselves. For example, one such provider might utilize machine learning to gauge the credit score of a bank account holder based on their transaction history. To provide their services, these third-party providers need to access the bank account holder's data. Methods such as screen scraping were used to provide this access. However, its insecurity and weaknesses in such a high-stake high-risk environment necessitated a secure alternative. With that in mind, the OpenID Financial-grade API (FAPI) specification describes a hardened version of the OAuth 2.0 Authorization Framework and the OpenID Connect Core 1.0 (OIDC) Authentication Layer. It makes use of several new extensions such as *Pushed Authorization Requests* (PARs) and *Rich Authorization Requests* (RARs) as well as *JSON Web Signature* (JWS) to offer non-repudiation, which is critical should, e.g., a client attempts to refute they ever initiated a payment request. While the first version of the FAPI, namely FAPI 1.0, has been finalized in early 2021, its successor, FAPI 2.0, is still in its infancy. Despite this, the FAPI 2.0 is designed to provide the same strong security guarantees while mitigating attacks on the first version that were discovered [17, 26]. As the specification is still being drafted, it has garnered relatively little public attention. Even so, end-users and developers alike, especially in the FinTech industry, should benefit from a demonstration of this new specification, specifically as a software implementation. This thesis covers the development of a prototype for the FAPI 2.0 with which end-users can simulate the Baseline and Advanced profile flows. Developers can gain insight into the specifics of an example implementation of the profiles.

Kurzfassung

Mit dem Aufstieg der Finanztechnologiebranche (FinTech) und der Einführung der Payment Services Directive 2 (PSD 2) [33] bewegen sich die Banken in Richtung Digitalisierung. Damit einher geht die Möglichkeit für Drittunternehmen und Dienstleister, Bankkontoinhabern ihre Dienste unabhängig von den Banken selbst anzubieten. Ein solcher Anbieter könnte zum Beispiel maschinelles Lernen nutzen, um die Kreditwürdigkeit eines Kontoinhabers auf Basis seiner Transaktionshistorie zu ermitteln. Um ihre Dienste anbieten zu können, müssen diese Drittanbieter auf die Daten des Kontoinhabers zugreifen, wofür Methoden wie Screen Scraping verwendet wurden. Die Unsicherheit und Schwächen dieses Verfahrens in einer Umgebung mit hohem Risiko erforderten jedoch eine sicherere Alternative. Vor diesem Hintergrund beschreibt die OpenID Financial-grade API (FAPI) Spezifikation eine gehärtete Version des OAuth 2.0 Authorization Frameworks und der OpenID Connect Core 1.0 (OIDC) Authentication Layer. Sie nutzt mehrere neue Erweiterungen wie Pushed Authorization Requests (PARs) und Rich Authorization Requests (RARs). Um Non-Repudiation zu bieten, können JSON Web Signatures (JWS), was notwendig ist, wenn z. B. ein Client versucht, zu widerlegen, dass er jemals eine Zahlungsanfrage initiiert hat. Während die erste Version der FAPI, nämlich FAPI 1.0, Anfang 2021 fertiggestellt wurde, steckt ihr Nachfolger, FAPI 2.0, noch in den Kinderschuhen. Trotzdem soll die FAPI 2.0 die gleichen starken Sicherheitsgarantien bieten und gleichzeitig Angriffe auf die erste Version abwehren, die entdeckt wurden [18, 26]. Da sich die Spezifikation noch in der Entwurfsphase befindet, hat sie bisher relativ wenig öffentliche Aufmerksamkeit erregt. Dennoch sollten sowohl Endbenutzer als auch Entwickler, insbesondere in der FinTech-Branche, von einer Demonstration dieser neuen Spezifikation profitieren, insbesondere in Form einer Softwareimplementierung. Diese Arbeit umfasst die Entwicklung eines Prototyps für die FAPI 2.0, mit dem Endbenutzer die Baseline- und Advanced-Profile-Flows simulieren können. Entwickler können einen Einblick in die Spezifika einer Beispielimplementierung der Profile erhalten.

Contents

List of Figures

List of Listings

List of Abbreviations

AC authorization code. 19, 20, 24, 26–28, 37, 46

AI Artificial Intelligence. 15

API Application Programming Interface. 15, 16, 22, 24, 30, 49

AS authorization server. 19, 20, 22, 24, 26–31, 33, 36, 37, 41, 44–46

AT access token. 19, 20, 22, 24, 26–28, 37, 44, 46

CA Certificate Authority. 34

DOM Document Object Model. 15

FAPI Financial-grade API. 3, 16, 19, 20, 22, 24, 27, 31, 33–35, 39, 46, 49

FinTech Financial Technology. 3, 15

HTTP Hypertext Transfer Protocol. 20, 26, 34, 36, 47

IDE Integrated Development Environment. 35

IdP Identity Provider. 22, 26, 28, 31, 46

JAR JWT Secured Authorization Request. 27, 30, 31, 47

JARM JWT Secured Authorization Response Mode for OAuth 2.0. 27, 31, 47

JWE JSON Web Encryption. 31

JWS JSON Web Signature. 3, 27, 31, 47

JWT JSON Web Token. 26, 27, 31

MTLS Mutual Transport Layer Security. 24, 26, 28, 34, 44–46

OIDC OpenID Connect Core 1.0. 3, 16, 19, 22, 24, 31, 33, 34, 37, 44, 49

ORM object–relational mapping. 33

PAR Pushed Authorization Request. 3, 24, 26, 28–30, 33, 42, 44–46

PKCE Proof Key for Code Exchange. 26, 27, 37, 46

PSD 2 Payment Services Directive 2. 3, 15

RAR Rich Authorization Request. 3, 11, 24, 26, 30, 33, 44

RO resource owner. 19, 20, 22, 24, 26–30, 47

RP Relying Party. 22, 31

RS resource server. 19, 20, 26, 28, 46

RT refresh token. 19, 26, 46

SSO Single Sign On. 16, 19

TLS Transport Layer Security. 13, 24, 26, 28, 34, 36, 44–46

UA user agent. 19, 20, 24, 26–29, 34, 36, 41, 44

URI Uniform Resource Identifier. 19, 26, 27, 29, 46

URL Uniform Resource Locator. 36

UUID Universally Unique ID. 35

YAML YAML Ain't Markup Language. 35, 37, 49

1 Introduction

With the advent of the Internet, individuals and institutions alike saw an opportunity to exploit this new technology in ways considered unfeasible before. For example, e-commerce has disrupted the traditional brick-and-mortar businesses by providing goods in a much more convenient manner.

The financial industry is no exception; the rise of the FinTech industry is possible thanks to the digitization of financial services. Examples of FinTech innovation includes the use of artificial intelligence (AI) in order to better assess the credit score of a customer based on their transaction history, as well as cryptocurrencies such as Bitcoin [4] and Ethereum [25], the latter having smart-contract capabilities allowing the automatic execution of certain tasks upon an event such as a transaction confirming the purchase of goods. One consequence of this shift is that most prominent banks now provide online and mobile banking services; using these banks' applications, whether Web-based or available as native applications on a smartphone, consumers can now access their data, perform transactions on-demand, and perform other actions that would have otherwise necessitated a visit to a brick-and-mortar branch belonging to that bank or an ATM.

However, a problem has become clear: Traditional banks have complete control over the consumer's data; there has been little incentive for these banks to provide an interface accessible to third parties so that they can offer the services powered by the new technologies mentioned above. One way around this was the use of "screen scraping"; briefly explained, the consumer would give their online banking login credentials to the third-party application, which would then use them to log into the consumer's account and "scrap" the displayed information through the Document Object Model (DOM), such as the consumer's account's balance, and transaction history.

One fatal problem of this approach is that now the third-party provider has full access to the account, and, short of changing the password entirely, there exists no simple mechanism in place for the user to revoke access from the third-party provider. Furthermore, even under the assumption that this provider is honest and will not abuse the credentials, if the provider improperly stores them in a database (presumably to continue providing their services without having to ask for login details each time), then an attack on the database could leak not just this one consumer's info, but every consumer who used the provider's services.

To combat this issue and allow third parties to enter the financial industry and thus boost FinTech as a whole, several initiatives have been undertaken to establish a legal framework within which customers can consent to third-party providers accessing their data at their respective banks in a secure and controlled manner. One prominent example of such an initiative is the PSD 2 [33] of the European Union came into effect in 2018. Under this directive, banks have to provide an application programming interface (API) through which third-party services can retrieve or modify customer data or even perform actions on behalf of the user. The user determines the scope of the data the third party is allowed to access.

While these represent steps towards a more digitalized financial sector, it is not without its caveats. For one, financial accounts and related information is at a much higher risk of compromise by attackers due to the high-stakes nature of it; should an attacker, for instance, manage to retrieve the bank credentials of a user due to a security vulnerability in the API, it could lead to financial ruin for the customer and irreparable damage to the reputation of the bank as well as possibly the third-party service.

In light of this, it has become apparent that an open standard for interfaces between different parties which provided some security guarantees was essential. These interfaces have to take a specific attacker model in mind, which assumes what actions an attacker attacking the system can perform and what information they can retrieve. To that end, the FAPI has been in development by the OpenID Foundation. Built on top of the already existing OIDC[1], itself built on top of the OAuth 2.0 standard, the FAPI provides a specification of protocols with security and privacy in mind. As the name implies, the primary target for this specification is APIs in the financial sector. However, any high-stakes scenario where attack-incurred losses are considered unacceptable can benefit from this specification.

The FAPI specification consists of two versions, FAPI 1.0 and 2.0. FAPI 2.0 is still in the Internet-Draft stage, but it presents new technologies and improvements over the first version. With that in mind, a hands-on demonstration of these specifications, specifically the Baseline and Advanced profiles would be beneficial to anyone wishing to understand how the new specifications work. A previous prototype for simulating OAuth 2.0 and OpenID Connect 1.0 was developed by Erdemann [12]. This work builds upon and extends this prototype with the FAPI 2.0 flows.

While the primary audience of previous prototypes were end-users who are not concerned with the concrete implementation of the prototype, this work attempts to cover the needs of both end-users as well as developers; we uncover several rooms for improvement while building upon previous work, without which development and work on the prototype became unnecessarily difficult. For example, linters along with providing type annotations for variables in Python have allowed us to discover bugs in even the first iteration of the prototype before any run of the software was necessary. We thus implement several fixes to the prototype and propose guidelines on how to fix and mitigate such issues in the future, thereby reducing the effort needed to get accustomed to the codebase and extending its functionality.

This thesis attempts to solve this by providing a prototype implementation of FAPI 2.0 Baseline and Advanced profiles. The prototype simplifies several aspects which may not precisely mirror real use-cases. The goal is that end-users can gain a better understanding of how the FAPI 2.0 works by simulating the flows themselves. With appropriate documentation, developers should be able to understand the codebase with low difficulty and, if one desires, extend the prototype to support other protocols not covered by this work.

The thesis is structured as follows:

Chapter 2 — Foundations & Related Work We cover the basics of OAuth 2.0 and OIDC necessary for the FAPI as well as extensions utilized in FAPI and the rationale behind their usage. In addition, we discuss other work which deal with variants of Single Sign On (SSO).

[1] Note that OIDC consists of several documents, each addressing a specific aspect of the overall OpenID Connect Protocol Suite [32]. We concern ourselves with the 'OpenID Connect Core 1.0 incorporating errata set 1' document [21].

Chpater 3 — Analysis of Existing Prototype We go over the existing prototype, including functionality relevant to the FAPI which has either been partially or fully implemented, and demonstrate an example flow of the program for reference.

Chapter 4 — Implementation We cover the specifics of our implementation, including how each requirement is fulfilled and which simplifications were factored in. In addition, requirements which were not implemented are mentioned with clarification as to why.

Chapter 5 — Conclusion We provide a summary of the work, including the takeaway from the project.

2 Foundations & Related Work

Section 2.1 discusses the theory behind the popular SSO schemes OAuth 2.0 and OpenID Connect as well as the FAPI 2.0. Section 2.2 covers similar work that has been carried out and whose results complement those presented in this thesis.

2.1 Foundations

Section 2.1.1 gives a brief introduction to the OAuth 2.0 Authorization Framework, including the *Authorization Code Grant*, which is necessary for the FAPI 2.0. Section 2.1.2 covers the OpenID Connect Core 1.0 authentication layer which sits on top of OAuth 2.0 to provide authentication. Section 2.1.3 outlines the Baseline as well as Advanced Profiles, including their attacker models. Both of these profiles make use of various security extensions to provide certain security guarantees. These extension are finally discussed in Section 2.1.4.

2.1.1 OAuth 2.0

The OAuth 2.0 framework [23] defines an open standard for *access delegation*. A *resource owner* (RO) grants a third-party (referred to as the *client*) access to a *resource* owned by the RO. This access grant is represented by an *access token* (AT) issued by an *authorization server* (AS). Using this AT, the client gains access to the resource stored at a *resource server* (RS).

ATs have a limit on the length of their validity. As such, *refresh tokens* (RTs) are used to request a new AT from the AS. There is a security reason why OAuth 2.0 issues a different token with which the client can obtain a new *authorization code* (AC). The AT is exchanged between the client and RS, while the RT is exchanged only with the AS. Having a long-living AT that is leaked to an attacker essentially grants them access to the RO's protected resources until the AT is revoked, while a short-lived AT is only useful for the narrow time window in which it's valid, thereby reducing the time window within which the attacker can gain access. For more information, see [38].

A simple example of access delegation would be a user (the RO) of an online photo editor (the client) granting it access to the user's photo (the resource) stored in their personal Google Drive online storage (with Google Drive acting as the RS and Google the AS). How the user grants this access to the online photos storage and the sequence of requests and responses necessary to achieve this is defined by so-called *flows*, also called *grants*.

For the delegation to work, the client has to have registered itself at the AS, after which it obtains a client ID (also denoted as `client_id`) which uniquely identifies *all* instances of the client and optionally a client secret (also denoted as `client_secret`). In addition, it registers one or more *redirection Uniform Resource Identifiers (URIs)*, which are necessary for redirecting the *user agent* (UA) to the correct URI when utilizing specific flows Section 2.1.1.

Furthermore, we differentiate between clients that can securely store their credentials and those that cannot. the former are classified as *confidential* clients while the latter are considered *public* clients [3, 41].

Public clients They cannot securely store long-term secrets, in particular their `client_secret`. Examples include a JavaScript application running in a browser on an end user's device; the user can read out the secret from the JavaScript source code, and binary executables can be decompiled to reveal the `client_secret`. As a result, public clients cannot authenticate themselves to ASs and are thus not issued a client secret.

Confidential clients They can keep secrets securely stored such that no other party can access them. Examples include clients running on dedicated Web servers where one can assume that no malicious application is running. Since they can keep secrets, they are usually required to authenticate themselves to the AS before obtaining the AT. Authentication can be performed either by sending a *client secret* or by proving the possession of such a secret.

OAuth 2.0 defines several flows. Each flow defines a sequence of steps to be performed so that the AS issues the AT, and is designed for specific use-cases, such as whether the client should act on its or the user's behalf. They include the *Authorization Code Grant*, *Implicit Grant*, *Resource Owner Password Credentials Grant*, and *Client Credentials Grant*. Since the FAPI 2.0 covered by this thesis only permits the use of the Authorization Code Grant due to security considerations (discussed in a later chapter), we will only consider the Authorization Code Grant in this thesis.

Authorization code Grant

The *authorization code* (AC) grant is one of the flows specified in the OAuth 2.0 standard, and lays the foundation for the FAPI (see Section 2.1.3). In this flow, the AS issues an *AC*. This authorization code represents the grant the RO has issued to the client. This AC is passed on from the AS to the client via a redirect in the RO's UA. The client exchanges this AC for the AT necessary to access the resource at the RS.

On a high level, the grant works as follows: The RO initiates the flow at the client[1]. The client then sends an Hypertext Transfer Protocol (HTTP) redirect to the AS, where the RO authenticates themselves to the AS, usually by providing login credentials such as a username and password. Once the AS authenticates the RO, the AS redirects the RO back to the client with the AC. The client extracts the value of the authorization code from the HTTP redirect and uses it to request an AT. Upon receiving the AT, the client sends an HTTP request to the RS in order to access the data.

[1] In reality, the precise method with which the flow is started is out of the scope of the OAuth 2.0 specification. Therefore, it is up to the individual implementations how they should initiate the flow. Usually, an HTTP request is sufficient for that purpose.

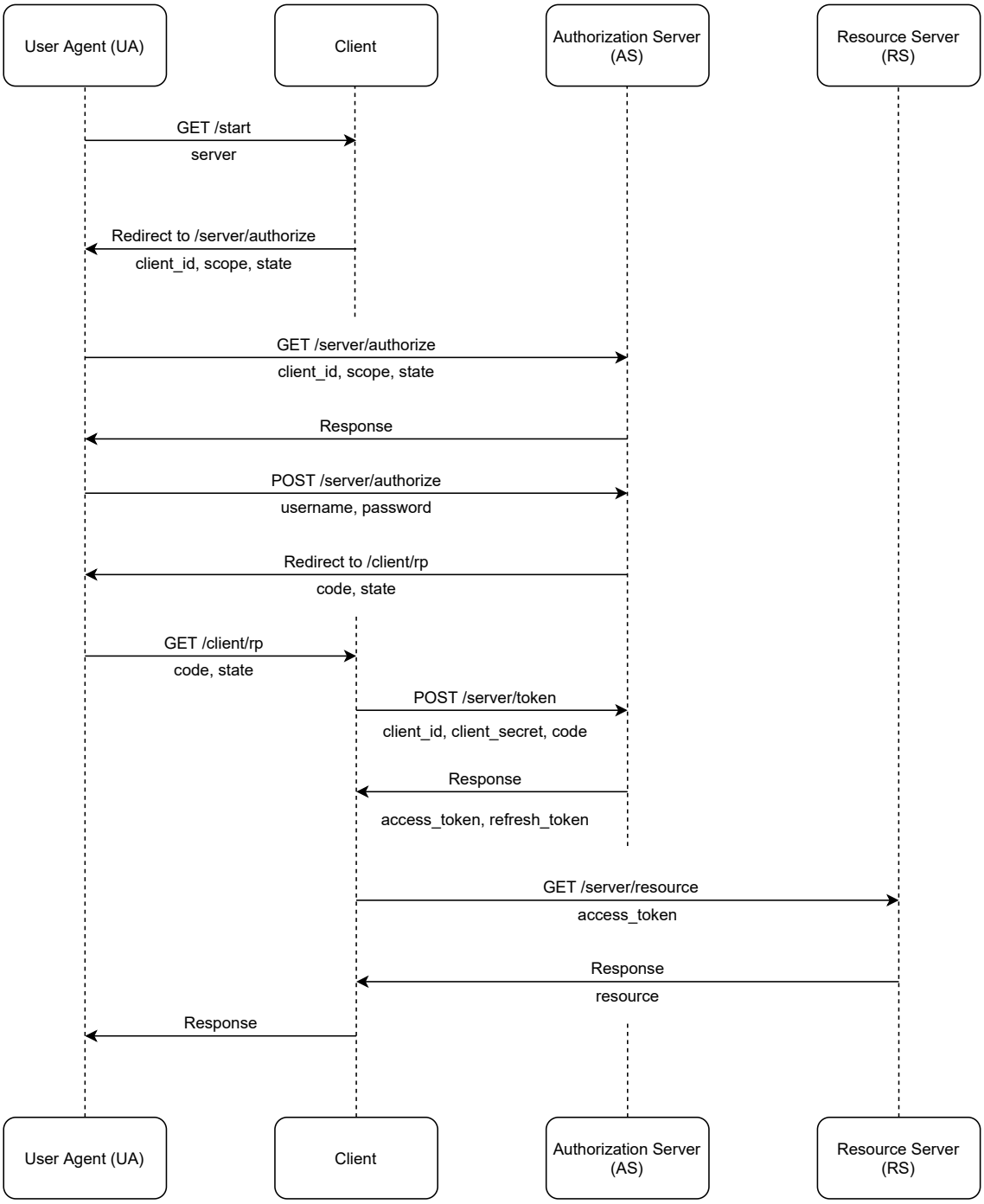

Figure 2.1: Authorization Code Grant.

Since the AT is an opaque string, the RS has to be able to check the various properties of the AT, for example, whether it is still valid as well as the identity of the AS that issued it. In OAuth 2.0, ATs are *bearer tokens*, whose specification outlines how they should be used and which checks to perform for security [22]. For Token Introspection, [35] provides guidelines on how RSs can query an AS about a given AT, for example, what resources the client can access, and whether the AT is still valid.

2.1.2 OpenID Connect Core 1.0

While OAuth 2.0 is designed for access delegation, it does not contain any mechanisms for *authentication*; the idea being that a user authenticates themselves to a *Relying Party* (RP)[2]using the authorization step performed at the AS. Upon successful login at the AS, the AS then allows the client to whom the RO granted consent to obtain scoped claims, i.e., selected assertions about the user associated with the issued AT, such as name and date of birth.

Usage of the OAuth 2.0 AT directly for authentication purposes can lead to severe attacks since *anyone* in possession of the AT can then use it to impersonate the victim [42, 44]. A simple analogy to highlight the flaw of implying authentication through authorization would be a person A proving to person B they live in a house at a given address by giving them a key to the house; person A may have proven that they have access to the house by *delegating access to person B* and therefore live in the house, but now person B has unfettered access to the house as well, and could prove to a third person C that they (person B) live in that house (which may well call into question whether person A lives in that house, but the flaw remains the same) or even give the key to a malicious person E intending on robbing the house.

Figure 2.2 shows how OpenID Connect Core 1.0 (OIDC) retrofits the regular OAuth 2.0 Authorization Code flow, with the additions by OIDC shown in blue. OIDC acts as an authentication layer on top of OAuth 2.0; it retrofits existing grants defined in the OAuth 2.0 specification with cryptographically secured ID tokens along with a dedicated endpoint for obtaining user information. In particular, OIDC achieves this through the introduction of an *id token* which contains a set of *claims*. A claim is an assertion made or "claimed" by the *Identity Provider* (IdP)[2], for example, the unique ID of the user at the IdP and the client as the subject of the id token. It serves as a one-time proof of the user's identity to the client.

In addition, a new endpoint called the UserInfo endpoint is introduced. The client can also obtain information about the client by submitting their AT. If validation checks come through, they receive information about the RO as allowed by the user.

[2]While the terminology used varies depending on whether one is considering OAuth 2.0 or OIDC, the entities involved in the flows are largely the same. For example, the RP in OIDC is usually also the client in OAuth 2.0.

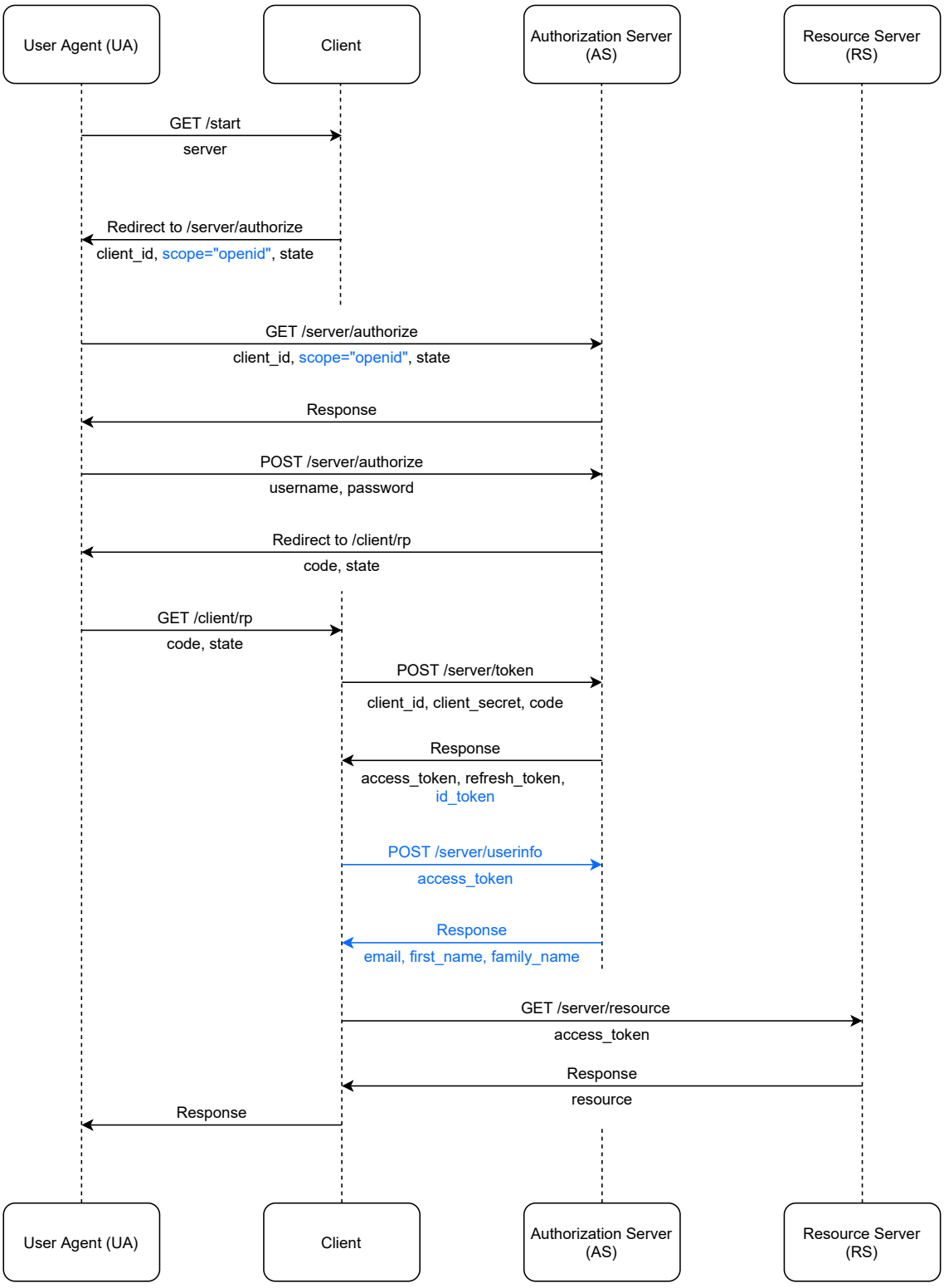

Figure 2.2: OpenID Connect Core 1.0: Authorization Code Flow.

2.1.3 Financial-grade API

The Financial-grade API (FAPI) is yet another layer, sitting on top of OIDC (recall that OIDC itself sits on top of OAuth 2.0 and provides the option of authentication *in addition to* authorization, not serve as a complete replacement). The audience of FAPI are financial accounts and, more broadly, any high-stakes high-risks scenarios, for example, government APIs.

The FAPI assumes a stronger attacker model than either standard OAuth 2.0 or OIDC. As such, both these profiles impose more stringent requirements. For details on the attacker model, see [30].

Baseline Profile

The Baseline profile is intended to be used in scenarios where authorized clients can only read a RO's data. Clients utilizing this profile should not be able to, e.g., perform transactions on the user's behalf or other state-changing operations since the Baseline profile does not offer non-repudiation. For read-write scenarios, API developers should use the Advanced profile (see Section 2.1.3).

Below is a brief overview of the most important requirements imposed by the Baseline profile. Since it is still in the draft stage, the profile specification is subject to changes. For reference, this work deals with the profile at the git[3] commit **977d75a** [31].

Authorization code grant only The Baseline profile makes exclusive use of the authorization code mode ([23, Section 4.1]), and rejects any authorization requests made using the Implicit or Resource Owner Password Credentials grants [23, Sections 4.2, 4.3]. The reason for rejecting the Implicit Grant is due to multiple weaknesses inherent to the fact that the AS passes the AT to the RO's user agent as a fragment [16, PDF Slide 23]. As for the Resource Owner Password Credentials Grant, it allows clients to see the username and password of the RO used as login credentials, which poses a serious security risk if the client ever becomes dynamically corrupted or taken over by an attacker [16, PDF Slide 43].

Pushed Authorization Requests Only The Baseline Profile mandates the use of PARs for all authorization requests (see Section 2.1.4 for details on the specification), a different method by which the Authorization Request parameters such as `response_type` and `client_id` are sent from the client to the AS directly. In addition, any parameters sent outside the PAR request must be rejected except for the `client_id` and `request_uri` parameters.

Support Rich Authorization Requests There may be situations where the original `scope` parameter introduced by OIDC is not expressive enough. For example, a client wishing to perform a one-time transaction of 100 Euros is unable to express it using the regular `openid` parameter value. The AS is likewise unable to inform the RO of the specific transaction value. To solve this, RARs allows clients to request finer permissions and the AS can more accurately inform the user of the authorization request in question.

[3] https://git-scm.com/

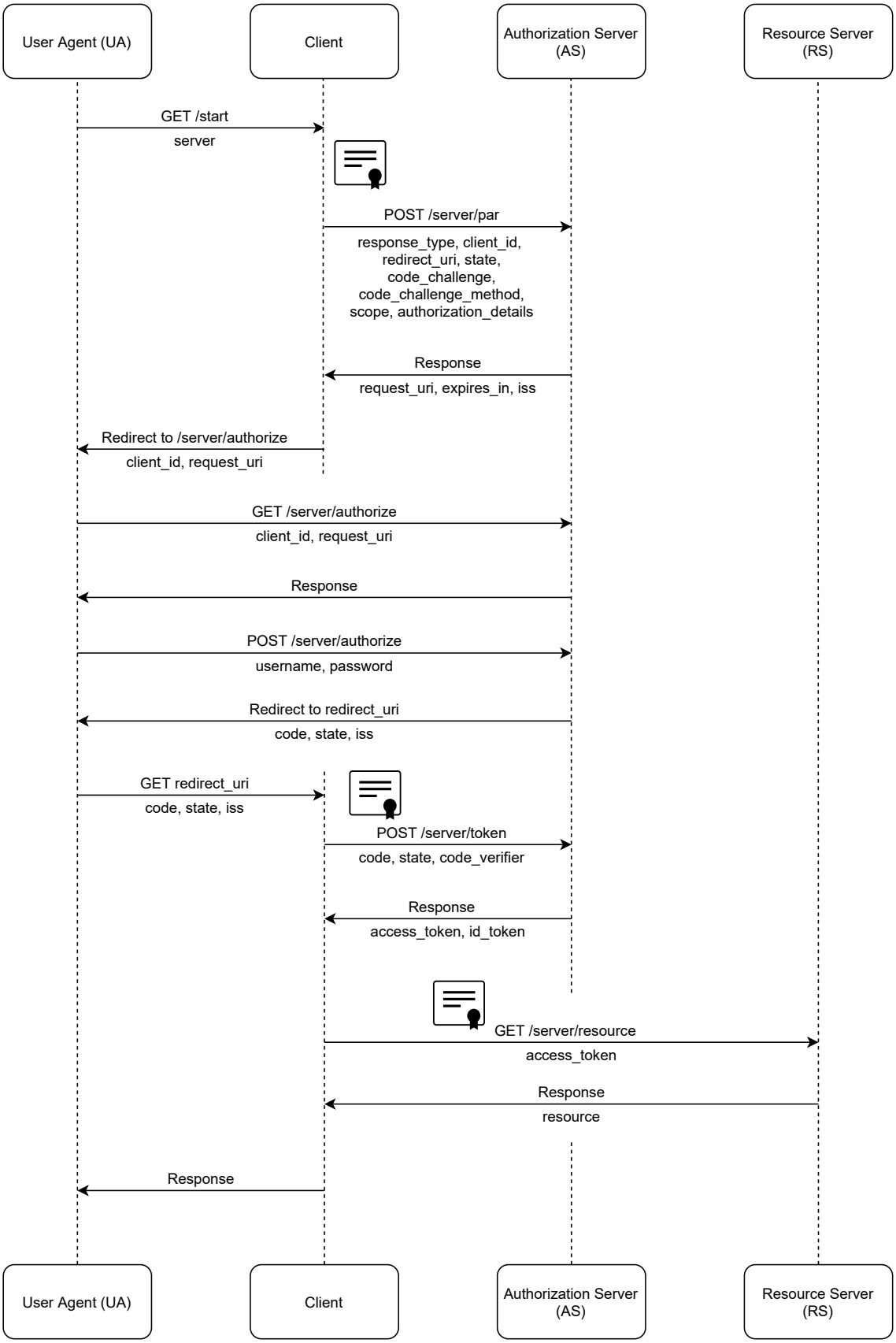

Figure 2.3: FAPI 2.0 Baseline Profile.

Sender-constraining *access tokens* (ATs) In order to prevent the usage of leaked ATs by unauthorized clients, ASs must constrain the issued access tokens to the clients the RO granted access to their resources. This can be done either using Mutual Transport Layer Security (MTLS), where the AT is bound to the certificate the client uses to authenticate itself to the AS.

Client Authentication In order for clients to verify their identity to the AS, they must use a method to authenticate themselves. One approach is using MTLS or, if the client has registered a public key, use the corresponding private key to sign a *JSON Web Token* (JWT) [21, Section 9]. Since MTLS can be used for both client authentication and token binding (see previous point), we implement only MTLS.

Proof Key for Code Exchange To bind the AC the proper client and prevent its use by malicious applications, Proof Key for Code Exchange (PKCE) must be used. In addition, the `code_challenge` must be the hashed variant of its corresponding `code_verifier` using the SHA-256 hash algorithm.

Sender-constraining Refresh Tokens In addition to sender-constraining ACs and ATs, RTs shall be sender-constrained as well.

`redirect_uri` in PAR As a result of using MTLS for client authentication and PAR for sending the complete Authorization Request in the back channel, there is no need for the client to pre-register redirect URIs at the AS.

`iss` Paramter in Authorization Responses In order to thwart IdP Mix-up Attacks [19, Section 3.2], the `iss` parameter corresponding to the value of `issuer` as defined in [7, Section 3.3] can be added to authorization responses in order to indicate the client which AS issued the AC.

Verifying ATs ASs have to provide RSs with some mechanism to check ATs in order to verify their validity, scope (including details from RAR if provided), integrity, sender-constraining, expiration, and revocation status. One method to achieve this is by providing an Introspection Endpoint [35].

Avoid Using HTTP 307 Response When using 307 as the status code for an HTTP response, Web browsers will not only perform a request to the given URL in the Location header with the given POST parameters, but will also reuse the POST parameters in the original request [2], which for an Authorization Response corresponds to the RO's credentials at the AS. A malicious client can thus obtain the login credentials of the RO if the AS uses the 307 Temporary Redirect status code.

No Open Redirectors An open redirector is an endpoint included as a parameter in an Authorization Response to the UA to which they are redirected without any validation [23, Section 10.15]. Since open redirectors allow a class of attacks, namely phishing and (in the event that wildcards are allowed in the authority component of the redirect URI) passing the AC to an endpoint under the attacker's control.

Advanced Profile

While the Baseline profile offers strong security guarantees, it fails in providing non-repudiation; an arbitrary client can claim that an Authorization Request sent under their name to an AS was forged by a malicious party, or an arbitrary AS can dispute the authenticity of an AC issued by them. The Advanced profile addresses this by mandating, in addition to the requirements listed by the Baseline profile, application-level signatures, thereby allowing any third party to verify whether, for example, a payment request was indeed authorized by the client or not.

As with the Baseline profile, the Advanced profile is in the draft stage and thus subject to changes after the publication of this thesis. For reference, this thesis deals with the profile at the git commit **94d45dd** [29].

Non-repudiation In order to provide non-repudiation, the Advanced profile mandates that clients and ASs sign Request Objects and Authorization Responses, respectively. To achieve this, *JWT Secured Authorization Request* (JAR) [36] and *JWT Secured Authorization Response Mode for OAuth 2.0* (JARM) [11] specify how Request Objects[4] shall be signed and how a signed Authorization Response should be constructed by the AS, respectively.

For signing Request Objects, clients and ASs must use JWS [6]. While JAR and JARM outline how Request Objects and Authorization Responses can be signed then encrypted, the Advanced profile only outlines signing. Therefore, our prototype only signs the Request Objects and Authorization Responses.

2.1.4 Extensions

The following sections briefly go into the various extensions which are used in by FAPI.

Proof Key for Code Exchange

A problem for clients running as native applications on the RO's device is that different applications can register the same custom URI as the honest client native application. When the Authorization Code Grant is in use, the Authorization Response can leak to these other applications who registered that custom URI. Thus, the AC is leaked to the malicious client.

To solve this, Proof Key for Code Exchange (PKCE) can be used, which introduces three new parameters, the Code Challenge, its Code Verifier, and the Code Challenge Method. The Code Challenge, which is either the same as its corresponding Code Verifier or the hash of the Verifier using SHA-256 (this is determined by the Code Challenge Method Parameter), is sent with the Authorization Request from the UA to the AS, and the AS binds the request to this Challenge. When the client attempts to exchange the AC for the AT, it sends the Code Challenge's corresponding Code Verifier, and the AS checks whether the Verifier (or its hash, again depending on the Challenge Method) matches the Challenge sent before. If it is a match, the AS sends back the AT.

[4] A Request Object in the context of JAR is "JWT [. . .] whose JWT Claims Set holds the JSON encoded OAuth 2.0 authorization request parameters." [36, Section 2.1]

Fett et al. [18, Section IV] demonstrated an attack on the supposed security offered by PKCE against leaked ACs called the *PKCE Chosen Challenge Attack*. The mitigation involved signing the Authorization Request, through which a signed Request Object (see Section 2.1.4). A caveat of this fix, however, is that the public client, by definition, cannot securely store long-term secrets, which would include the private key necessary for signing the Authorization Request in order to guarantee to the AS that the PKCE challenge indeed originated from the honest client.

Mutual Transport Layer Security

In order to ensure that only the intended client can use an issued AT, the AS can bind the AT to the client's own X.509 [45] certificate. With this, only the client in possession of the certificate's private key can use the certificate at the RS, since otherwise the client authentication at the TLS level would fail. This ensures that the AT is sender-constrained. Furthermore, the client's certificate can be used to authenticate the client to the AS, allowing the AS to unambiguously verify the identity of the client. Both of these methods are specified using Mutual Transport Layer Security [5].

`iss` Parameter

Fett et al. [19, Section 3.2] describes a form of attack called the IdP Mix-Up Attack. Briefly described, a malicious IdP acting as a Man-in-the-Middle between the honest UA of the RO and the honest client confuses the client about the true IdP at which the RO wishes to authenticate themselves. This occurs by hijacking the connecting at the beginning of the OAuth 2.0 flow and changing the true IdP to which the RO would authenticate themselves at. Another variant of the attack does not require a Network Attacker capable of acting as a Man-in-the-Middle, but does assume that the RO wishes to log in under the attacker's own IdP [28].

The result of this attack is that the malicious IdP gains the AC issued to the honest client with which an access token can be redeemed at the honest IdP.

The core problem of this vulnerability is that the client learns nothing about the origin of the Authorization Response. Therefore, an effective fix is that honest IdPs attach their identity as a new parameter in the Authorization Response, aptly named `iss` [39]. Its value must be the same as the value of the `issuer` parameter as defined in [7, Section 2].

An important note: The original *oauth-proto* included an incorrect implementation of this requirement, presumably because the draft was first published in early January of 2021, well after development *oauth-proto* has finalizeds. Therefore, any checks in the original prototype of the `iss` parameter are likely to throw errors. This further emphasizes the importance of avoiding duplicates in the codebase and instead centralize logic which will be reused, since such bugs can only be fixed by scouring the codebase for occurrences of the duplicate block of code responsible for the checks, fixing the bugs, and re-testing the already existing prototype to check for regressions. This re-testing, however, is made even more unfeasible by the lack of automated test suites.

Pushed Authorization Requests

As is typical for OAuth 2.0, Authorization Request parameters, including `client_id` and `scope`, are sent as query parameters from the UA to the AS via a redirection initiated by the client (See [23, Section 4.1.1]). This poses challenges regarding security, however, as no cryptographic integrity or authenticity is offered since no confidential data is transferred between the RO and client before authentication. Also, these parameters are passed in the clear at the UA, which becomes a problem if network data is stored in logs and they leak, which in Open Banking scenarios is unacceptable.

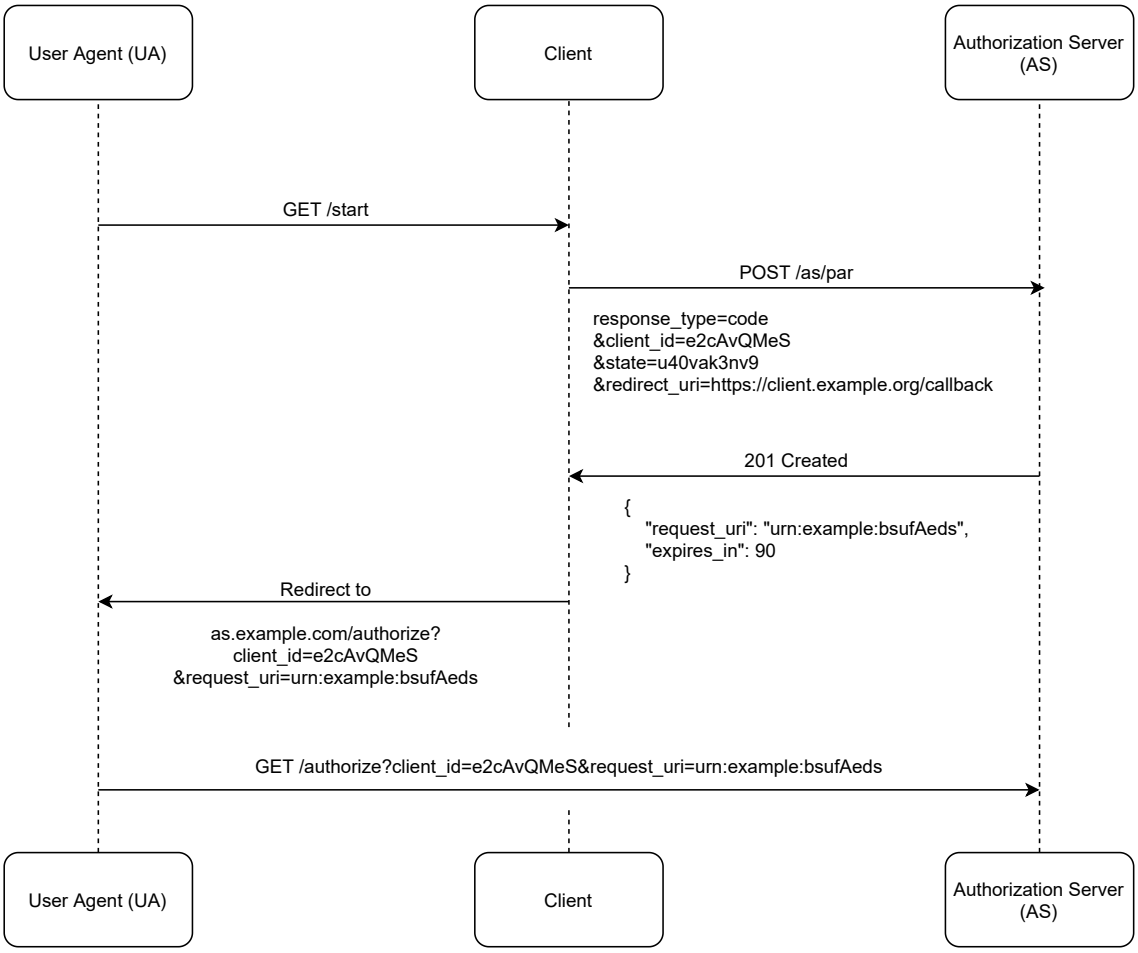

Figure 2.4: Pushed Authorization Requests.

Pushed Authorization Requests (PARs) [37] solve this by mandating the client to push these request parameters directly to the AS. The flow of an Authorization Request is shown in Figure 2.4.

The AS then stores this request in some form and generates a `request_uri`, a URI identifying the Authorization Request in question, and sends it back to the client as a response. Instead of the client redirecting the RO by their UA to the AS with the query parameters, they are redirected with just the `client_id` and `request_uri` to the AS.

This approach to transmitting Authorization Request properties provides several benefits:

Listing 2.1 Example of RAR [9, Section 2].

```
{
    "type": "payment_initiation",
    "locations": [
        "https://example.com/payments"
    ],
    "instructedAmount": {
        "currency": "EUR",
        "amount": "123.50"
    },
    "creditorName": "Merchant123",
    "creditorAccount": {
        "iban": "DE02100100109307118603"
    },
    "remittanceInformationUnstructured": "Ref Number Merchant"
}
```

Client Authentication from the Get Go Before any further interaction between the RO and the AS can take place concerning an Authorization Request, the AS can first check the authenticity of the client. Thus, attempts by attackers to spoof honest clients can be thwarted early in the process before the RO even has a chance to delegate access to their resources to said attacker.

Confidentiality and Integrity of Request Data Since the request data are transmitted using TLS directly between the client and AS without having to first pass through the front channel, the confidentiality and integrity of the data at this stage is protected.

Since the specification for PARs is still in the Internet-Draft stage, changes can occur at any time. For reference, this thesis deals with **draft-ietf-oauth-par-05**.

Rich Authorization Requests

While the `scope` parameter is sufficient for static use-cases, such as requesting access to the cloud storage of the RO, it fails in situations where the exact parameters are dynamic. This is especially the case with banking and transactions, as transactions details can vary widely from transaction to transaction, such as the amount to be transferred, the recipient of this payment, the currency, and more.

To solve this, *Rich Authorization Requests* (RARs) [9] allow clients to define fine grained information regarding their Authorization Request. Listing 2.1 shows an example of a RAR for initiating a payment.

The `type` parameter is the only mandatory key in a RAR. All others are optional but recommended since this makes RAR usable across several APIs.

Since the specification for RARs is still in the Internet-Draft stage, changes can occur at any time. For reference, this thesis deals with **draft-ietf-oauth-rar-03**.

JWT Secured Authorization Request

While PARs are concerned with packaging the Authorization Request data into a single payload to be transmitted at once, it fails at offering non-repudiation. For example, a RO can claim that an initiated payment transaction was a forgery.

JWT Secured Authorization Requests mitigate this issue by allowing the clients to cryptographically sign the Request Object. A Request Object is a JSON Web Token [8] which holds the Authorization Request parameters as JWT Claims [8, Section 4]. The Request Object can be optionally signed using JSON Web Signature [6] and optionally encrypted using *JSON Web Encryption* [24].

JAR mandates that Request Objects be signed. Encryption is optional. With that being said, the Advanced profile is the only profile of FAPI 2.0 that mandates the use of JAR, not the Baseline profile. The Advanced profile furthermore makes no mention of encrypting the Request Object, only signing the Request Object. Thus we will only consider JWS and not JWE in this work.

JWT Secured Authorization Response Mode for OAuth 2.0

While JAR ensures that the integrity of Authorization Request parameters is cryptographically protected by mandating clients to sign the Request Objects, the same cannot be said for Authorization Responses. JWT Secured Authorization Response Mode for OAuth 2.0 is the equivalent of JAR, but for ASs. It provides integrity protection of the Authorization Responses by mandating that the AS sign the Authorization Response. Furthermore, non-repudiation can be offered if asymmetric signatures are employed, such as using the AS's public key to verify an Authorization Response signed by the AS using the AS's private key.

Symmetric signatures only offer integrity protection but not non-repudiation, because third parties cannot verify the message's authenticity. Since more than one party has the symmetric key, any one of those parties (the client and the AS in this case) could have signed the Authorization Response. In addition, it would require that the third party have the symmetric key, which as per Section 10.1 of [21] is the `client_secret`, which should understandably be kept secret.

2.2 Related Work

While research thus far has been primarily focussed on the formal analysis of the underlying OAuth 2.0 Authorization Framework [19] or empirical analysis of deployments of OAuth 2.0 in the wild [46], the practical demonstrations of these attacks and, more generally speaking, the OAuth 2.0 and OIDC flows themselves, has only been addressed by Erdemann [12] and Mainka [27]. Furthermore, given that FAPI 2.0 is still in its infancy it may be some time before more IdPs and RPs adopt it, whether in the form of migration from existing vanilla OAuth 2.0 or OIDC implementations, or as an upgrade from the more mature FAPI 1.0, which has indeed undergone rigorous formal analysis using an extensive model of the Web [20] based on the Dolev-Yao model of cryptographic protocols [10]. Mainka presents an implementation of an *OpenID Attacker*, a malicious IdP. The aim is to demonstrate several attacks within the context of OIDC. However, as noted in [12, Chapter 5], OpenID Attacker does not implement concrete attacks whose effects can be directly observed and interpreted by the user. Furthermore, it focused primarily on the OIDC specification.

3 Analysis of Existing Prototype

The goal of this thesis is to serve as a demonstration of the FAPI 2.0 (referred to as just FAPI for brevity) Baseline and Advanced Profiles as well as provide an easy-to-understand codebase with options for extensibility and guidelines on doing so.

To that end, we begin by examining the existing prototype and understanding the codebase. This allows us to understand how we can extend it to fulfill the functional requirements for FAPI. With this knowledge, we implement the functionality needed for the FAPI Baseline and Advanced profile flows. This is covered in Section 3.1.

During this examination, we realize multiple rooms for improvement for the codebase, and we propose and implement several of them and demonstrate the benefits introduced by these changes. These improvements are outlined in Section 3.2.

The software implementation developed throughout this work, which we will refer to from here on out as *fapi-proto*, builds upon a prototype produced by Erdemann [12], itself based on a previous prototype by Fett [15] (the former will be referred to as *oauth-proto* and the latter as *proto6749* for brevity).

3.1 Existing Prototypes

proto6749 concerned itself with raw software implementations of OAuth 2.0, primarily the implementation of an AS, as well as the basics of PAR, specifically an endpoint for it, and RAR. *oauth-proto* extends functionality by implementing OIDC (Authorization Code Grant only), clients within the project, a logger for the executed flows, and a Web interface.

3.1.1 Web Server

The prototype is implemented as a Django[1] project, a Web framework based on Python. Django is based on the Model-Template-View architectural pattern [13] where the server is responsible for both the front-end and the back-end. The *model* represents the data stored in Django's database of choice, the *view* represents the subset of data presented to the user, and the *template* represents the presentation of the view to the user, i.e., how the data is displayed to the user.

[1] https://www.djangoproject.com/

3.1.2 Database

The choice of database to store the models used by Django is PostgreSQL[2]. Django uses an object–relational mapping (ORM) to map native Python objects to data records in PostgreSQL.

A built-in feature of Django is the ability to run a development server, which auto-restarts upon detecting a change to one of its files, greatly simplifying development[3].

3.1.3 Reverse Proxy

To more accurately simulate the behavior of OAuth 2.0, OIDC, and the FAPI, a reverse proxy is employed to serve as a middleman between the UA and the Django Web server. nginx was chosen since *proto6749* as the reverse proxy of choice, and this has remained the same for *oauth-proto*. The connection between the UA and nginx is secured using TLS, whereas the one between Django and nginx is in plain HTTP.

3.1.4 X.509 Certificates

To generate the public certificate needed by nginx such that the UA recognizes the connection as secure, mkcert [43] is used to generate a development Certificate Authority (CA). This is achieved by first installing the mkcert CA in the system trust store and then issuing an X.509 certificate signed by the mkcert CA together with its corresponding private key.

In addition to TLS certificate for the reverse proxy, mkcert is also used to generate certificates for the client, which will be important later when using MTLS for client authentication in the FAPI.

3.2 Rooms for Improvement

During our examination of the codebase, we discover several rooms for improvement. They aim at improving aspects such as understandability, readability, maintainability, and better code quality.

Data Types for Various Constructs Python and JavaScript belong to the class of dynamically typed languages. A fundamental property of both these languages is that they do not require providing data types for constructs such as variables and return value types for functions before runtime. As such, it can become difficult to correctly infer, for example, what properties a variable has.

While this poses less of an issue for smaller codebases, *oauth-proto* is not what could be considered a small codebase. For instance, the `main.js` file representing the JavaScript logic for the main end-user Web interface to the prototype constitutes of over 700 lines of code.

[2] https://www.postgresql.org/

[3] Note that this feature does *not* consider file additions and changes to static files such as JavaScript files and images.

Furthermore, by providing data types for variables and information regarding function signatures, the codebase becomes less prone to runtime errors due to operations involving incompatible types or non-existing properties that are not caught because of lack of data types before runtime. We demonstrate that after inferring data types for various variables, we were able to avoid potential bugs.

Adding Meaningful Constants and Enum Types Our analysis has further revealed the use of Universally Unique IDs (UUIDs) throughout the code. These were hard-coded and came with no documentation as to what entities they are supposed to represent, which leads to having to make assumptions and guesses about the nature of these objects. Furthermore, several constructs could be better represented by more clear enum types, such as the chosen flow. We fix this by declaring prototype-wide constants that accurately describe the objects at hand.

Deploying Linters Linters are static code analysis tools used to identify suspicious code constructs that may lead to runtime errors, such as potential dangling references or the possibility of accessing a pointer that points to nothing. We demonstrate how, together with explicitly providing data types to variables, the usage of a linter, specifically a type checker, potential runtime errors were avoided.

Removing Duplicate Code The use of duplicate code poses several issues. First, it makes the code harder to understand by external developers; instead of having one function representing the desired logic, the same logic is written in several places throughout the codebase. Second, this makes it harder to fix bugs where duplicate code is involved since developers have to scour the codebase looking for these duplicate blocks of code in order to fix them. We fix this by extracting often used blocks of code and providing clear documentation on their usage and signatures.

Break up Large Functions Several methods and functions spanning hundreds of lines of code were discovered throughout the analysis. Of note is the main function of the YAML Ain't Markup Language (YAML) logger. While one can argue that such functions could be left untouched throughout development, this becomes problematic when bugs caused by such functions are discovered. We break up much of the existing codebase and maintain this standard on our own implementations of the FAPI.

Implementing Code Best Practice A quick inspection using the Integrated Development Environment (IDE) PyCharm[4] revealed several warnings related to smelly code. We improve upon this by migrating the entire codebase to TypeScript[5] and splitting the file into more manageable modules. In addition, we utilize JavaScript's "strict mode" [40] which paves the way for cleaner, more understandable JavaScript code.

[4] https://www.jetbrains.com/pycharm/
[5] https://www.typescriptlang.org/

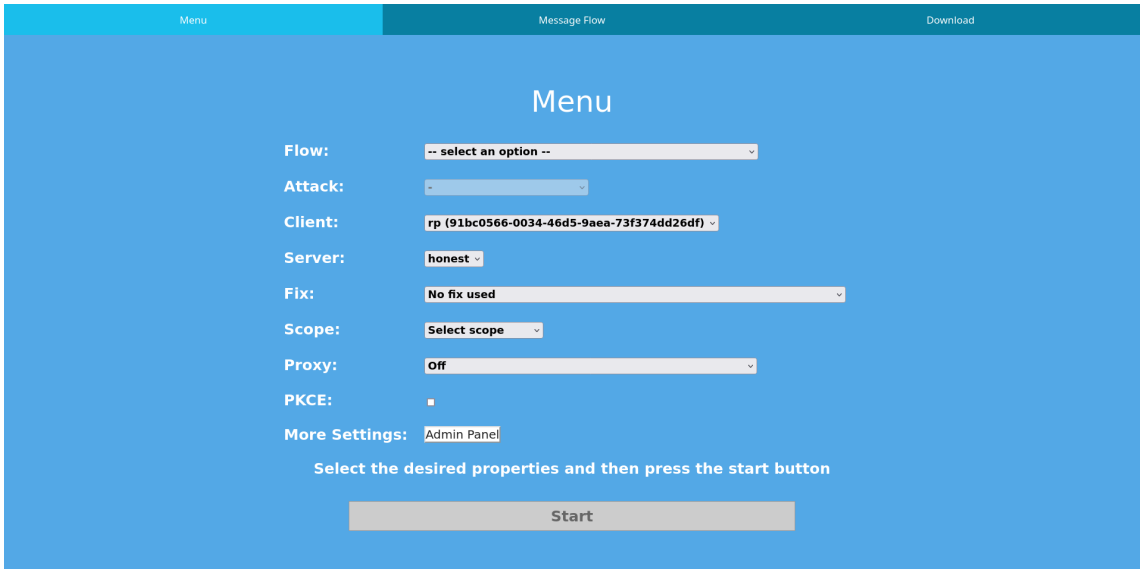

Figure 3.1: The main end-user interface of *oauth-proto*.

3.3 Happy Path

In order to familiarize the reader with the usage of *oauth-proto*, we briefly describe the happy path of the prototype as present in *oauth-proto*, i.e., the default usage of the prototype assuming no errors or exceptions occur.

3.3.1 Setting up

To set up the prototype, the required PostgreSQL database server must be installed on the local machine and a user set up using which Django will gain access to the database to store information regarding its models, such as ASs, clients, and sessions.

Next, a Python virtual environment[6] is set up along with the project's dependencies.

Parallel to this, the nginx server is set up as a reverse proxy to handle TLS from UAs to the server, since the test server runs only using HTTP.

3.3.2 Running the program

Once this is complete, the user can access the primary end-user interface shown in Figure 3.1 at the Uniform Resource Locator (URL) `https://localhost/start`, where the tab "Menu" is displayed by default. From this interface, the user selects the following properties for the flow to be simulated:

Flow The desired flow to run. For example, vanilla OAuth 2.0 Authorization Code.

[6] A Python virtual environment is "a self-contained directory tree that contains a Python installation for a particular version of Python, plus several additional packages." [1]

Attack The attack, if any, to execute on the chosen flow.

Client The client which the user wishes to authorize access to their resources

Server The AS of choice.

Fix The fix to use. For example, a fix for the IdP Mix-up attack by adding the `iss` parameter in the Authorization Response.

Scope The scope to be selected.

Proxy The Network Attacker. Users can choose whether to activate them or not and, if activated, which modification to the messages the user may simulate.

PKCE Whether the client should activate and make use of PKCE.

More Settings Opens a link to the Django admin panel where the user can modify various properties of the stored models and users, including adding new ones or removing existing models.

Once the user has chosen the properties of the flow, the user clicks "start" and follows the prompts shown. In the case of OAuth 2.0, for example, the user is redirected to the AS's Authorization Endpoint to login and verify their identity. In addition, they are shown the data to which the client is delegated access. This depends on whether OAuth 2.0 is used in conjunction with OIDC or not.

Once the user confirms the authorization, they are redirected to the client with the AC which the client redeems for the AT. The resource obtained in the prototype is a representation of the session in question, including, but not limited to, the AS, the client, the Code Challenge and Code Challenge Method (if PKCE were activated).

The flow of messages is logged and recorded in a separate tab under the name of "Message Flow". Users can refresh the view to display any new messages as well as download a copy of the YAML for download, either in detail or with abbreviations for the parameters. The message flows are stored in YAML format and is encoded in AnnexLang [14], a markup language for describing protocol flows and exporting them to semantically equivalent LaTeX code.

4 Implementation

Here, we describe on a high level the modifications we made to the prototype. Section 4.1 outlines the steps taken before work on the FAPI profiles have taken place to address the points mentioned previously in Section 3.2. We also describe a few more improvements and their impact on development and deployment. Section 4.2 describes the changes and additions made in order to fulfill the requirements as laid out before in Section 2.1.3. Section 4.3 analogously does the same for the Advanced profile.

4.1 Setting up the Development Environment

The first step in the implementation was setting up the development environment in such a form that it offers the following:

Enforcing Common Standards As discussed in our analysis, the lack of use of either linters or style checkers has led to poor code quality and difficulty in understanding the codebase. To solve this, we enforce various linters and style checkers across the entire codebase.

Containerization of the Prototype While the original approach of manually setting up the running environment, including the PostgreSQL server and nginx, it faces issues when attempting to reproduce it on another machine. The adage "it works on my machine" is especially problematic given that we cannot make assumptions about the running environments of other end-users and developers alike. We utilize Docker[1] to provide an easily reproducible software system that can also be used for development.

Using Data Types In order to help developers better understand the properties and type of the various variables as well as functions and their signatures, we explicitly provide data types for both the front-end and back-end.

4.1.1 Style Checkers

To aid in development and enforce certain style guides so that the readability of the code is satisfactory, we utilize the following tools:

[1] https://www.docker.com/

black[2] A style guide checker and formatter for Python adhering to the standard style guide laid out in [34]. Its advantage over other formatters such as autopep8[3] or yapf[4] is its popularity (over 20k stars on GitHub), which means that it is more likely that Python users inspecting our prototype would be familiar with the style imposed by black. Furthermore, its lack of configurability means that most Python projects using black are bound to look the same regardless of the project at hand.

Prettier[5] This is the equivalent of black, but applied to languages such as JavaScript and TypeScript. Like black, its relative popularity and opinionated nature paves the way for similar-looking code across different projects.

4.1.2 Linters

Linters are central to a software project where susceptible and smelly code can be detrimental to the extensibility, maintainability, and ability to comprehend the codebase. We utilize the following tools to assist in detecting bad code constructs and thus solve them:

flake8 A wrapper around three separate Python modules, flake8[6] analyzes the Python codebase and checks for suspicious code constructs. For example, it warns of misuse of variable declarations which could lead to null pointer exceptions under certain conditions

pyright A type checker for Python written by Microsoft, pyright[7] checks for type incompatibilities, such as attempting to return a Boolean and store its result in a variable previously declared as a string.

ESLint[8] A popular linter for JavaScript and TypeScript, ESLint checks codebases for suspicious or problematic patterns. It is characterized by the multitude of configuration options available. For our prototype, we decided on applying the Standard[9] style. A special package for TypeScript called `ts-standard` is included to ensure Standard works for the TypeScript codebase (see Section 4.1.6).

Other tools worth mentioning but are of lesser importance than the ones outlined above include, but are not limited to, isort[10], pydocstyle[11], and autoflake[12]. For a full list, refer to the accompanying readme file.

[2] https://github.com/psf/black
[3] https://github.com/hhatto/autopep8
[4] https://github.com/google/yapf
[5] https://prettier.io/
[6] https://github.com/PyCQA/flake8
[7] https://github.com/microsoft/pyright
[8] https://eslint.org/
[9] https://standardjs.com/
[10] https://github.com/PyCQA/isort
[11] https://github.com/PyCQA/pydocstyle
[12] https://github.com/myint/autoflake

4.1.3 Pre-commit Hooks

To further ensure that checks by these checkers, we use pre-commit git hooks[13] which run these checkers against the codebase before a commit takes place. Assuming these checks are not bypassed by future developers, this should impose the same format and guidelines on the codebase.

4.1.4 Containerization of the Prototype

With Docker, we create so-called images. These images represent a set of filesystem changes that are layered on top of one another. From images, containers are created, which represents the program in execution. A fitting analogy to images would be programs and their containers as the corresponding processes from those programs.

Since out prototype consists of several components, namely the Django server, nginx reverse proxy, and PostgreSQL database, we use Docker Compose[14] to orchestrate the execution of these three services. These services are as follows:

web This represents the Django server. It utilizes a bind mount so that any changes to the files on the local filesystem are immediately reflected in the Docker container. Since we utilize the Django development server which uses a file watcher to notify of any changes to its files, the server can auto-reload upon saving changes to the server's files. This saves a great deal of effort and time and leads to faster development since the Docker container doesn't have to be rebuilt each time a change is made.

db This represents the PostgreSQL database. the username and password used by Django to access it are defined in the `docker-compose.yaml` and can be changed if desired. In this case, one must ensure that the changes are reflected in Django's settings.

nginx This represents the nginx reverse proxy. All requests made to `https://localhost`, including the `/start` path for the Web interface and the `/client` and `/server` paths are first routed over this server, even from clients (which are Django views themselves) to ASs.

For networking, the network driver is set to "host". This means that Docker does *not* isolate the network stack of the containers from the rest of the system. Note, however, that this feature is **only available on Linux systems. Windows and macOS are not supported.**[15]. We have attempted to utilize the "bridge" network mode which would have worked on all platforms. However, the UA being in one network stack and isolated from the remainder of the prototype have lead to issues regarding how requests are transmitted from and to ASs and clients.

[13] https://pre-commit.com/
[14] https://docs.docker.com/compose/
[15] https://docs.docker.com/network/host/

4.1.5 Python Virtual Environment Management

To ensure that Python builds are deterministic and thus avoid any errors that could arise due to breaking changes from updates, we utilize Poetry[16]. Poetry is packaging and dependency manager for Python that allows the automatic creation of a Python virtual environment isolated from the system-wide Python installation.

Information regarding packages and their corresponding versions are stored in the `pyproject.toml` file located in the `<project root>/app/` directory. Development dependencies are separated from deployment dependencies. Development dependencies are not installed when building the Docker image to ensure the image size is as small as possible.

4.1.6 Front-end Codebase Overhaul

The JavaScript front-end codebase has been improved. Functionality is now split across coherent ECMAScript modules, each written for a well-defined task. In addition, strict mode is used which assists in avoiding error-prone constructs, such as undeclared variables polluting the global scope.

To support compile-time type checking, we migrate the entire codebase to TypeScript, a strict syntactical superset of JavaScript. Each variable is given an explicit data type to help detect and resolve potential type incompatibility issues before runtime.

4.1.7 Reorganization of Codebase

To improve the manageability and provide developers with a better overview of the codebase, several refactoring changes have been undertaken. First, blocks of code related to a single function, e.g., constructing the payload for PAR, are extracted into their own functions, together with documentation. This lends the way for code that is easier to read and navigate through.

4.1.8 Usage of Enum Types

Enum data types are used for variables whose set of possible values is finite and which can be represented by an expressive name. For example, flows are now specified by enums such as `Flow.FAPI_BASELINE`. This helps keep the code self-describing and easier to understand, which also makes it easier to debug, especially with external developers who are unfamiliar with the codebase. Listing 4.1 shows the enums defined in our prototype.

4.1.9 Explicit Data Types for Python Constructs

We provide data types for various constructs in Python, including class and instance attributes, and function parameters and their return values.

[16] https://python-poetry.org/

Listing 4.1 Enums defined in the prototype.

```
class Flow(IntEnum):
    OAUTH_CLIENT_CREDENTIALS = 0
    OAUTH_RESOURCE_OWNER_PASSWORD_CREDENTIALS = 1
    OAUTH_IMPLICIT = 2
    OAUTH_AUTHORIZATION_CODE = 3
    OIDC_Authorization_Code = 4
    FAPI_ONE_CIBA = 5
    FAPI_TWO_BASELINE = 6
    FAPI_TWO_ADVANCED = 7


class Fix(IntEnum):
    NONE = 0
    IDP_MIXUP_ATTACK = 1
    CUCKOO_TOKEN_ATTACK = 2


class ResponseType(Enum):
    NONE = "none"
    AUTHORIZATION_CODE = "code"
    ACCESS_TOKEN = "token"
    ID_TOKEN = "id_token"
    ID_TOKEN_ACCESS_TOKEN = "id_token token"
    AUTHORIZATION_CODE_ID_TOKEN = "code id_token"
    AUTHORIZATION_CODE_ACCESS_TOKEN = "code token"
    AUTHORIZATION_CODE_ID_TOKEN_ACCESS_TOKEN = "code id_token token"


class Scope(Enum):
    OPENID = "openid"
```

Listing 4.2 Bug in code from *proto6749*.

```
def expires_in(self):
    return ((now() - self.created) - self.MAX_LIFETIME).seconds

def expired(self):
    return self.expires_in < 0
```

This has assisted us in resolving a bug in the codebase. Listing 4.2 shows an example of a function in the `Session` model named `expired` attempting to call a function `expires_in`. Upon closer inspection, it is apparent that the actual behavior would not match the expected behavior; the function itself would be returned, not the time remaining before the session is considered expired.

Using a type checker such as pyright, it throws a warning since the operation seems to be involving incompatible types, namely an integer, and a function. The bug can be therefore before it has a chance of leading to problems, whether during development or deployment.

4.2 Baseline Profile

As covered in Section 2.1.3, the Baseline profile mandates the adherence to several restrictions aimed at providing several security guarantees. We discuss below how each requirement is to be fulfilled by the *fapi-proto*:

Authorization Code Grant

This requirement is straightforward; when choosing the flow as shown in Figure 3.1, the user chooses the Baseline profile. Since only the Authorization Code Grant of OAuth 2.0 is permitted, the prototype always executes this flow.

Pushed Authorization Requests only

This was one of the more challenging requirements to be implemented. The lack of libraries for implementing PAR meant that the entire logic had to be written form scratch, and care was required if our implementation of PAR is to conform to its specification [37].

Nonetheless, this is achieved by adding another if-else branch to the client. In addition to the flows already implemented in *oauth-proto*, the client checks if the selected flow corresponds to the flow. Helper methods construct the necessary data payloads and sends them to the server in order to process the Authorization Request. Once that is done, the client redirects the UA to the AS with the `request_uri` and the client's `client_id`.

Support Rich Authorization Requests

The issue with RAR is that it is highly dependent on the concrete use case of it. Furthermore, analysis of the codebase has revealed that consent templates, which represent HTML templates for various scopes of data, e.g., displaying claims when OIDC is in use, include one template for authorization details Listing 4.3. It seems to be the case that the template involves the use of bank accounts.

As such, we implement a basic BankAccount Django model with which this information can be represented. In addition, we add a corresponding Pydantic[17] model to enable run-time validation and parsing of the data.

[17] https://pydantic-docs.helpmanual.io/

Listing 4.3 Authorization Details Template.

```
{
    "match": {
        "authorization_details": {
            "type": "account_information",
        }
    },
    "template": """
    You are providing '{{ client.name }}' access to the following account information:
    <ul>
        <li>Type of requested data: {{ authorization_detail.type }}</li>
        <li>
            Requested Actions:
            <ul>
                {% for authorization_request_action in authorization_detail.actions %}
                <li>{{ authorization_request_action }}</li>
                {% endfor %}
            </ul>
        </li>
        <li>Requested resource servers:
            <ul>
                {% for authorization_request_location in authorization_detail.locations %}
                <li>{{ authorization_request_location }}</li>
                {% endfor %}
            </ul>
        </li>
        <li>Datatypes of requested data:
            <ul>
                {% for authorization_request_datatype in authorization_detail.datatypes %}
                <li>{{ authorization_request_datatype }}</li>
                {% endfor %}
            </ul>
        </li>
    </ul>
    """,
},
```

Sender-constraining *access tokens* (ATs)

In order to constrain ATs to the intended clients, we use MTLS. This is achieved by generating an X.509 certificate along with its private key. To that end, we make use of the tool *mkcert*[18], which allows the generation of X.509 certificates for development purposes.

One thing to note is that the Certificate Authority (CA) is installed by mkcert in the system trust store using a special command. This allows Web browsers to trust the certificate presented by the client to the AS.

[18] https://github.com/FiloSottile/mkcert

Client Authentication

This is also achieved using MTLS. The original implementation of the prototype, *proto6749*, pre-registers the X.509 certificate and binds it to the client Django model. When the AS receives an Authorization Request either at the PAR Endpoint, Token Endpoint, or Introspection Endpoint and the client chooses to authenticate itself using MTLS, the AS looks in the PostgreSQL database and checks whether the transmitted X.509 public certificate matches the pre-registered one.

To validate the client's possession of the private key corresponding to the X.509 certificate's public key, TLS is utilized [5, Section 2.2]. If Proof-of-Possession is demonstrated, the authentication is considered successful.

Proof Key for Code Exchange

While MTLS binds the AT to the client, PKCE binds the AC. This thwarts attacks where the attacker steals the AC, since an AT is only issued upon sending the correct Code Verifier.

To that end, there exists a Python library that implements much of the logic needed for PKCE. *pkce*[19] is a Python module that implements rudimentary functionality such as generating PKCE parameters. The actual verification on the AS's side is achieved by a hand-written function existing from *oauth-proto*.

Sender-constraining Refresh Tokens

The previous logic in *oauth-proto* used an AT expiry time equal to 9,999,999 seconds, just below four months. Furthermore, the expiry of the RT was actually not being checked in the first place. Since it would only make sense to consider RTs when ATs ever expire, let alone within a short time window (generally a couple of minutes), we believe this feature is not critical to the understanding of either the FAPI 2.0 Baseline or Advanced profiles. Nonetheless, it should be mentioned to highlight the necessity of binding RTs to their intended clients.

`redirect_uri` in PAR

Previously, clients pre-register their redirect URIs at the AS. However, this is made unnecessary when pairing PARs with MTLS for client authentication. Since the client authenticates themselves at the PAR Endpoint, the AS can be sure that the client indeed has control of the redirect URIs contained in the payload of the PAR.

`iss` parameter in Authorization Responses

In order to thwart IdP Mix-up attacks, the AS informs the client who is to receive the AC of their (the AS's) identity. We modify the respective blocks of code responsible for generating the Authorization Response URI to reflect this change.

[19] https://pypi.org/project/pkce/

Verifying ATs

The prototype is not intended to exactly mirror every possible real-life scenario. One consequence of this simplification is the fact that the RS is the same as the AS. With this, the Introspection Endpoint, while defined, is not made extensive use of, and always returns `True` whenever an AT is sent for introspection.

Avoid Using HTTP 307 Response

This is achieved by simply not making use of this redirect, and instead using other status codes for redirection. We use the HTTP 302 Found status code, which does not elicit the Web browser to resend the POST parameters of the original request, thus avoiding leaking the RO's login credentials to the client.

No Open Redirectors

This requirement is simply fulfilled by ensuring that no redirections are dependent on query parameters.

4.3 Advanced Profile

The Advanced profile mandates the same requirements on the Baseline, with the addition of JAR and JARM for non-repudiation purposes. Since no Python libraries offer functionality of either of these specifications (except for signing using JWS, which is provided by the Python library *python-jose*[20]), we opt to use the same flow we wrote for the Baseline profile, using a simple Boolean flag to determine whether the extensions for the Advanced profile are to be used.

[20] https://pypi.org/project/python-jose/

5 Conclusion

The output prototype deals with the demonstration of the FAPI 2.0. To achieve this, an existing prototype focused on OAuth 2.0 and OIDC was extended to support the Baseline and Advanced profiles. A demonstration using the already available Web interface can be executed. Protocol runs can be downloaded as YAML files using AnnexLang as markup, which allows for export into LaTeX. In addition to the end-user's perspective, software and API developers can inspect the codebase and make use of the extensive documentation to gain a clearer picture of how FAPI 2.0 works, which should assist them when integrating the FAPI into their own organizations. In addition, several improvements presented and implemented in the software should serve as a compass for development and emphasize the need for guidelines on architecture and usage of toolchains which support the software development process.

Bibliography

[1] *12. Virtual Environments and Packages — Python 3.9.4 Documentation*. URL: https://docs.python.org/3/tutorial/venv.html (visited on 05/02/2021) (cit. on p. 36).

[2] *307 Temporary Redirect - HTTP | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/307 (visited on 04/26/2021) (cit. on p. 26).

[3] Auth0. *Confidential and Public Applications*. Auth0 Docs. URL: https://auth0.com/docs/ (visited on 04/24/2021) (cit. on p. 20).

[4] "Bitcoin - Open Source P2P Money". In: (Mar. 2021). URL: https://bitcoin.org/en (cit. on p. 15).

[5] J. Bradley, B. Campbell, T. Lodderstedt, N. Sakimura. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. URL: https://tools.ietf.org/html/rfc8705 (visited on 05/03/2021) (cit. on pp. 28, 46).

[6] J. Bradley, N. Sakimura, M. Jones. *JSON Web Signature (JWS)*. URL: https://tools.ietf.org/html/rfc7515 (visited on 04/27/2021) (cit. on pp. 27, 31).

[7] J. Bradley, N. Sakimura, M. Jones. *OAuth 2.0 Authorization Server Metadata*. URL: https://tools.ietf.org/html/rfc8414 (visited on 04/26/2021) (cit. on pp. 26, 28).

[8] J. Bradley, N. Sakimura, M. B. Jones. *JSON Web Token (JWT)*. URL: https://tools.ietf.org/html/rfc7519 (visited on 05/02/2021) (cit. on p. 31).

[9] B. Campbell, T. Lodderstedt, J. Richer. *OAuth 2.0 Rich Authorization Requests*. URL: https://tools.ietf.org/html/draft-ietf-oauth-rar-03 (visited on 05/02/2021) (cit. on p. 30).

[10] D. Dolev, A. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* 29.2 (Mar. 1983), pp. 198–208. ISSN: 1557-9654. DOI: 10.1109/TIT.1983.1056650 (cit. on p. 31).

[11] *Draft-02: Financial-Grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)*. URL: https://openid.net/specs/openid-financial-api-jarm-ID1.html#terms-and-definitions (visited on 04/27/2021) (cit. on p. 27).

[12] M. Erdemann. "Eine prototypische Protokollimplementierung des OAuth 2.0 Protokolls mit Demonstration von Angriffen". In: (2020). In collab. with U. Stuttgart, U. Stuttgart. DOI: 10.18419/OPUS-11358. URL: http://elib.uni-stuttgart.de/handle/11682/11375 (visited on 04/25/2021) (cit. on pp. 16, 31, 33).

[13] *FAQ: General | Django Documentation | Django*. URL: https://docs.djangoproject.com/en/3.2/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names (visited on 04/28/2021) (cit. on p. 33).

[14] D. Fett. *Danielfett/Annexlang*. Oct. 30, 2020. URL: https://github.com/danielfett/annexlang (visited on 05/02/2021) (cit. on p. 37).

[15] D. Fett. *Danielfett/Proto6749*. Jan. 9, 2020. URL: https://github.com/danielfett/proto6749 (visited on 04/25/2021) (cit. on p. 33).

[16] D. Fett. *How (Not) to Use OAuth - Danielfett.De*. URL: https://danielfett.de/talks/2019-09-24-how-not-to-use-oauth/ (visited on 04/28/2021) (cit. on p. 24).

[17] D. Fett, P. Hosseyni, R. Kusters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2019, pp. 453–471. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00067. URL: https://ieeexplore.ieee.org/document/8835218/ (visited on 09/21/2020) (cit. on p. 3).

[18] D. Fett, P. Hosseyni, R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). May 2019, pp. 453–471. DOI: 10.1109/SP.2019.00067 (cit. on pp. 5, 28).

[19] D. Fett, R. Küsters, G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 1204–1215. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978385. URL: https://doi.org/10.1145/2976749.2978385 (visited on 05/03/2021) (cit. on pp. 26, 28, 31).

[20] D. Fett, R. Küsters, G. Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System". In: *2014 IEEE Symposium on Security and Privacy*. 2014 IEEE Symposium on Security and Privacy. May 2014, pp. 673–688. DOI: 10.1109/SP.2014.49 (cit. on p. 31).

[21] *Final: OpenID Connect Core 1.0 Incorporating Errata Set 1*. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 04/26/2021) (cit. on pp. 16, 26, 31).

[22] D. Hardt, M. Jones. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. URL: https://tools.ietf.org/html/rfc6750 (visited on 04/24/2021) (cit. on p. 22).

[23] D. Hardt <dick.hardt@gmail.com>. *The OAuth 2.0 Authorization Framework*. URL: https://tools.ietf.org/html/rfc6749 (visited on 04/26/2021) (cit. on pp. 19, 24, 26, 29).

[24] J. Hildebrand, M. Jones. *JSON Web Encryption (JWE)*. URL: https://tools.ietf.org/html/rfc7516 (visited on 04/27/2021) (cit. on p. 31).

[25] "Home Ethereum.Org". In: *Ethereum* (Mar. 2021). URL: https://ethereum.org/en (cit. on p. 15).

[26] S. P. Hosseyni Damabi. "Security Analysis of the OpenID Financial-Grade API". In: (2018). In collab. with U. Stuttgart. DOI: 10.18419/OPUS-10080. URL: http://elib.uni-stuttgart.de/handle/11682/10097 (visited on 09/18/2020) (cit. on pp. 3, 5).

[27] C. Mainka. "Developing a Security Analysis Tool for OpenID-Based Single Sign-on Systems". In: *Bachelor thesis, Ruhr-Universität Bochum* (2013) (cit. on p. 31).

[28] *Mix-Up, Revisited - Danielfett.De*. URL: https://danielfett.de/2020/05/04/mix-up-revisited/ (visited on 05/03/2021) (cit. on p. 28).

[29] *Openid / Fapi / FAPI_2_0_Advanced_Profile.Md — Bitbucket*. URL: https://bitbucket.org/openid/fapi/src/94d45dd8fe34471d46a59f822974a472332c288d/FAPI_2_0_Advanced_Profile.md?at=master (visited on 04/27/2021) (cit. on p. 27).

[30] *Openid / Fapi / FAPI_2_0_Attacker_Model.Md — Bitbucket*. URL: https://bitbucket.org/openid/fapi/src/master/FAPI_2_0_Attacker_Model.md (visited on 05/03/2021) (cit. on p. 24).

[31] *Openid / Fapi / FAPI_2_0_Baseline_Profile.Md — Bitbucket*. URL: https://bitbucket.org/openid/fapi/src/977d75a7dea78880d84b675b0a182b29f55a9cef/FAPI_2_0_Baseline_Profile.md?at=master (visited on 04/26/2021) (cit. on p. 24).

[32] *OpenID Connect | OpenID*. Aug. 1, 2011. URL: https://openid.net/connect/ (visited on 04/26/2021) (cit. on p. 16).

[33] "Payment Services (PSD 2) - Directive (EU) 2015/2366". In: *European Commission - European Commission* (Dec. 2016). URL: https://web.archive.org/web/20210328165231/https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en (cit. on pp. 3, 5, 15).

[34] *PEP 8 – Style Guide for Python Code*. Python.org. URL: https://www.python.org/dev/peps/pep-0008/ (visited on 05/03/2021) (cit. on p. 40).

[35] J. Richer <jricher@mitre.org>. *OAuth 2.0 Token Introspection*. URL: https://tools.ietf.org/html/rfc7662 (visited on 04/26/2021) (cit. on pp. 22, 26).

[36] N. Sakimura, J. Bradley, M. Jones. *The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)*. Internet-draft draft-ietf-oauth-jwsreq-30. Internet Engineering Task Force / Internet Engineering Task Force. 35 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-jwsreq-30 (cit. on p. 27).

[37] N. Sakimura, F. Skokan, T. Lodderstedt, B. Campbell, D. Tonge. *OAuth 2.0 Pushed Authorization Requests*. URL: https://tools.ietf.org/html/draft-ietf-oauth-par-05 (visited on 05/02/2021) (cit. on pp. 29, 44).

[38] K. Selden. *Re: [OAUTH-WG] Refresh Tokens*. URL: https://mailarchive.ietf.org/arch/msg/oauth/vSmJ0zjQzZFjeFbRz_qpvjfpAeU/ (visited on 05/03/2021) (cit. on p. 19).

[39] K. zu Selhausen, D. Fett. *OAuth 2.0 Authorization Server Issuer Identifier in Authorization Response*. URL: https://tools.ietf.org/html/draft-ietf-oauth-iss-auth-resp-00 (visited on 05/03/2021) (cit. on p. 28).

[40] *Strict Mode - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode (visited on 05/01/2021) (cit. on p. 35).

[41] *Terminology Reference*. OAuth 2.0 Simplified. URL: https://www.oauth.com/oauth2-servers/definitions/ (visited on 04/24/2021) (cit. on p. 20).

[42] *The Problem with OAuth for Authentication*. URL: http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html (visited on 04/24/2021) (cit. on p. 22).

[43] F. Valsorda. *FiloSottile/Mkcert*. May 3, 2021. URL: https://github.com/FiloSottile/mkcert (visited on 05/04/2021) (cit. on p. 34).

[44] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, Y. Gurevich. “Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 399–414. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_rui (cit. on p. 22).

[45] *X.509 Certificates*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/cert3.html (visited on 05/03/2021) (cit. on p. 28).

[46] F. Yang, S. Manoharan. “A Security Analysis of the OAuth Protocol”. In: *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM). Aug. 2013, pp. 271–276. DOI: 10.1109/PACRIM.2013.6625487 (cit. on p. 31).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature