Audience Injection Attacks: A New Class of Attacks on Web-Based Authorization and Authentication Standards

Pedram Hosseyni Ralf Küsters Tim Würtele

{pedram.hosseyni, ralf.kuesters, tim.wuertele}@sec.uni-stuttgart.de

Institute of Information Security – University of Stuttgart, Germany

We introduce *audience injection attacks*, a novel class of vulnerabilities that impact widely used Web-based authentication and authorization protocols, including OAuth 2.0, OpenID Connect, FAPI, CIBA, the Device Authorization Grant, and various well-established extensions, such as Pushed Authorization Requests, Token Revocation, Token Introspection, and their numerous combinations. These protocols underpin services for billions of users across diverse ecosystems worldwide, spanning low-risk applications like social logins to high-risk domains such as open banking, insurance, and healthcare.

Audience injection attacks exploit a critical weakness in a core security mechanism of these protocols – the handling of so-called audiences in signature-based client authentication mechanisms. This vulnerability allows attackers to compromise fundamental security objectives whenever these mechanisms are utilized across two or more server endpoints. They enable the attacker to impersonate users and gain unauthorized access to their resources, even in high-security protocol families specifically designed for sensitive applications.

We responsibly disclosed these vulnerabilities to the relevant standardization bodies, which recognized their severity. In collaboration with these organizations, we developed fixes and supported a coordinated response, leading to an ongoing effort to update a dozen of standards, numerous major implementations, and far-reaching ecosystems.

1 Introduction

Over the last decade, protocols for authorization and authentication on the Web have gained significant importance. A prominent example of the initial usage of such protocols are so-called "social logins", which allow users to log in to third-party Web pages using existing accounts at commonly used Websites, like Google or Facebook. In addition to login, these protocols enable delegated authorization, allowing a third-party service, for example, to have write access to one's Dropbox or Facebook account. This kind of delegated authorization and authentication, often called *Single Sign-On (SSO)*, is ubiquitous on the Web, used by billions of users, and is supported, among others, by major companies such as Google, Meta, Microsoft, and Apple. With the rise of IoT, new use cases emerged that require connecting devices, e.g., smart TVs, to existing SSO accounts. Authentication and authorization in this context is well supported by big SSO providers too [30, 54], with standardized protocols that account for limitations of IoT devices, such as input constraints and limited display capabilities.

In recent years, SSO protocols have increasingly been adopted for high-risk applications. For instance, many financial institutions enable end-users to authorize third-party Fintech companies to access their account data, a practice commonly referred to as *open banking*. In some jurisdictions, open banking is even mandated by legal regulations, such as in the European Union, the UK, Australia, Brazil, the UAE, and Saudi Arabia [2, 7, 16, 56, 71], and even without legal requirements, many banks worldwide support open banking, e.g., in the US and Canada. Similarly, an increasing number of countries, including Australia and Brazil, are enacting regulations to enable data sharing in the insurance sector, commonly referred to as *open insurance*. Likewise, there are use cases in the healthcare sector, such as access to

electronic patient records in the US, UK, Norway, and Germany [28, 34, 55, 77]. Overall, these high-risk use cases alone account for hundreds of millions of users.

Furthermore, such authorization and authentication protocols are widely used in cloud and distributed computing environments, not only to authenticate end-users towards various services without the need to log in at each service individually, but also for securing service-to-service communication [1, 31].

The OAuth-derived Modular Authentication and Authorization Protocol Families. The most widely used protocols for authorization and authentication belong to the *OAuth 2.0* and *OpenID Connect (OIDC)* protocol families or are derived from them. In that follows, we briefly sketch the collection of all these protocols, their extensions, and derived protocol families, and in the rest of this paper refer to this collection as the *OAuth-derived Modular Authentication and Authorization (OMAA)* families of protocols.

OAuth 2.0. The *OAuth 2.0 Authorization Framework* [33], standardized in 2012, defines central concepts and multiple protocol variants, often called *flows*, for delegated authorization. In a nutshell, these flows allow an end-user (the *resource owner*) who owns resources, such as a bank account, to provide a so-called *client*, e.g., a Fintech account aggregation service, access to their resources. The access to the user's resources is managed by a so-called *authorization server (AS)*, e.g., the user's bank. The client requests access to a resource at the AS, which provides an *access token* to the client based on permission from the resource owner. The resources themselves are stored on *resource servers* (potentially different from the AS). If resources are owned by the client itself, e.g., in the service-to-service context mentioned before, the so-called *client credentials flow* can be used, which enables the client to directly retrieve access tokens from the AS.

OpenID Connect. The OpenID Connect standard [68] (OIDC), originally specified in 2014, defines an authentication layer on top of OAuth 2.0. Roughly speaking, OIDC establishes user authentication by allowing the client to obtain identifying information about the end-user from the AS.[1]

Security and Functional Extensions. Over the years, several security-related extensions have been developed for these families of protocols, for example, *Pushed Authorization Requests (PAR)* [50], *Token Revocation* [51], or *Proof Key for Code Exchange* [64], which provide better integrity and authenticity of some messages or keep the protocols secure even if certain values leak. Furthermore, there are functional extensions, like *Dynamic Client Registration* [63, 67] and automatic discovery of AS configuration [45, 69], which allow clients to retrieve AS configuration such as endpoints and public keys and to automatically register at an AS, given just a single identifier of the AS. These are just a few examples. There is, in fact, a wide range of such functional and security extensions (e.g., [11–13, 18, 22, 43, 44, 49, 52, 61, 62, 70, 81]), which can be combined in various ways, amounting to hundreds of standards-compliant protocols; hence, the attribute "modular" in OMAA. We emphasize that combining extensions is not just a theoretical possibility: many combinations are in actual use across thousands of deployments [40, 60], and large SaaS providers for authentication and authorization, such as Okta and Authlete, serving hundreds of customers with millions of users, provide modular approaches in which customers can switch extensions on and off as needed [4, 6].

The OpenID FAPI Protocol Families. A particularly notable combination of OIDC with several security and functional extensions are the high-security OpenID FAPI 1.0 [65, 66] and FAPI 2.0 [21, 26, 27] families of protocols, also called profiles.[2] These profiles were developed for high-risk use-cases like financial and healthcare applications and, among others, were created with the assumption that particularly powerful and highly incentivized attackers can directly compromise certain secrets in protocol runs of honest parties.

Decoupled Flows. The previously mentioned protocol families (except for the aforementioned client credentials grant) were designed for use cases where the end user authorizes a client via an AS where both client and AS are accessed by the user via their browser. However, new use-cases called for cross-device capabilities, e.g., to allow users to connect input-constrained devices like smart TVs to their accounts, e.g., at YouTube. For such use-cases, so-called *decoupled flows* were developed, which have significantly different protocol structures: the device which receives authorization (*consumption device*, formerly the client) is different from the device used in the user–AS interaction (*authentication device*). Examples of such decoupled flows are the *OAuth 2.0 Device Authorization Grant* [19], *OpenID Connect Client-Initiated Backchannel Authentication Flow (CIBA)* [20], as well as variants of those such as the FAPI-CIBA [75] profile of CIBA and combinations with various extensions.

Evaluation of Security in the Scientific Literature. The security of various of the OMAA families of protocols has been extensively studied. Formal analyses go back to drafts of OAuth 2.0 in [8, 9, 58, 78], which conduct tool-based formal analyses using Alloy, ProVerif, and ASLan++. Chari et al. [17] analyze a draft of OAuth 2.0 in the UC model.

[1] In the context of OIDC, the AS is called *identity provider*, and the client is called *relying party*. For simplicity, in this paper we stick to one terminology and use the OAuth 2.0 terminology.

[2] A profile is a combination of base protocols and a set of extensions.

In [24, 25], the OAuth 2.0 and OIDC core flows are formally analyzed in a detailed model of the Web infrastructure. The security of the various flows defined by both versions of FAPI was analyzed using the same analysis framework in [23, 35, 36].

In addition to these analyses on the protocol level, implementations and live deployments have also been analyzed thoroughly, e.g., in [10, 53, 59, 72, 73, 79, 80]. These works test implementations and deployments against specific kinds of vulnerabilities or more generally, for example, against security requirements of the standards or the security best practices specified by the IETF OAuth Working Group [48]. A detailed survey of such works is provided in [40].

Client Authentication. A core security mechanism in all of the OMAA families of protocols is *client authentication*, in which the client authenticates to the AS when sending requests to certain endpoints, e.g., when requesting access tokens. There are multiple standardized client authentication methods based on both symmetric secrets and public key cryptography. The symmetric methods transmit a shared secret or use it with a MAC scheme, whereas the public key methods employ mutual TLS or use signatures (see Section 2.1.2 for details). Public key-based methods are generally considered to be more secure, for example, the FAPI specifications mandate the use of asymmetric authentication methods.

A Novel Class of Attacks. Despite the aforementioned analyses and more than a decade of active use of authorization and authentication protocols, we found a fundamental flaw in the main signature-based client authentication methods. As client authentication is a core security mechanism, breaking it can obviously have severe consequences. In general, an attacker can break both *authorization*, i.e., get access to an honest user's resources, and *authentication*, i.e., log in at an honest client under an honest user's identity. The concrete effects of a successful attack depend on the specific protocol and setting at hand, e.g., attacks on flows that involve human end-users are naturally different from attacks on client credentials flows. For illustration, we describe an example in a typical Web setting from the victim's point of view: Consider an end-user registered at the AS *as* and trusting the (honest) client *c*. In particular, the user already authorized *c* to access resources managed by *as*. The attacker now sends an email to the user, copying the client's corporate design and requesting a re-authorization under some pretext.[3] The attacker's email contains a link to *as*, and when following the link, the user is asked whether they want to authorize *c*. We highlight that the user opens a legitimate Website of *as*, and fully expects to be asked to authorize the honest and trusted *c*. I.e., at this point, the user cannot detect any fraudulent behavior. Once the user authorizes this request, the attacker gets access to the user's resources. We again emphasize that this is just one instance of an audience injection attack (see Sections 3 and 4).

This flaw constitutes a novel class of attacks that we call *audience injection attacks*. These attacks affect many protocols in the OMAA families, including the high-risk FAPI 1.0 [65, 66] and FAPI 2.0 [21, 26, 27] profiles. While there are some nuances that we discuss in Sections 2-4, if clients and ASs support (but not necessarily use) one of the vulnerable client authentication methods, then decoupled and client credentials flows are immediately vulnerable to audience injection attacks. For other flows, an audience injection vulnerability can for example be introduced by using one of the following extensions: PAR [50], the automatic client registration mechanism defined in OpenID Federation [46], Token Revocation [51], or Token Introspection [61]. Note that if a client supports a single vulnerable flow (even if not used), the attacker is often able to also impersonate the client in otherwise unaffected flows.

Interestingly, as far as we know, none of the previous works even considered the vulnerable authentication methods, except for the FAPI 2.0 analyses in [35, 36]. The analyzed FAPI 2.0 protocol drafts in these two analyzes did not contain an audience injection vulnerability due to a clause in the specifications that was meant to ensure better interoperability. However, changes made to the specifications after these analyses re-introduced the issue.

Responsible Disclosure. Audience injection attacks affect a wide range of protocols and real-world use cases, including multiple high-risk environments, and a similarly wide range of standards for protocols, extensions, and profiles. In particular, we identified the following standards that need to be revised to prevent audience injection attacks: Client authentication methods as defined in RFCs 7521, 7522, 7523, and OIDC [14, 15, 41, 68], PAR as defined in RFC 9126 [50], the automatic client registration mechanism defined in OpenID Federation [46], CIBA [20], FAPI 1.0 [65, 66], FAPI 2.0 [27], and a standard that collects security best practices for OAuth 2.0 [48]. These standards are managed by the *OAuth Working Group* of the IETF as well as several working groups at the OpenID Foundation (OIDF), which are also in contact with major deployment stakeholders. In September of 2024, we approached the OIDF and the authors of the IETF standards, who acknowledged the attacks. Furthermore, the OIDF asked us to accompany a responsible disclosure process with the goal of fixing existing implementations before the attack becomes public. At the time of this writing, there is a coordinated update of the affected specifications, which we are accompanying, including discussions of potential fixes. Such a coordination is necessary due to the public nature of the specifications and to

[3] Legitimate re-authorization can be required for technical reasons, due to provider-specific policies, or even be mandated by regulation. A prominent example is the European PSD2, which requires such a re-authorization in the financial context every 180 days: https://www.eba.europa.eu/publications-and-media/press-releases/eba-publishes-final-report-amendment-its-technical-standards

ensure alignment of the mitigations. Likewise, the publication of this paper is agreed by and coordinated with the responsible standardization bodies. We refer to Section 6 for a detailed discussion of the responsible disclosure process.

Contributions. We summarize our contributions as follows:

- We identify and characterize *audience injection attacks*, a novel class of severe vulnerabilities targeting a broad family of authentication and authorization standards relied upon by billions of users worldwide. These attacks undermine the core security objectives of these protocols: safeguarding access to users' resources and identity information, even in high-risk environments.
- Audience injection attacks affect more than a dozen standards. We worked with the standardization bodies in charge to develop fixes and update the specifications.
- We also accompanied a several months long responsible disclosure process between the standardization bodies and major providers and deployments of SSO services, including high-risk ecosystems in finance, insurance, and healthcare.

Structure of This Paper. Section 2 gives an overview of the OMAA protocol families, including a detailed description of the aforementioned signature-based client authentication mechanisms. In Section 3, we show one instance of an audience injection attack in detail and continue with several other instances of the attack class in Section 4. We discuss possible fixes in Section 5 and report on the impact and our responsible disclosure in Section 6, before concluding in Section 7.

2 Overview of the Families of Protocols

As audience injection attacks impact a wide range of OMAA protocol families and their variants, it is infeasible to detail all such protocols in this paper. Instead, we focus on two illustrative instances: one using the original OIDC flow and the other utilizing the decoupled CIBA flow. Additionally, we briefly outline the basic FAPI 2.0 flow as an example of a protocol specifically designed for high-risk applications. We reiterate that these examples represent only a small subset of the numerous combinations of base protocols and extensions affected by audience injection attacks.

2.1 A First Instance of an OIDC Flow

We now present the original OIDC *authorization code flow* with the *OpenID Connect Discovery* [69] and *Pushed Authorization Request (PAR)* [50] extensions, a very common combination in practice. With this protocol instance it is also rather easy to illustrate the concept of audience injection.

Overview. At a high level, the protocol proceeds as follows: An end user visits the client's, say *c*'s, website and wishes to authenticate to *c* via the AS *as*. If no existing relationship exists between *c* and *as*, *c* uses the Discovery extension to retrieve *as*'s OIDC configuration from the *configuration endpoint*, derived from *as*'s identifier provided by the user. This configuration includes endpoint URLs, public keys, supported extensions and so on. Next, *c* registers with *as*, receiving a client ID and registering *c*'s public key for later authentication of *c* to *as* (see Section 2.1.2 for details on this and other client authentication methods).

Once registered, *c* sends a PAR request to *as*'s *PAR endpoint* with its client ID, a data structure signed with the registered key for authentication, and details of the requested access. In response, *as* issues a unique session identifier, which *c* includes when redirecting the user to *as*'s *authorization endpoint*. There, the user authenticates and consents to *c*'s request. Upon consent, *as* redirects the user to *c*'s *redirect endpoint* with a nonce, which *c* exchanges for tokens at *as*'s *token endpoint*, while in this last step *c* also authenticates to *as*.

2.1.1 Protocol Description

Let us give a more detailed explanation that follows Figure 1 (all message exchanges use HTTPS). We note though that our presentation still lacks some specifics not relevant to this paper's discussion, such as *refresh tokens* that are used to renew expired access tokens (see the specifications in [33, 50, 68, 69] for all available flows and full details).

In Step 1 the user (and resource owner) indicates to the client *c* that they want to authenticate to *c* and authorize *c* using the authorization server *as* identified by https://as.com, its so-called *issuer identifier* [81]. Since we assume that *c* does not yet have a relationship with *as* and so far only knows *as*'s identifier, *c* now performs Discovery (in Steps 2 and 3), i.e., *c* requests *as*'s configuration document from *as*'s configuration endpoint. The URL of this endpoint is

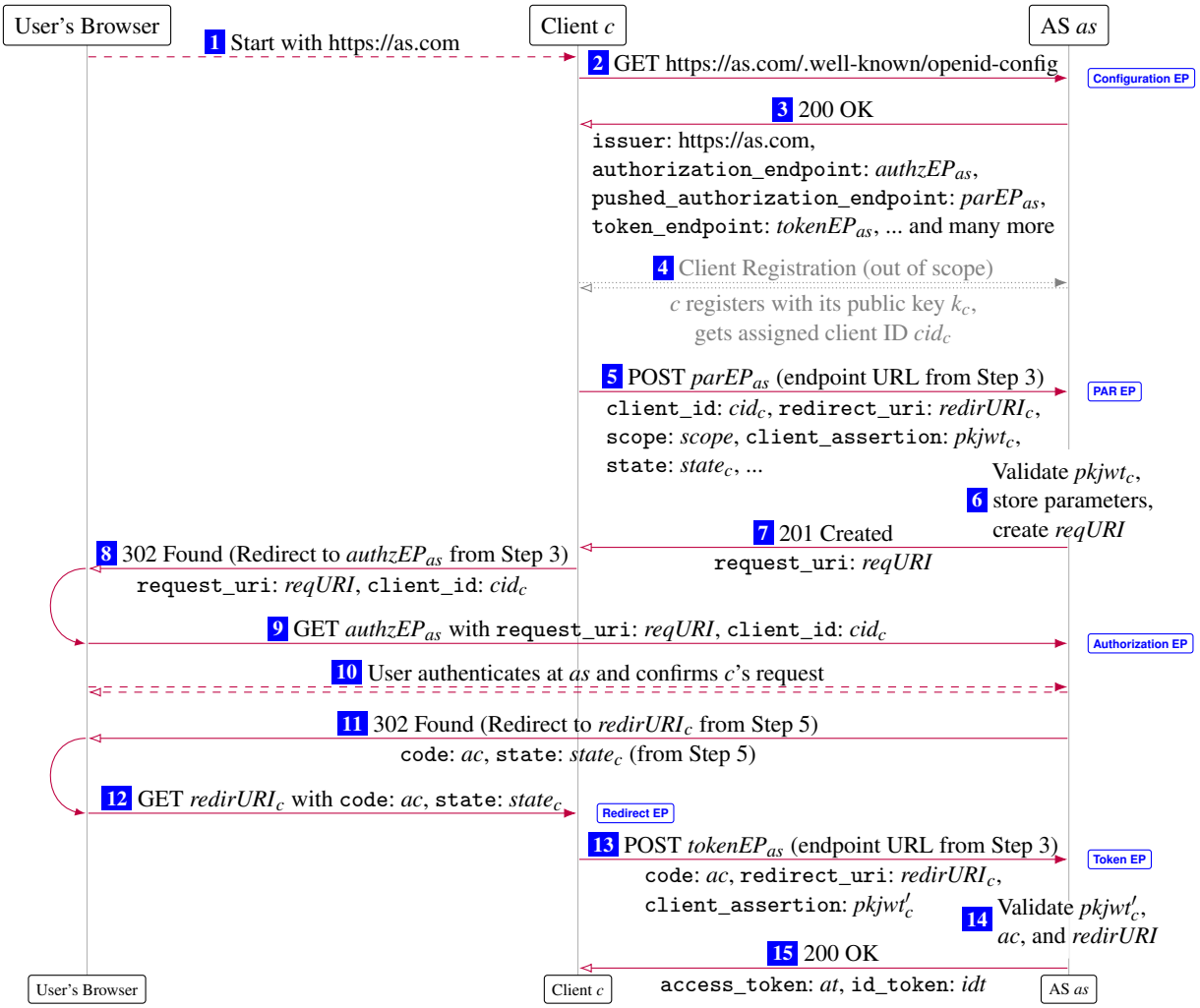

Figure 1: The OIDC authorization code flow with PAR and OID Discovery. The browser subsumes the end-user's behavior. All messages are exchanged via HTTPS.

https://as.com with a so-called well-known path [32] appended to it (see Step 2). This configuration document (Step 3) contains a list of endpoints for the protocols and extensions supported by *as*, its issuer identifier, public keys to verify signatures, and so on. After verifying that the issuer identifier value in the configuration document matches what *c* expects, it stores this configuration.

Once *c* obtained *as*'s configuration, it needs to register with *as* (Step 4). The registration process itself is out of scope of the OIDC specification, but typically, registration is either done out-of-band, e.g., by administrators, or by using one of the extension protocols for dynamic client registration [63, 67]. For the purposes of this paper, we focus on only two aspects of the registration: (1) *c* registers its public key k_c to later authenticate to *as*, and (2) *c* is issued a client ID cid_c by *as*. Since this data is stored by both parties, discovery and registration are necessary only once or in some regular interval.

With *c* registered with and in possession of *as*'s configuration, *c* can start the flow by sending a PAR request (Step 5) to *as* with: (1) its client ID cid_c; (2) a client-chosen *redirect URI* that will later be used by *as* to send the user back to *c*;[4] (3) a *scope* value indicating the type of access requested by *c*, e.g., identity information for logging in the user, write access to the user's timeline, or read access to photos; (4) a so-called *client assertion* that authenticates *c* by means of a signature (see Section 2.1.2); (5) a *state* value that *as* will later include in its response such that *c* can identify which session the response belongs to; (6) additional parameters that we omit for brevity.

[4]When using plain OIDC, *c* needs to explicitly register this redirect URI with *as* beforehand, but the PAR extension lifts this requirement due to the required client authentication at the PAR endpoint.

Upon receiving the PAR request, *as* validates *c*'s authentication, i.e., the client assertion (see Section 2.1.2), checks the scope and some other parameters for validity, stores the request and generates a random *request URI reqURI* that identifies *c*'s request and is returned to *c* in response to the PAR request (Step 7). The client now redirects the user's browser to *as*'s authorization endpoint, passing along the *reqURI* and *c*'s client ID cid_c (Step 8).

The browser follows this redirect to *as* (Step 9), where *as* uses the *reqURI* and cid_c to identify *c*'s original request and asks the user for their consent to *c*'s request after the user has authenticated to *as*, e.g., via username and password (Step 10). For this, *as* typically shows *c*'s name and logo and the list of permissions that *c* has requested.

Once the user gives their consent, their browser is redirected back to *c* (Step 11), specifically, to $redirURI_c$ provided by *c* in Step 5. Included in that redirect is a fresh nonce created by *as*, the *authorization code ac*, as well as the state value *c* sent in Step 5. Besides identifying the session for *c*, this state value serves as a security measure against CSRF attacks on *c*'s redirect endpoint: once *c* receives the redirected request from the browser in Step 12, it verifies that the received state value is associated with a session with that same browser, e.g., by comparing session cookies.

Using *ac*, *c* can now request tokens from *as* (Step 13) by sending *ac*, the $redirURI_c$, and a fresh client assertion to *as*'s token endpoint, where *as* validates the client assertion (see Section 2.1.2), *ac*, and that the $redirURI_c$ matches the one from Step 5. If these checks succeed, *as* responds with the requested access and ID tokens – access tokens allow *c* to access the user's resources stored at resource servers, while ID tokens contain details about the user itself, e.g., name and email address, for *c* to authenticate the user. ID tokens are signed by *as* and besides user details, they contain *as*'s issuer identifier and *c*'s client ID cid_c; *c* validates these values and the signature before accepting an ID token.

2.1.2 Client Authentication in Detail

As mentioned in Section 1, the protocols in the OMAA families offer multiple client authentication methods (see [13, 15, 33, 41, 68]) to be used in Steps 5 and 13. We briefly discuss the available methods, but focus on the methods relevant to our novel class of attacks. The symmetric *client secret*-based methods in which *as* issues a shared secret to *c* during registration that can be used either similar to a password to authenticate *c* to *as* or as a MAC key. The parties may also use asymmetric methods, which fall in one of two categories: Mutual TLS-based methods with self-signed or CA-signed certificates, and the methods relevant to this paper which are based on signatures. These are the *Private Key JWT* [68], the *JWT Bearer Token Authentication* [41], and the *SAML Bearer Assertion* (for OAuth 2.0 client authentication) [14] methods, all of which work in a similar way, at least as far as this paper's discussions are concerned. For simplicity, we present the protocols w.r.t. Private Key JWT (pkJWT), but emphasize that our results work with all these methods.

As mentioned above, *c* registers its public key k_c of the key pair $(k_c, \hat{k}_c)$ with *as*.[5] Whenever *c* needs to authenticate to *as*, it includes a pkJWT, signed with $\hat{k}_c$ in a `client_assertion` parameter. This pkJWT contains several so-called claims: (1) the `iss` claim contains *c*'s client ID and identifies *c* as the pkJWT's issuer. (2) the `sub` claim also contains the client ID and identifies the client as the subject of the pkJWT. (3) the `aud` claim identifies the intended so-called *audience* of the pkJWT, i.e., the authorization server. While the specifications do not strictly mandate the exact value of the `aud` claim, the value most often recommended to use is the authorization server's token endpoint URL (see Section 9 of [68], Section 5.1 of [15], Section 3 of [41], and Section A.5 of [66]), and where the token endpoint URL is not the recommended value, it is at least explicitly allowed (see Section 2 of [50], Section 7.1 of [20], and Section 5.3.3 of [27] prior to the fix implemented with [74] due to our findings). The `aud` claim may also contain multiple values as an array. (4) The `jti` claim contains a nonce, and finally, (5) the `exp` claim sets an expiration time for the pkJWT. Thus, a pkJWT $pkjwt_c$ of *c* has the form $\text{sign}(\langle \texttt{iss}: cid_c, \texttt{sub}: cid_c, \texttt{aud}: tokenEP_{as}, \texttt{jti}: n, \ldots \rangle, \hat{k}_c)$

When *as* receives such a pkJWT, it validates the signature using k_c as registered by *c*, verifies that the `iss` and `sub` claims contain *c*'s client ID, that the pkJWT is not expired, and that *as* can identify itself with at least one of the values in the `aud` claim. Unsurprisingly, given that the token endpoint URL is a recommended (or at least explicitly allowed) value for the `aud` claim, the specifications require *as* to consider its token endpoint URL an identifying value. Finally, the `jti` nonce is used to prevent replays.

2.2 Decoupled Flows

As mentioned in the introduction, decoupled flows are used when users want to authorize their input/display constraint devices (*consumption devices*), e.g., Smart TVs, to access the users' resources at an authorization/resource server (e.g., Youtube), where for this they use an *authentication device* (e.g., a smartphone) to authenticate to the AS. Among the various protocols and extensions in the OMAA families, we here present CIBA as one instance of these decoupled flows;

[5] *c* may use $(k_c, \hat{k}_c)$ across different ASs. Such reuse is common in practice and sometimes even codified, e.g., in the OpenID Federation standard [46].

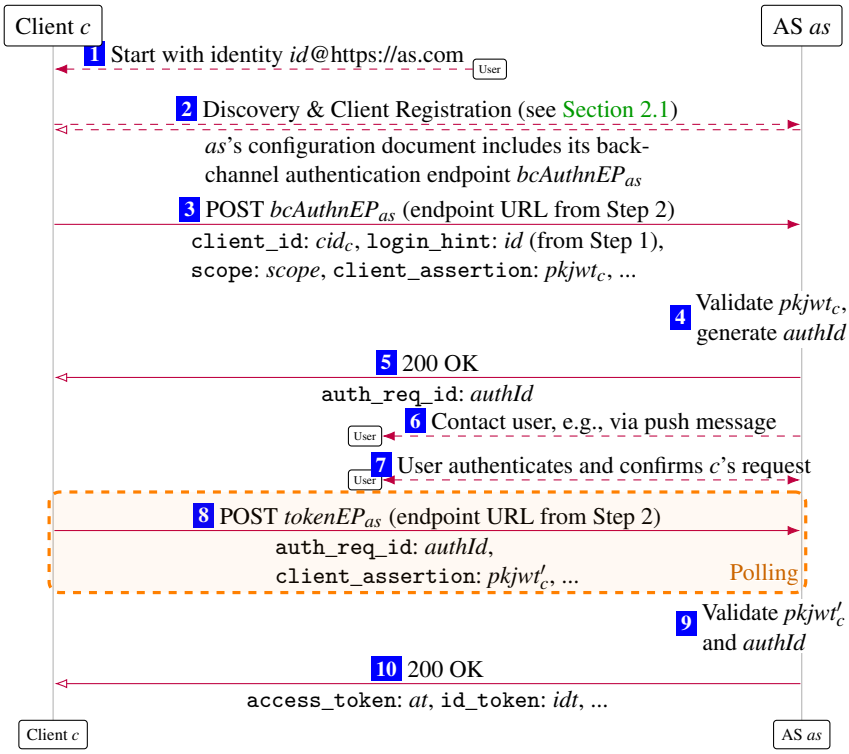

Figure 2: The CIBA core flow in its poll mode with the Discovery extension. The client subsumes the consumption device. All messages are exchanged via HTTPS.

specifically CIBA in its *poll mode* (see below for alternatives), and with the *Discovery extension*, depicted in Figure 2. Note that this setting is quite different to the one in Section 2.1 (and other protocols instances in the OMAA protocol families) since clients for decoupled flows are not expected to provide a browser and full input/display capabilities.

In Step 1, the user informs the consumption device that they want to authorize it with an identity *id* at some AS *as*, e.g., by providing an email address or phone number (the details are out of scope of the specifications). If the client does not yet have a relationship with *as*, it uses the Discovery extension to learn *as*'s configuration, including endpoints, and registers with *as* (Step 2, see Section 2.1); again this information is stored and can be used for multiple protocol runs.

To commence the actual authentication flow, *c* sends a request to *as*'s backchannel authentication endpoint with its client ID cid_c, the *id* provided by the user in Step 1, a scope value indicating what kind of access *c* requests, and a client assertion to authenticate *c* (see Section 2.1.2).

This request, in particular the client assertion, is validated by *as* in Step 4, who generates a unique identifier *authId* for the client's request. This *authId* is then returned to *c* (Step 5).

At the same time, *as* uses the *id* to identify and contact the user, e.g., with a push message to their smartphone (Step 6), informing the user about *c*'s request and asking for the user's consent (and, if deemed necessary, user authentication) in Step 7. The details of Steps 6 and 7 are out of scope of the protocol specifications, but typically include the *as* presenting the user with the client's name, logo, and details about what kind of access the client requests.

After *c* receives the *authId* in Step 5, it begins polling *as*'s token endpoint with token requests that include the *authId*, a fresh client assertion to authenticate *c*, and additional parameters (Step 8). Until the user confirms *c*'s request, *as* replies with "authorization pending". Once the user approves *c*'s request, *as* responds with the requested tokens (Step 10).

Alternative Modes. In addition to the poll mode described above, CIBA offers the ping and push modes. Each results in slightly different protocol variants in terms of message exchanges and their contents. The key difference lies in how the client requests and receives tokens. In ping mode, the AS notifies the client once tokens are available. In push mode, the AS directly sends the tokens to the client once they are available.

2.3 The FAPI 2.0 High-security Profiles

The FAPI 2.0 profiles are protocol variants of OIDC designed for and widely used in high-risk applications. In a nutshell, the FAPI 2.0 profiles are instances of the OMAA protocol families consisting of OIDC, a collection of security-enhancing extensions, and fixed selections for some options offered by OIDC and the used extensions. We briefly sketch the most important of the extensions used in the *FAPI 2.0 Security Profile* [27]: (1) PAR [50] (already mentioned in Section 2.1), requiring the client to authenticate at the AS when sending its initial request. (2) Proof Key for Code Exchange (PKCE) [64], which links the client's token request (Step 13 in Figure 1) to its first request (Step 5) by including the hash of a nonce in the first request and the nonce itself in the token request. (3) AS Issuer Identification [81], which prevents AS mix-up attacks (see [24]). (4) Access token sender constraining using *OAuth 2.0 Demonstrating Proof of Possession (DPoP)* [22] or mutual TLS [13], in which the AS binds each access token to a public key provided by the client – the resource servers then require the client to include a proof of possession of the corresponding private key when using the access token, thus offering protection even if access tokens leak to the attacker. (5) OpenID Discovery [69] as described above.

3 A Simple Audience Injection Attack Instance

After a brief discussion of the core security goals shared by all members of the OMAA families of protocols, we illustrate the class of audience injection attacks by an instance of the attack on the OIDC authorization code flow with the PAR and Discovery extensions presented in Section 2.1, and show how it breaks the core security goals.

Security Goals. Except for FAPI 2.0 [21], the relevant standards do not explicitly define security goals. However, prior analyses of OMAA protocol families provide detailed definitions [23–25, 35, 36] that also align with FAPI 2.0, and which we paraphrase here. Unsurprisingly, the primary security goals of these protocols are *authentication* and *authorization*. **Authentication** ensures that no attacker can log in to an honest client using an honest user's identity, if the AS managing the identity is honest. Similarly, **authorization** prevents attackers from accessing an honest user's (or client's) resources stored on an honest resource server, if the AS and any clients authorized to access these resources are honest.

Overview. An audience injection attack comprises two largely independent phases. In the first phase, an honest client c is tricked into providing the attacker with a valid client assertion, signed by c, with an `aud` claim value acceptable to an honest AS *as*, e.g., *as*'s token endpoint. This enables the attacker to impersonate c to *as*, which in itself may be considered a successful attack, but as such does not break the core security goals. Hence, in the second phase, the attacker leverages this impersonation to, for instance, obtain access tokens for an honest user's (or c's) resources, thereby compromising the security goals. For most protocol variants, attackers can choose from multiple patterns for each phase and combine them, yielding a variety of attack flows.

In the remainder of this section, we detail one example of each attack phase and refer to Section 4 for additional variants.

3.1 Simple Instance of the First Attack Phase

In the first attack phase, the attacker's goal is to obtain a valid client assertion, signed by an honest client c, with an `aud` value that will be accepted by an honest AS *as*.

For the protocol instance presented in Section 2.1, one way for the attacker to achieve this goal is depicted in Figure 3.

We assume an honest client c is already registered with an honest AS *as*, got assigned client ID cid_c and uses c's key pair $(k_c, \hat{k}_c)$ for authentication. Furthermore, let that *as*'s token endpoint be $tokenEP_{as}$. If c is not yet registered with *as*, the attacker can pose as a user of c to trigger the registration process (Steps 1–3 of Figure 3).

The actual attack begins with the attacker posing as a user of c attempting to authenticate with an attacker-controlled AS as_{att} (Step 4). This prompts c to retrieve as_{att}'s configuration document (Step 5), which falsely lists $tokenEP_{as}$, i.e., the token endpoint of *as*, as its token endpoint; this is where *audience injection* starts since that value will later be used by c as the audience claim value of a client assertion. Note that c has no way to externally validate as_{att}'s configuration, i.e., c must accept it as-is. Consequently, c uses $tokenEP_{as}$ as the token endpoint in all subsequent interactions with as_{att}.

Next, c registers with as_{att} (Step 6), using the same key k_c that it already uses with *as* (see "Client Key Management" below). As usual, c gets assigned a client ID by as_{att} – here, the attacker chooses the same cid_c that c got assigned by *as* before (see "Client ID Assignment" below).

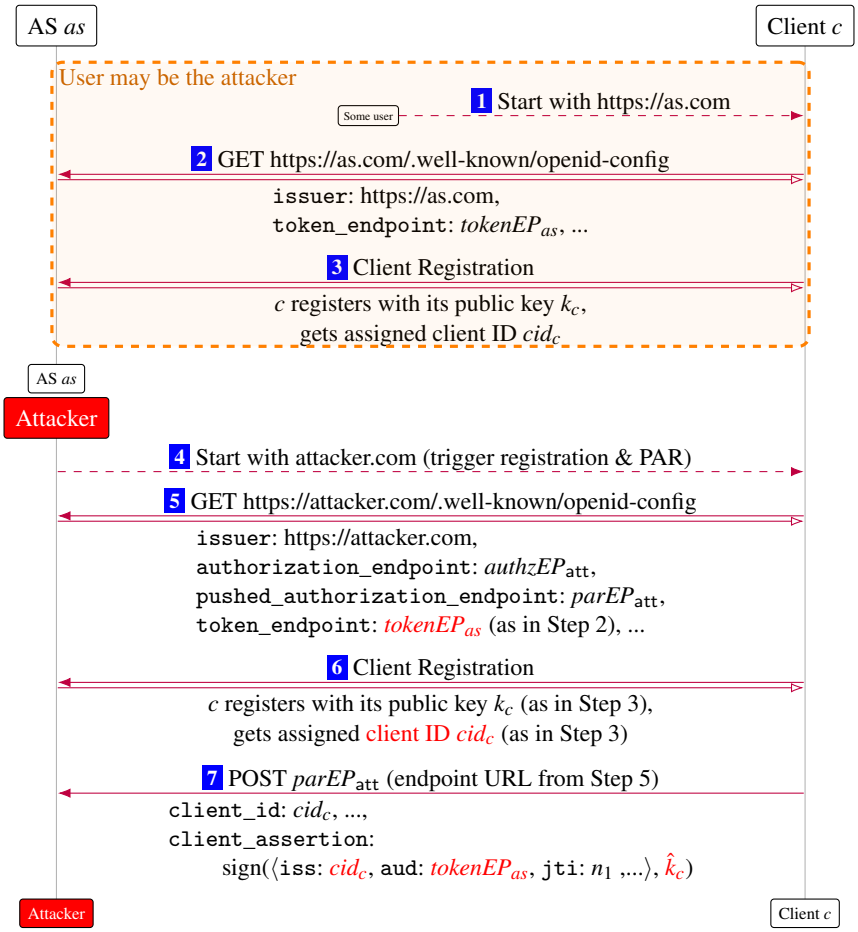

Figure 3: Example for the first attack phase

Following registration, c sends a PAR request to as_{att} (Step 7) with its assigned client ID, a client assertion, and so on. As described in Section 2.1.2, this client assertion is signed with c's private key $\hat{k}_c$ and contains `iss` and `sub` claims with the client ID, i.e., cid_c, as well as an `aud` claim. The value of that `aud` claim is token endpoint of the intended recipient (as_{att}), i.e., $tokenEP_{as}$ (due to Step 5).

Hence, the attacker obtains a client assertion signed by c, with as's token endpoint as its audience, and cid_c as the issuer and subject – thus, as will accept this client assertion to authenticate c. Repeating Steps 4 and 7 provides the attacker with multiple such client assertions, each with a fresh `jti` nonce.

Client Key Management. We emphasize that using the same key pair for the same operation (signing) within the same protocol is common practice and violates neither general cryptographic, nor any specific best practices. In fact, some of the OMAA family protocols even require this [46]; likewise, OMAA SaaS providers like Okta and Authlete limit the number of client key pairs that can be used simultaneously to low one-digit numbers [3, 5].

Client ID Assignment. Since client IDs are "AS-local", nothing in the standards prevents duplicate client IDs across different ASs. Furthermore, client IDs are not required to be high-entropy values either: since client IDs are passed "through" the user's browser, they are considered to be publicly known values anyway. In fact, some OMAA standards even mandate that a client always gets the same client ID from all ASs [46, 76].

3.2 Second Phase of the Attack

In the second attack phase, the attacker aims to use one or more client assertions obtained in the first phase to gain access to a user's (or client's) resources or to log in to a client under an honest user's identity. We illustrate the second attack phase for OIDC with the PAR extension in Figure 4, where the attacker compromises the authorization goal. Recall that after the first attack phase, the attacker to possesses client assertions from an honest client c that are valid at

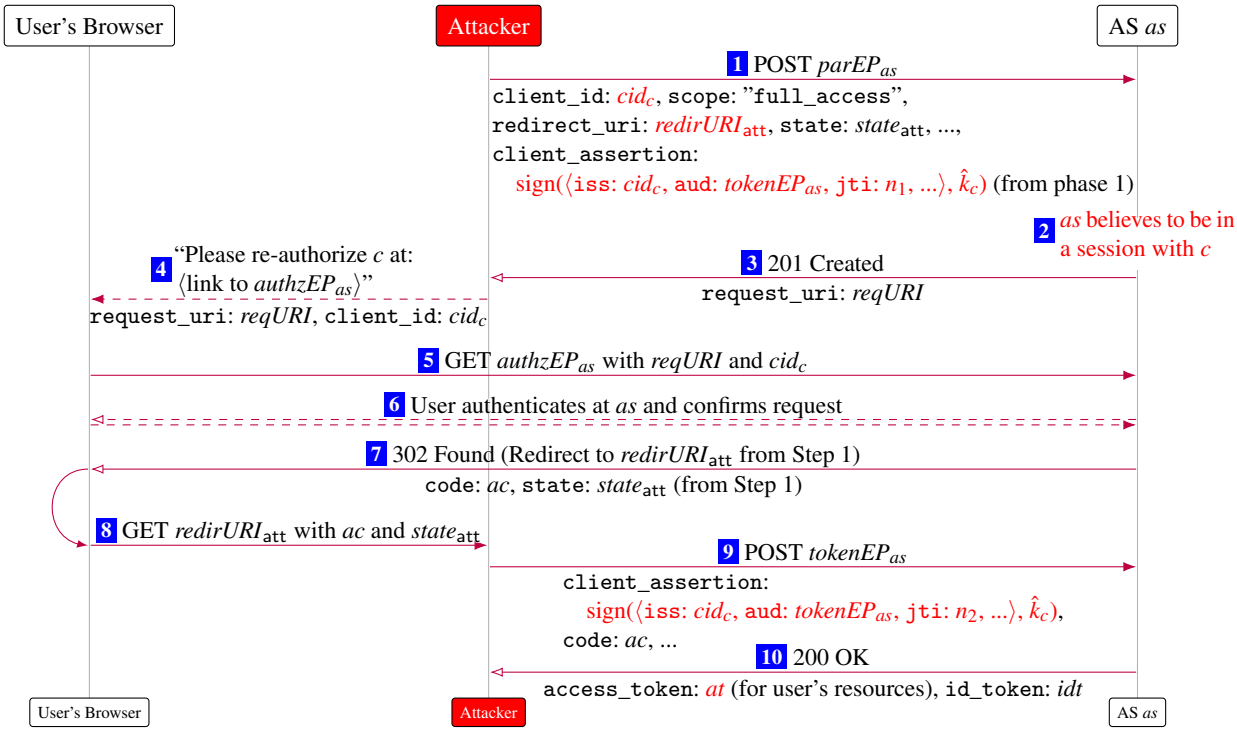

Figure 4: Example of the second attack phase. The attacker already obtained valid client assertions from a client c that is registered with public key k_c and client ID cid_c at as.

an honest AS as.

In the second phase, the attacker sends a PAR request to as (Step 1) with cid_c, an attacker-chosen scope, a redirect URI to an attacker-controlled server, and a client assertion obtained in the first phase. The assertion includes cid_c in the `iss` and `sub` claims and as's token endpoint in the `aud` claim, i.e., to as, this client assertion authenticates c, and this client assertion has never been used at as before. Hence, as responds with an identifier $reqURI$ (Step 3).

Next, the attacker persuades the user to follow a link to as's authorization endpoint, including cid_c and $reqURI$, e.g., via an email mimicking c's corporate design and requesting reauthorization (Step 4, see also Footnote 3). Note that this link appears legitimate, pointing to the correct AS as and using the correct client ID.

The user, rightfully trusting as and c, clicks the link, authenticates, and is asked to authorize c's (seemingly legitimate) request in Step 6 (recall: as is convinced to be in a protocol run with c). We emphasize that the user is only interacting with an honest party, as, and fully expects to authorize c at this point, i.e., the user has no way to detect this attack.

Following the user's consent, as redirects the user to the redirect URI $redirURI_{\mathsf{att}}$ from the PAR request, i.e., to the attacker, with an authorization code ac (Steps 7 and 8).

The attacker then uses ac in Step 9 to send a token request to as, including another client assertion from the first phase, i.e., with a different `jti` nonce. Recognizing the assertion as authenticating c and with ac belonging to a protocol run with c and the victim user, as issues the requested tokens, including an access token for the victim's resources, thus compromising the authorization goal.

4 Further Audience Injection Attack Instances

As explained, audience injection attacks consist of two phases, and for most members of the OMAA families of protocols, there are multiple attack patterns for both phases that may be combined freely. In the following, we present overviews of several of these patterns for both phases and for multiple protocol variants. Still, the set of patterns presented here is by no means exhaustive.

4.1 Further Instances of the First Attack Phase

Recall that the attacker's goal in the first attack phase is to obtain a client assertion from an honest client c that authenticates c towards an honest AS as.

In the example presented in Section 3.1, the attacker receives c's client assertion at the attacker's PAR endpoint. Since PAR is an (in most cases) optional extension, we discuss other ways for the attacker to receive such a client assertion below. Without PAR, Steps 5–7 in Figure 1 are omitted, the parameters of Step 5 are instead included in the redirect in Step 8 – except for the client assertion, i.e., without PAR, this first request from c to as "through" the user's browser is not authenticated.

In all cases, we assume an honest client c that is registered with both an honest AS as and an attacker-controlled AS as_{att}, using the same public key for both, and being assigned the same client ID cid_c by both. Furthermore, we assume that as_{att}'s configuration document falsely lists as's token endpoint $tokenEP_{as}$ as its own token endpoint.

We emphasize that while our examples consistently use the Discovery extension, any other method for the client to obtain AS configuration are just as vulnerable (e.g., manual configuration, OpenID Federation [46], or other extensions such as the AS Metadata extension [45]). Likewise, recall that while we use the pkJWT client authentication method for presentation, these attacks affect all signature-based methods (see Section 2.1.2).

Token Revocation. With the OAuth 2.0 Token Revocation [51] extension, a client can notify the AS that an access token is no longer needed, e.g., when the corresponding end user logs out of the client. To support this, the AS provides a *revocation endpoint* that expects an access token and a client assertion (see Section 2.1.2). In this attack, as_{att} includes such a revocation endpoint in its configuration document, pointing to itself. The attack idea is for the attacker to complete a protocol run with c and as_{att}, and then trigger token revocation to obtain a client assertion with $tokenEP_{as}$ as its audience.

The attacker poses as a user of a client c that wants to authenticate with as_{att}. Hence, c redirects its user (the attacker) to the chosen AS's authorization endpoint (as_{att}) with cid_c, redirect URI, etc., but no client assertion, since PAR is not used. The attacker now forwards this request to as's authorization endpoint,[6] where as verifies that the redirect URI matches what the client with ID cid_c has registered beforehand (see Footnote 4), which is the case, and asks the user (attacker) to authenticate and confirm c's request. After that confirmation, as responds with a redirect to c's redirect endpoint that includes an authorization code ac. The attacker follows that redirect and c obtains an access token from as (because the token endpoint c uses for interactions with as_{att} is $tokenEP_{as}$).

At this point, the attacker logs out of c, triggering c to send a revocation request to as_{att}, including a client assertion with audience $tokenEP_{as}$ (see Section 2.1.2). Note that the access token included in c's revocation request is only valid for resources of the attacker.

FAPI 1.0 and FAPI 2.0. While both FAPI protocol families employ additional security measures, both are directly susceptible to the first phase of the audience injection attack as presented in Section 3.1: while for FAPI 1.0 support for PAR is optional, FAPI 2.0 mandates the use of PAR.

Token Introspection. With the OAuth 2.0 Token Introspection [61] extension, a client/resource server can inquire about the status of an access token at the issuing AS, for example, to learn the expiration time of such a token (see Section 4 of [61]). For token introspection, the AS offers an *introspection endpoint* that expects an access token and client authentication. The attack is then very similar to the token revocation case above, except that as_{att}'s configuration document of course lists an introspection endpoint, pointing to as_{att}, and the client sends an introspection request once it received the tokens, i.e., there is no need for the attacker to further interact with the client.

CIBA. With the CIBA protocol presented in Section 2.2, the first attack phase becomes even simpler: the attacker AS as_{att} includes a backchannel authentication endpoint, pointing to as_{att}, in its configuration document and asks the client to start a protocol run with as_{att} and some made-up identity. The client will then send an authenticated, i.e., containing a client assertion, request to as_{att}'s backchannel authentication endpoint.

OAuth 2.0 Device Authorization Grant. As mentioned in Section 1, the Device Authorization Grant [19] standard defines a decoupled flow. The flow begins with the consumption device requesting a user code and a device code from the AS's *device authorization endpoint*. The user code is then shown to the user, along with the authorization endpoint of the AS – the end-user is then supposed to visit the AS's authorization endpoint, enter the user code, authenticate, review the consumption device/client's request, and so on. Importantly, that first request from the consumption device to the AS is authenticated, i.e., contains a client assertion. Hence, as_{att} publishes a configuration document with a device

[6] A key challenge for the attacker is to ensure that c obtains an access token (otherwise, c would not use token revocation): the attacker lists $tokenEP_{as}$ as its token endpoint, therefore, c sends its token request to that endpoint, i.e., the honest as. However, without knowledge of the protocol run initiated by the attacker, as would reject the request, hence, the need for the attacker to interact with as.

authorization endpoint that points to $as_{\mathtt{att}}$, the attacker triggers the consumption device to start a protocol run with $as_{\mathtt{att}}$, and $as_{\mathtt{att}}$ receives a client assertion with the audience value $tokenEP_{as}$.

Reducing Attack Footprint. In the instance of the first attack phase described in Section 3.1, the attacker initiates a protocol run with the honest c (Step 4 of Figure 3). The attack ends once the attacker obtains c's PAR request and a valid client assertion. However, simply abandoning the protocol run at this point may lead to an investigation by c's operators, particularly in closely monitored high-risk environments.

If the attacker continues the protocol run (see Figure 1), c will eventually send a token request to the token endpoint it associates with the protocol run's AS (the attacker), i.e., $tokenEP_{as}$ (the token endpoint of the honest AS as). Since as is unaware of this interaction, it will return an error, again risking unwanted attention.

If c supports both client assertion and mutual TLS (mTLS) authentication – common in high-risk environments – the attacker can avoid this: the standards for PAR [50] and for mTLS client authentication [13] permit different authentication methods at the PAR and token endpoints, determined by the AS's configuration. Furthermore, AS configurations may specify a dedicated token endpoint for mTLS authentication in addition to the "regular" token endpoint that is used for everything else.

The attacker now lists $tokenEP_{as}$ as the regular token endpoint and specifies an mTLS token endpoint pointing to the attacker. During registration, the attacker instructs c to use client assertions at the PAR endpoint and mTLS authentication at the token endpoint. Consequently, the token request for the attacker-initiated protocol run is sent to the attacker's mTLS endpoint, ensuring it does not fail, while c's PAR request is still authenticated with a client assertion using $tokenEP_{as}$ as its `aud` claim.

JWT Authorization Grants. Besides defining a framework for assertion-based client authentication, RFC 7521 [15] also introduces such a framework for an authorization grant based on assertions; which is instantiated in RFCs 7522 and 7523 [14, 41]. This grant requires the client to obtain an assertion from a so-called *security token service* (STS). The client then exchanges the assertion for an access token via a request to the token endpoint of the AS. In the following, we describe a variation of the attack based on RFC 7523, which defines JWT-based assertions; the attack is similar for RFC 7522, which uses SAML assertions. In both cases, such an assertion is signed by the issuer and contains an identifier of the issuer, i.e., the STS, a subject, e.g., the user on whose behalf the STS issued the assertion, an audience, typically the token endpoint URI of the AS at which the client is supposed to use the assertion, and some additional values like a nonce for de-duplication and an expiration time.

In contrast to the previous variants, the attacker's goal is not to obtain an authentication assertion for impersonating the client, but an assertion that the attacker can exchange for an access token at the honest AS.

In a nutshell, the attacker starts the attack at an honest client c and selects $as_{\mathtt{att}}$ as the AS. Hence, c then requests an assertion from the STS for use at $as_{\mathtt{att}}$. We assume that the STS uses the token endpoint that $as_{\mathtt{att}}$ (maliciously) publishes in its configuration document as the assertion's audience value, i.e., the token endpoint of as. Hence, the assertion issued by the STS looks as follows: $\mathrm{sign}(\langle \mathtt{iss}\text{: sts}, \mathtt{sub}\text{: hon-user@ex.com}, \mathtt{aud}\text{: } tokenEP_{as}, \ldots \rangle, \hat{k}_{\mathrm{STS}})$. This assertion is then used by c in a token request to $as_{\mathtt{att}}$, and $as_{\mathtt{att}}$ can use it to obtain an access token as as.

Note that this attack requires the additional assumption that there is disagreement between the client and the STS on the token endpoint of $as_{\mathtt{att}}$: from STS' perspective, the token endpoint of $as_{\mathtt{att}}$ must be $tokenEP_{as}$, but from c's perspective, it must be an actual endpoint of $as_{\mathtt{att}}$ – otherwise, c would send the assertion to $tokenEP_{as}$, thus not leaking it to the attacker. Thus, for the attack to work, the malicious AS must distribute different configuration documents to c and the STS, for example, depending on the IP address from which the configuration document is requested.

4.2 Further Instances of the Second Attack Phase

For the second phase, the attacker already obtained client assertions signed by an honest client c with a client ID cid_c in the `iss` and `sub` claims, and an honest AS as's token endpoint $tokenEP_{as}$ in the `aud` claim. The attacker's goal is to compromise the authorization and/or authentication security goals.

Client Credentials Grant. As mentioned in Section 1, the client credentials grant [33] allows a client to request an access token for its own resources token from an AS. To do so, the client just sends an authenticated token request to the AS's token endpoint, i.e., without an authorization code, and the AS responds with a corresponding access token. Since the attacker has a valid client assertion to authenticate as c at as, the attacker can send such a token request, thus compromising the authorization goal by gaining access to c's resources.

Decoupled Flows. With decoupled flows like **CIBA** (see Section 2.2), the attacker can initiate a forged request to as's backchannel authentication endpoint, impersonating c and specifying an arbitrary user ID id (Step 1 of Figure 2). In response, as issues the attacker an identifier $authId$, which the attacker can use to poll as's token endpoint. The user identified by id is then prompted by (the honest and trusted) as to approve (the honest and trusted) c's request. Once

the user gives their consent, the attacker obtains an access token for that user's resources, thereby compromising the authorization goal. This attack can be further facilitated by sending the user an email that appears to originate from c, informing them of the need for re-authorization. This makes the request from as seem legitimate and expected to the user. It is important to note that this attack is also applicable to CIBA's push and ping modes.

Furthermore, a very similar second-phase attack can be carried out for the **OAuth 2.0 Device Authorization Grant**.

FAPI 1.0 and FAPI 2.0. In Section 2.3, we briefly introduced the PKCE extension [64] used in FAPI 2.0 that is supposed to prevent attackers from exchanging leaked authorization codes for tokens by including the hash of a nonce in the PAR request and the nonce itself in the token request. However, this security measure does not prevent the attack presented in Section 3.2, because the attacker is the sender of both the PAR request (Step 1 in Figure 4) and the token request (Step 9). Likewise, none of the other security mechanisms included in FAPI 2.0, such as access token sender constraining, prevent the attack.

For FAPI 1.0, a specific clause in the standards happens to prevent a successful second-phase attack.

Attack on Authentication. The authentication goal is compromised when the attacker gets logged in at an honest client under an honest user's identity the user has at an honest AS. We illustrate such an attack using *implicit flows*; another type of flow included in many OMAA protocols besides the ones discussed in Section 2.1 (i.e., authorization code, client credentials, and decoupled flows).

In an implicit flow, Steps 5–7 of Figure 1 are omitted. Instead, the client directly includes all request details, such as the scope of requested access and client ID (but no client assertion), in the redirect in Step 8. After the user provides consent, the redirect from the AS to the client (Steps 11 and 12 of Figure 1) already contains the requested ID tokens with user identity information, eliminating the need for a separate token request.

To mitigate certain injection attacks, the implicit flow requires the client to include a nonce n_c in its initial (and only) request, i.e., the modified Step 8. The AS includes n_c in the ID token and the client must validate the nonce before accepting the token (in addition to the ID token validation rules described in Section 2.1.1).

Now, in a nutshell for the attack on authentication the attacker (as a user) starts an implicit flow with c, asking to authenticate with as, thus receiving n_c. In a separate protocol run (that may use any type of flow, see below), the attacker then obtains an ID token idt_{n_c} with an honest user u's details and the nonce n_c, issued by as for c (i.e., containing cid_c). The attacker (as an AS) then returns idt_{n_c} to c in the initial implicit flow and gets logged in as u at c.

To obtain idt_{n_c}, the attacker can, for example, use the second-phase attack as described in Section 3.2, except that in Step 7 of Figure 4, the attacker adds n_c to the PAR request parameters (the nonce can be used with any flow, but is optional in the authorization code flow). Consequently, as includes n_c in the issued ID token (that the attacker receives, see Step 10 of Figure 4).

4.3 Discussion

As previously noted, the two phases of the attack are largely independent, allowing an attacker to freely combine different options for each phase. For instance, an attacker could acquire client assertions via a client's CIBA backchannel authentication endpoint and then use them in a client credentials flow to access the client's resources. This example also highlights that attacks can span multiple protocols within the OMAA families, such as CIBA and OAuth 2.0 with one of the signature-based client authentication methods. This flexibility results in a wide range of potential attack instances, affecting various standardized client authentication methods, base protocols, and extensions (see Section 6).

Notably, for both phases, it suffices if the client and/or AS merely support a vulnerable protocol, even if that protocol is never used in their interactions. E.g., in the example above, the client might never want to use the CIBA flow with the honest AS targeted by the attacker. This is particularly concerning given the widespread adoption of libraries and SDKs, which often include support for various OMAA protocols and configurations.

In general, an opportunity for an audience injection attack arises whenever a client authenticates at multiple endpoints of an AS.

5 Fixes

At its core, the first phase of audience injection attacks exploits the ability of an attacker to trick an honest client into sending a client assertion with an attacker-chosen `aud` claim value A to an attacker-controlled endpoint B. Therefore, the entire class of audience injection attacks can be mitigated by ensuring this scenario cannot occur.

Hence, we discussed various approaches to achieve this for all protocols of the OMAA families with members of the responsible standardization bodies. These discussions quickly boiled down to three options that are particularly straightforward to integrate into existing standards and avoid introducing additional messages, secrets, or values.

AS-side Fixes. Our initial discussions with the standardization bodies (see Section 6) revealed a strong inclination to address security issues at the AS, e.g., by requiring a specific audience value (instead of accepting multiple different ones). This preference stems from the perception of clients as lower-trust participants and the fact that the number of clients far exceeds the number of ASs in most ecosystems.

However, audience injection attacks cannot be mitigated solely by the AS, at least not without introducing additional values or messages: consider, for instance, the client assertion obtained by the attacker in Step 7 of Figure 3. The attacker can choose any value for the token endpoint in its configuration document (Step 5), in particular the value expected by the honest AS, bypassing any AS-level defenses. Thus, addressing audience injection attacks necessitates changes in how the client determines the `aud` value for its client assertions.

Actual Endpoint as Audience. A somewhat obvious option is to mandate that clients always set the `aud` claim value to the *exact* endpoint where the corresponding client assertion will be used. For example, in Step 7 of Figure 3, the client would set the `aud` claim to $parEP_{\mathsf{att}}$, which corresponds to a URL controlled by the attacker. The honest AS, of course, does not identify itself with that URL.

Nevertheless, this approach has practical limitations and challenges, for example, in handling alternative endpoint URLs, such as those for mTLS (see "Reducing Attack Footprint" in Section 4.1). These scenarios complicate the consistent application of this requirement and may introduce additional implementation overhead.

AS Issuer Identifier as Audience. In the protocol descriptions in Section 2, we mention the *AS issuer identifier*, which is essentially the HTTPS domain of the AS. Clients already rely on this value when using extensions like Discovery, as discussed in Section 2. And even without such extensions, clients need the AS issuer identifier to validate the issuer of ID tokens. As its name implies, the issuer identifier *uniquely identifies* an AS, making it a suitable value for the `aud` claim to prevent the attack. For instance, in the first-phase attack described in Section 3.1, the client would have to use https://attacker.com as the `aud` claim value for the client assertion in Step 7 of Figure 3. Consequently, the client assertion would be invalid for use at the honest AS, as the honest AS does not identify itself with that `aud` value.

As detailed in Section 6, this approach is being adopted in updated and new standards, as well as in existing deployments to mitigate audience injection attacks, due to its straightforward integration into both standards and implementations without requiring additional messages or values.

6 Impact and Responsible Disclosure

As outlined in Section 1, we contacted the OIDF and authors of the IETF standards, who considered the class of attacks we found a significant threat. After our initial disclosure in September 2024, the ensuing discussions among the involved authors, the OIDF leadership, ourselves, and additional people brought into the disclosure process by the OIDF resulted in a coordinated effort to update affected deployments, implementations, and standards. In the following, we give an overview of these efforts. Given the severity of a successful attack, the OIDF and standards authors decided to keep this process confidential until the involved parties had sufficient time to implement mitigations.

Due to the public nature of these standards, the finalization of some updates requires a public discussion. In coordination with the OIDF and standards authors, this public discussion, and hence, public disclosure, began end of January 2025.

6.1 Implementations and Deployments

To prevent potential attacks on existing deployments, the OIDF organized a responsible disclosure process with several important ecosystems and implementers. We supported this disclosure process on the technical side regarding details of the attacks and potential fixes. Note that we only give a rough overview and cannot name all affected parties.

FAPI Ecosystems. The FAPI 1.0 and FAPI 2.0 protocol families are widely used in high-risk ecosystems, such as Open Banking in the US, UK, Brazil, Australia, the UAE, and Saudi Arabia, as well as Norway's healthcare sector and Brazil's open insurance sector, collectively serving hundreds of millions of users. To ensure these ecosystems understand the threats and mitigation strategies, the OIDF individually notified them. For around 20 key high-risk ecosystems, the OIDF also organized meetings with their technical leadership.

Major Implementers. Besides the FAPI ecosystems, major implementers and SaaS providers of OMAA protocols, such as Microsoft, Ping Identity, Verimi, Okta, and Authlete, and other important OMAA software vendors were informed through the OIDF. These stakeholders have deployments of all protocols and extensions of the OMAA family of protocols.

OpenID Certification Tests. The OIDF offers a certification program [57] to test the conformance of implementations with various OIDF standards, with several hundred certified implementations. These conformance tests are being updated to reflect the required changes to the specifications (see below).

6.2 Specifications

As mentioned before, audience injection attacks affect a wide range of specifications, hence, preventing them requires updates to various standards. Note that while some of the standards that need to be updated are technically still drafts,[7] most of them are final specifications. As detailed below, new versions are planned and, at the time of writing, are being worked on. We emphasize that updating final standards has been quite rare in the past, since this is a complex process that requires consensus and public discussions within the standardization bodies. Standardization bodies often preferred to point out and fix security issues in separate best practice documents like [48] or as errata. The planned new versions for final specifications, hence, highlight the relevance of audience injection attacks from the standardization bodies' perspective. In the following, we give an overview of the changes planned and currently being worked on by the standardization bodies.

OAuth 2.0 Assertions. The RFCs 7521, 7522, and 7523 define assertions that, amongst others, can be used for client authentication (see Section 2.1.2). In particular, the token endpoint of the intended receiver is explicitly allowed as a possible audience value. There is ongoing work to publish a new standard that obsoletes these RFCs to restrict the audience value to the AS issuer identifier [42], as discussed in Section 5.

OpenID Connect. OIDC defines the Private Key JWT client authentication method and recommends using the token endpoint as the audience: "The Audience SHOULD be the URL of the Authorization Server's Token Endpoint" (Section 9 of [68]). When considered in isolation, OIDC is unaffected by this attack because clients authenticate only to the token endpoint, preventing attackers from obtaining a pkJWT with an honest AS's token endpoint in its audience claim.

However, the Private Key JWT method is included in the IANA registry[8] for OAuth 2.0 token endpoint authentication methods [37], making this method (and with it, audience injection attacks) applicable wherever specifications require client authentication, such as the revocation endpoint (**OAuth 2.0 Token Revocation**, Section 2.3 of [51] and [45]), the introspection endpoint (**OAuth 2.0 Token Introspection**, [61]), the device authorization endpoint (**OAuth 2.0 Device Authorization Grant**, Section 3.1 of [19]), and any other endpoint that requires client authentication (see also below).

At the time of writing, the OIDF plans to publish a new version of the OIDC specification to restrict the audience value of pkJWTs to the AS issuer identifier. OIDC was also published as an ISO standard [38], which will be updated as well.

Security Best Current Practices. The OAuth 2.0 Security Best Current Practice document [48] is a collection of best practices that have evolved over the years. Even though that document has been published very recently, the IETF OAuth Working group asked us to propose an update that describes the attack and mitigation options, and plans to publish the update fast.

FAPI. As already mentioned in Section 1, the FAPI 1.0 and FAPI 2.0 specifications were formally analyzed in [23, 35], but only [35] considered the pkJWT client authentication method. However, the specification draft analyzed there mandated the AS issuer identifier as the client assertion's `aud` value – not because of security concerns, but to improve interoperability. Unfortunately, a later change made to be backwards-compatible allowed the token endpoint to be used. Due to our findings, this change was reverted in [74]. At the time of writing, the responsible Working Group is updating FAPI 1.0 accordingly.

CIBA. CIBA explicitly requires the AS to accept its token endpoint URL as the client assertion audience value (see Section 7.1 of [20]). Therefore, just updating the audience requirements in the OIDC standard is not sufficient. Hence, at the time of writing, the CIBA specification is being updated to require usage of the AS issuer identifier.

OpenID Federation. OpenID Federation is a mature draft specification that defines, among other things, additional registration mechanisms for OIDC. While OpenID Federation is technically still a draft, it is already used in practice, for example, as part of the implementation of the Italian Digital Identity Wallet and within the digital infrastructure of the German healthcare system [29, 39]. Due to our findings, the responsible Working Group decided to require the issuer identifier as the sole audience value [47, Sec. 12.1.1.2].

[7] However, due to how the standardization processes work, these drafts often are in widespread use. For example, FAPI 2.0 has been in widespread use in high-risk environments for several years before it was moved from a draft to a final specification. Another example is OpenID Federation, which is still a draft, see the corresponding paragraph below.

[8] The *Internet Assigned Numbers Authority (IANA)* maintains a global registry that manages, amongst others, constant values like metadata names used in protocols.

PAR. The PAR standard explicitly requires the AS to accept its token endpoint URL as the audience value (see Section 2 of [50]). Ongoing work will obsolete this standard and replace it with a fixed version [42].

7 Conclusion

We introduced *audience injection attacks*, a novel class of vulnerabilities that target signature-based client authentication methods used across numerous protocols within the OMAA protocol families. These protocols underpin authentication and authorization services for billions of users worldwide, including high-risk applications. We initiated a responsible disclosure with the relevant standardization bodies and worked with them to develop effective fixes, which, in an ongoing effort, are included in updates to a multitude of standards and deployments.

Notably, while several OMAA protocols have undergone prior security analyses, including formal ones, these efforts have mostly ignored or not thoroughly examined the security of client authentication methods. We therefore consider a rigorous formal security analysis of these methods as a promising avenue for future research.

Acknowledgments

This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as grant 443324941, and the OpenID Foundation.

References

[1] Amazon AWS Cognito Documentation. Machine-to-machine (M2M) authorization. https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools-define-resource-servers.html#cognito-user-pools-define-resource-servers-about-m2m.

[2] Australian Banking Association. Open Banking. https://www.ausbanking.org.au/priorities/open-banking/.

[3] Auth0 by Okta. Credential settings. https://auth0.com/docs/get-started/applications/credentials#available-credentials.

[4] Auth0 by Okta. Extension settings. https://auth0.com/docs/get-started/applications/{application-settings#grant-types, application-settings#refresh-token-rotation, configure-fapi-compliance#configure-fapi-compliance-for-a-client, configure-par#set-par-for-a-tenant, configure-jar#configure-jar-for-an-application, configure-sender-constraining/configure-client-for-sender-constraining}.

[5] Authlete. Client settings. https://www.authlete.com/kb/oauth-and-openid-connect/client-authentication/client-auth-private-key-jwt/#client-settings.

[6] Authlete. Extension settings. https://www.authlete.com/developers/{pkce/#64-pkce-configuration, token_exchange/#configuring-token-exchange}, https://www.authlete.com/kb/operations/service-configuration/v2/service-settings.

[7] Banco Central do Brasil. Open finance. https://www.bcb.gov.br/en/financialstability/open_finance.

[8] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[9] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 247–262, 2012.

[10] Michele Benolli, Seyed Ali Mirheidari, Elham Arshad, and Bruno Crispo. The full gamut of an attack: An empirical analysis of oauth csrf in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41, 2021.

[11] Vittorio Bertocci. JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. RFC 9068, October 2021.

[12] Vittorio Bertocci and Brian Campbell. OAuth 2.0 Step Up Authentication Challenge Protocol. RFC 9470, September 2023.

[13] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705, February 2020.

[14] Brian Campbell, Chuck Mortimore, and Michael B. Jones. Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants. RFC 7522, May 2015.

[15] Brian Campbell, Chuck Mortimore, Michael B. Jones, and Yaron Y. Goland. Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants. RFC 7521, May 2015.

[16] Central Bank of the U.A.E. Open finance regulation. https://rulebook.centralbank.ae/en/rulebook/open-finance-regulation.

[17] Suresh Chari, Charanjit Jutla, and Arnab Roy. Universally composable security analysis of oauth v2.0. Cryptology ePrint Archive, Paper 2011/526, 2011. https://eprint.iacr.org/2011/526.

[18] William Denniss and John Bradley. OAuth 2.0 for Native Apps. RFC 8252, October 2017.

[19] William Denniss, John Bradley, Michael B. Jones, and Hannes Tschofenig. OAuth 2.0 Device Authorization Grant. RFC 8628, August 2019.

[20] Gonzalo Fernandez, Florian Walter, Axel Nennker, Dave Tonge, and Brian Campbell. OpenID Connect Client-Initiated Backchannel Authentication Flow – Core 1.0, 2021. https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html.

[21] Daniel Fett. FAPI 2.0 Attacker Model, February 2025. https://openid.net/specs/fapi-attacker-model-2_0-final.html.

[22] Daniel Fett, Brian Campbell, John Bradley, Torsten Lodderstedt, Michael B. Jones, and David Waite. OAuth 2.0 Demonstrating Proof of Possession (DPoP). RFC 9449, September 2023.

[23] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. An Extensive Formal Security Analysis of the OpenID Financial-grade API. In *IEEE S&P*, pages 1054–1072, Los Alamitos, CA, USA, 5 2019. IEEE Computer Society.

[24] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE CSF*. IEEE Computer Society, 2017.

[25] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[26] Daniel Fett and Dave Tonge. FAPI 2.0 Message Signing – 1st Implementers Draft, March 2023. https://openid.net/specs/fapi-2_0-message-signing-ID1.html.

[27] Daniel Fett, Dave Tonge, and Joseph Heenan. FAPI 2.0 Security Profile, February 2025. https://openid.net/specs/fapi-security-profile-2_0-final.html.

[28] Gematik. gemSpec_IDP_Dienst_1.7.0. https://gemspec.gematik.de/docs/gemSpec/gemSpec_IDP_Dienst/gemSpec_IDP_Dienst_V1.7.0/.

[29] Gematik. gemSpec_IDP_FedMaster_V1.3.0. https://gemspec.gematik.de/docs/gemSpec/gemSpec_IDP_FedMaster/gemSpec_IDP_FedMaster_V1.3.0/.

[30] Google Developers Documentation. Sign-in on tvs and limited input devices. https://developers.google.com/identity/gsi/web/guides/devices.

[31] Google Developers Documentation. Using OAuth 2.0 for Server to Server Applications. https://developers.google.com/identity/protocols/oauth2/service-account.

[32] Eran Hammer-Lahav and Mark Nottingham. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785, April 2010.

[33] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.

[34] Helsenorge. Services provided at helsenorge. https://www.helsenorge.no/en/about-services-on-helsenorge/how-to-use-helsenorge-services#services-provided-at-helsenorge.

[35] Pedram Hosseyni, Ralf Küsters, and Tim Würtele. Formal Security Analysis of the OpenID FAPI 2.0: Accompanying a Standardization Process. In *37th IEEE Computer Security Foundations Symposium (CSF 2024)*. IEEE, 2024.

[36] Pedram Hosseyni, Ralf Küsters, and Tim Würtele. Formal Security Analysis of the OpenID FAPI 2.0 Family of Protocols: Accompanying a Standardization Process. *ACM Transactions on Privacy and Security*, 28(1):1–36, 2024.

[37] Internet Assigned Numbers Authority. OAuth Parameters. https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml.

[38] ISO/IEC 26131:2024. Information technology — OpenID connect — OpenID connect core 1.0 incorporating errata set 2. Technical report, October 2024.

[39] Italian EUDI Wallet Implementation. The Infrastructure of Trust. https://italia.github.io/eudi-wallet-it-docs/v0.9.3/en/trust.html.

[40] Louis Jannett, Christian Mainka, Maximilian Westers, Andreas Mayer, Tobias Wich, and Vladislav Mladenov. Sok: SSO-MONITOR - the current state and future research directions in single sign-on security measurements. In *9th IEEE European Symposium on Security and Privacy, EuroS&P 2024, Vienna, Austria, July 8-12, 2024*, pages 173–192. IEEE, 2024.

[41] Michael B. Jones, Brian Campbell, and Chuck Mortimore. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. RFC 7523, May 2015.

[42] Michael B. Jones, Brian Campbell, and Chuck Mortimore. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. Internet-Draft draft-ietf-oauth-rfc7523bis, Internet Engineering Task Force, February 2025. Work in Progress.

[43] Michael B. Jones and Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, October 2012.

[44] Michael B. Jones, Anthony Nadalin, Brian Campbell, John Bradley, and Chuck Mortimore. OAuth 2.0 Token Exchange. RFC 8693, January 2020.

[45] Michael B. Jones, Nat Sakimura, and John Bradley. OAuth 2.0 Authorization Server Metadata. RFC 8414, June 2018.

[46] Michael B. Jones, Andreas Åkre Solberg, John Bradley, Giuseppe De Marco, and Vladimir Dzhuvinov. OpenID Federation 1.0 – 4th Implementer's Draft, 2024. https://openid.net/specs/openid-federation-1_0-ID4.html.

[47] Michael B. Jones, Andreas Åkre Solberg, John Bradley, Giuseppe De Marco, and Vladimir Dzhuvinov. OpenID Federation 1.0 – Latest Editor's Draft, 2024. https://openid.net/specs/openid-federation-1_0.html.

[48] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. Best Current Practice for OAuth 2.0 Security. RFC 9700, January 2025.

[49] Torsten Lodderstedt and Brian Campbell. JWT Secured Authorization Response Mode for OAuth 2.0 (JARM), 2018. OpenID Foundation.

[50] Torsten Lodderstedt, Brian Campbell, Nat Sakimura, Dave Tonge, and Filip Skokan. OAuth 2.0 Pushed Authorization Requests. RFC 9126, September 2021.

[51] Torsten Lodderstedt, Stefanie Dronia, and Marius Scurtescu. OAuth 2.0 Token Revocation. RFC 7009, August 2014.

[52] Torsten Lodderstedt, Justin Richer, and Brian Campbell. OAuth 2.0 Rich Authorization Requests. RFC 9396, May 2023.

[53] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. Sok: Single sign-on security - an evaluation of openid connect. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 251–266. IEEE, 2017.

[54] Microsoft Entra Documentation. Microsoft identity platform and the oauth 2.0 device authorization grant flow. https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-device-code.

[55] NHS. Electronic Prescription Service - FHIR API. https://digital.nhs.uk/developer/api-catalogue/electronic-prescription-service-fhir.

[56] Open Banking Limited. Open Banking UK. https://www.openbanking.org.uk/.

[57] OpenID Foundation. OpenID Certification. https://openid.net/certification/.

[58] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika Pai, and Sanjay Singh. Formal verification of OAuth 2.0 using alloy framework. In *2011 International Conference on Communication Systems and Network Technologies*. IEEE, 2011.

[59] Pieter Philippaerts, Davy Preuveneers, and Wouter Joosen. Oauch: Exploring security compliance in the oauth 2.0 ecosystem. In *25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2022, Limassol, Cyprus, October 26-28, 2022*, pages 460–481. ACM, 2022.

[60] Justin Richer. TxAuth BoF Summary. https://datatracker.ietf.org/meeting/106/materials/slides-106-oauth-sessb-txauth-bof-00.

[61] Justin Richer. OAuth 2.0 Token Introspection. RFC 7662, October 2015.

[62] Justin Richer, Michael B. Jones, John Bradley, and Maciej Machulak. OAuth 2.0 Dynamic Client Registration Management Protocol. RFC 7592, July 2015.

[63] Justin Richer, Michael B. Jones, John Bradley, Maciej Machulak, and Phil Hunt. OAuth 2.0 Dynamic Client Registration Protocol. RFC 7591, July 2015.

[64] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015.

[65] Nat Sakimura, John Bradley, and Edmund Jay. Financial-grade API Security Profile 1.0 - Part 1: Baseline, 2021. https://openid.net/specs/openid-financial-api-part-1-1_0.html.

[66] Nat Sakimura, John Bradley, and Edmund Jay. Financial-grade API Security Profile 1.0 - Part 2: Advanced, 2021. https://openid.net/specs/openid-financial-api-part-2-1_0.html.

[67] Nat Sakimura, John Bradley, and M. Jones. OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 2, 2023. https://openid.net/specs/openid-connect-registration-1_0.html.

[68] Nat Sakimura, John Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 2, 2023. http://openid.net/specs/openid-connect-core-1_0.html.

[69] Nat Sakimura, John Bradley, M. Jones, and E. Jay. OpenID Connect Discovery 1.0 incorporating errata set 2, 2023. http://openid.net/specs/openid-connect-discovery-1_0.html.

[70] Nat Sakimura, John Bradley, and Michael B. Jones. The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR). RFC 9101, August 2021.

[71] Saudi Central Bank. Ksa open banking program. https://openbanking.sa/index-en.html.

[72] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 239–260. Springer International Publishing, 2015.

[73] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and Communications Security - CCS '12*. ACM Press, 2012.

[74] Dave Tonge. Boom! AS to enforce aud as a single value issuer. FAPI Working Group Bitbucket Repository, Commit 6f88d61, October 2024. https://bitbucket.org/openid/fapi/commits/6f88d61afa7ca3f4f29e47395c91f8f4e4c5d8ad.

[75] Davon Tonge, Joseph Heenan, Torsten Lodderstedt, and Brian Campbell. Financial-grade API: Client Initiated Backchannel Authentication Profile – 1st Implementers Draft, 2019. OpenID Foundation.

[76] The Treasury. Data standards body, 2025. https://dsb.gov.au/.

[77] US Department of Health and Human Services. 21st Century Cures Act: Interoperability, Information Blocking, and the ONC Health IT Certification Program. https://www.federalregister.gov/documents/2020/05/01/2020-07419/21st-century-cures-act-interoperability-information-blocking-and-the-onc-health-it-certification.

[78] Haixing Yan, Huixing Fang, Christian Kuka, and Huibiao Zhu. Verification for oauth using aslan++. In *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015*, pages 76–84. IEEE Computer Society, 2015.

[79] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016.

[80] Yuchen Zhou and David Evans. Ssoscan: Automated testing of web applications for single sign-on vulnerabilities. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium*, pages 495–510. USENIX Association, 2014.

[81] Karsten Meyer zu Selhausen and Daniel Fett. OAuth 2.0 Authorization Server Issuer Identification. RFC 9207, March 2022.

All figures were created with annexlang, see https://github.com/danielfett/annexlang.