Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: 6 November 2026

K. McGuinness
Independent

5 May 2026

OAuth 2.0 Client Instance Assertions using Actor Tokens
draft-mcguinness-oauth-client-instance-assertion-00

Abstract

This specification defines a profile for representing OAuth 2.0 client instance identity to an authorization server. It registers a new actor_token_type, urn:ietf:params:oauth:token-type:client-instance-jwt, that carries the instance identity as a signed JWT presented via the actor_token parameter defined by OAuth 2.0 Token Exchange (RFC 8693). To support presentation outside token exchange, this specification also permits actor_token and actor_token_type on selected additional grant types.

The profile does not introduce a new client_instance identifier in protocol messages. Instead, it defines client metadata parameters (applicable to clients identified by a Client ID Metadata Document (CIMD) or registered via OAuth Dynamic Client Registration (RFC 7591)) that let a client_id identify a logical client whose concrete runtime instances are authenticated by one or more trusted instance issuers (for example, workload identity systems).

The Authorization Server validates the instance assertion and represents the instance either as an act claim, when another principal is present (e.g., a user delegating to the instance), or as the access token’s sub, when the instance itself is the principal (e.g., a client credentials grant). The issued access token is sender-constrained to a key the instance possesses.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://mcguinness.github.io/draft-mcguinness-oauth-client-instance-assertion/draft-mcguinness-oauth-client-instance-assertion.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mcguinness-oauth-client-instance-assertion/.

Discussion of this document takes place on the Web Authorization Protocol Working Group mailing list (mailto:oauth@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/oauth/. Subscribe at https://www.ietf.org/mailman/listinfo/oauth/.

Source for this draft and an issue tracker can be found at https://github.com/mcguinness/draft-mcguinness-oauth-client-instance-assertion.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 6 November 2026.

Copyright Notice

Table of Contents

1. Introduction 4
2. Conventions and Definitions 6
3. Relationship to Other Specifications 7
4. Actor Token Grant Extension 9
4.1. Permitted Grant Types 10
4.2. Other Actor Token Types 10

5. Client Instance Model 11
5.1. Architecture 11
5.2. Client Registration Models 13
5.3. Trust Delegation Model 14
5.3.1. Delegation by the OAuth Client 14
5.3.2. Authority of the Authorization Server 14
5.3.3. Trust Lifecycle 15
5.3.4. Cross-Organization Federation 15
6. Metadata and Discovery 15
6.1. Client Metadata Extensions 15
6.1.1. instance_issuers 16
6.2. Authorization Server Metadata 18
7. Client Instance Assertion Format 19
7.1. Issuer Obligations 20
7.2. JWT Claims 20
7.3. Signing and JOSE Header 23
7.4. Example Assertion 23
8. Token Endpoint Processing 24
8.1. Token Request 24
8.2. Authorization Server Processing 25
8.3. Client Authentication via Instance Assertion 27
8.3.1. Request Format 28
8.3.2. Validation Procedure 28
8.4. Authorization-Time Consistency 29
8.5. Sender-Constrained Access Tokens 31
8.6. Access Token Representation 33
8.6.1. Classification 33
8.6.2. Delegation Case 34
8.6.3. Self-Acting Case 34
8.6.4. Actor Chain Merging 35
8.7. Refresh Tokens 36
8.8. SPIFFE Compatibility 37
8.8.1. Client ID Claim Omission 39
8.8.2. SPIFFE Trust Bundle Resolution 40
8.8.3. Sender-Constraint Binding for Raw JWT-SVIDs 41
8.9. Error Responses 41
9. Resource Server Processing 42
10. Adoption and Migration 44
11. Conformance 46
12. Security Considerations 46
12.1. Trust Model 46
12.2. Trust-Withdrawal Latency 47
12.3. Instance Lifecycle 47
12.4. Token Revocation 48
12.4.1. Per-Token Revocation 48
12.4.2. Per-Instance Revocation 49
12.4.3. Introspection Behavior on Revocation 49
12.4.4. Revocation and Refresh Tokens 50

12.5. Replay 50
12.6. Audience and Confused Deputy 51
12.7. Trust-Root Collapse 51
12.8. Mode-Switch Between Delegation and Self-Acting 51
12.9. Sender-Constraint Requirement 52
12.10. Delegation Control 52
12.11. Privacy 52
13. IANA Considerations 53
13.1. OAuth Token Type 53
13.2. OAuth Dynamic Client Registration Metadata 53
13.2.1. instance_issuers 53
13.3. OAuth Token Endpoint Authentication Method 53
13.4. OAuth Authorization Server Metadata 54
13.4.1. actor_token_types_supported 54
13.5. Media Type 54
13.6. OAuth Entity Profile 55
14. References 55
14.1. Normative References 55
14.2. Informative References 57
Design Rationale 58
Why not a client_instance request parameter? 58
Why not a dedicated client_instance_assertion request parameter? 58
Why extend actor_token to non-token-exchange grants? 59
Why client metadata as the trust anchor for instance issuers? 60
Why a dedicated actor_token_type URN? 60
Why a token_endpoint_auth_method rather than a client_assertion_type? 60
Why reuse actor_token in the self-acting case? 61
Worked Examples 62
Authorization Code with User Delegation 62
Client Credentials (Self-Acting) 65
Token Exchange with Prior Delegation Chain (Agent Spawns Sub-Agent) 66
SPIFFE Workload (Self-Acting, JWT-SVID Reuse with X.509-SVID Binding) 68
Acknowledgments 70
Author’s Address 70

1. Introduction

OAuth 2.0 [RFC6749] defines client_id as the identifier of a client. In modern deployments such as agentic workloads, autoscaled services, and ephemeral function executions, a single logical client routinely corresponds to many concrete runtime instances that come and go on a short timescale. Resource servers and authorization servers increasingly need to know not only _which_ client made a request but

which instance of that client made it. Instances may be acting on a user's behalf or as the principal themselves; this profile covers both.

OAuth 2.0 Token Exchange [RFC8693] defines the actor_token and actor_token_type token request parameters and the act claim for representing an actor in an issued token. The OAuth Actor Profile [ACTOR-PROFILE] further constrains the act claim and registers actor-related claims, but explicitly leaves out a token request parameter for proving an actor in flows other than token exchange.

This document defines a profile for representing client instance identity at the OAuth 2.0 token endpoint. It:

* Recognizes that an OAuth client_id commonly abstracts over many concrete runtime instances (a relationship already implicit in deployed OAuth practice; see Section 5), and defines client metadata describing the _instance issuers_ trusted to attest those instances. The metadata applies whether the client is identified by a Client ID Metadata Document [CIMD] or registered via [RFC7591]; see Section 5.2.

* Registers a new actor_token_type, urn:ietf:params:oauth:token-type:client-instance-jwt, that carries the instance identity as a JSON Web Token (JWT) [RFC7519] signed by an instance issuer published in the client's metadata.

* Requires issued access tokens to be sender-constrained to a key the instance possesses, and specifies how the instance assertion's cnf claim drives that binding in the interoperable re-minted assertion format.

* Registers client_instance_jwt as a token_endpoint_auth_method value, allowing deployments without an online client-controlled credential to authenticate the client via the instance assertion alone.

* Defines first-class support for SPIFFE workload identity, including optional direct presentation of JWT-SVIDs as actor_token.

* Defines authorization server metadata so that clients can discover support.

To enable instance assertions to be presented outside token exchange, this document also extends [RFC8693]'s actor_token and actor_token_type parameters to selected additional grant types (authorization_code, client_credentials, refresh_token, and the JWT bearer grant [RFC7523]). See Section 4; the extension is independent of any specific actor_token_type and may be used by other profiles.

What this document does _not_ do:

* It does not introduce a client_instance or client_instance_assertion request parameter (see Appendix "Why not a dedicated client_instance_assertion request parameter?").

* It does not change the syntax or processing of the act claim beyond what [ACTOR-PROFILE] already defines.

* It does not define authorization endpoint interactions for conveying actor identity; like [ACTOR-PROFILE], this is left for future work.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the following terms:

OAuth Client: As defined in [RFC6749], identified by a client_id (a CIMD URL or a registered client_id under [RFC7591]; see Section 5.2). In this profile, the OAuth client publishes the set of instance issuers permitted to authenticate its runtime instances (Section 5).

Client Instance: A concrete runtime of an OAuth client (for example, a particular process, container, function invocation, or session). See Section 5 for the class-and-instance relationship between an OAuth client and its instances.

Instance Issuer: An authority trusted by the client to authenticate client instances and issue instance assertions for those instances. Examples include workload identity providers (e.g., a SPIFFE control plane [SPIFFE]) and platform-managed identity services.

Client Instance Assertion: A JWT issued by an instance issuer

asserting the identity of a client instance, presented as the actor_token in a token request. This document distinguishes "actor_token" (the RFC 8693 request parameter, always typeset in backticks) from "Client Instance Assertion" or "instance assertion" in prose (the JWT artifact defined by this profile, which is what the actor_token parameter carries when its actor_token_type is urn:ietf:params:oauth:token-type:client-instance-jwt).

Actor Token Grant Extension: The wire-level extension defined in Section 4 permitting actor_token and actor_token_type ([RFC8693]) on the grant types listed in Section 4.1. This document's client-instance-jwt actor token type builds on the extension; other profiles may define their own actor token types under it.

Delegation Case: A token request whose grant produces a principal distinct from the instance presenting the actor_token (for example, a user under authorization_code or jwt-bearer). The issued access token's sub is the principal and the instance appears in act per Section 8.6.2.

Self-Acting Case: A token request whose grant produces no principal distinct from the instance (notably client_credentials). The issued access token's sub is the instance and act is omitted per Section 8.6.3.

3. Relationship to Other Specifications

RFC 8693 (Token Exchange). [RFC8693] defines actor_token and actor_token_type only on the token-exchange grant. This document permits these parameters on additional grant types (Section 4) and registers a new actor_token_type for asserting client instance identity (Section 7); the two contributions are separable. For the client-instance-jwt token type, delegation requests represent the instance in act (Section 8.6.2) and self-acting requests (notably client_credentials) represent it in top-level sub (Section 8.6.3); [RFC8693]'s "actor" framing strictly addresses only the delegation case, but this profile reuses the actor_token wire artifact for both, with classification determined by the grant (see Appendix "Why reuse actor_token in the self-acting case?"). Use on a token-exchange request remains governed by [RFC8693].

OAuth Actor Profile. [ACTOR-PROFILE] defines the structure of the act claim, the sub_profile claim, and nested actor representation. This document does not redefine those constructs; it defines how a client instance proves itself at the token endpoint and how the Authorization Server (AS) represents the validated assertion in issued access tokens (act for delegation, top-level sub for self-acting). Implementations of this document MUST also implement [ACTOR-PROFILE].

SPIFFE Client Authentication. [SPIFFE-CLIENT-AUTH] (an OAuth Working Group document) defines how a SPIFFE workload authenticates _as the OAuth client itself_ using a JWT-SVID or X.509-SVID in place of a client secret. This document operates at a different layer (actor / instance identity, not client authentication) and on different OAuth parameters and trust sources:

Layer	SPIFFE Client Auth	This document
What is authenticated	The OAuth client	An actor (instance) acting under an OAuth client
Token request parameter	client_assertion / client_assertion_type	actor_token / actor_token_type
Trust source (client metadata)	SPIFFE bundle endpoint and spiffe_id	instance_issuers
Where the SPIFFE ID surfaces	Validated against spiffe_id; not propagated	Surfaced in act.sub or top-level sub of issued access tokens

Table 1

The two specifications are orthogonal and MAY be combined: a typical combined deployment uses [SPIFFE-CLIENT-AUTH] (with a wildcard spiffe_id) to authenticate the OAuth client and this profile to surface and bind the specific instance. The same SVID MAY be presented as both client_assertion and actor_token in a single request when both profiles' audience, client-binding, and sender-constraint requirements are satisfied. This document does not require SPIFFE; instance issuers may use any subject_syntax, and the client may authenticate via any registered method. SPIFFE deployments get first-class support (Section 6.1.1, Section 8.8).

WIMSE Workload Credentials. The IETF WIMSE working group is defining specifications for workload identity ([WIMSE-CREDS], [WIMSE-ARCH]); WIMSE work is in progress. This profile's Client Instance Assertion is the OAuth-aware projection of the same workload identity model, carrying OAuth-specific bindings (client_id, aud) needed at the OAuth token endpoint. Deployments holding a WIMSE workload credential, SPIFFE JWT-SVID, Kubernetes projected service-account token, or other workload credential SHOULD use the OAuth-aware adapter pattern (Section 10) to mint a Client Instance Assertion. For sender-constraint, this profile pins the binding member of cnf to jkt ([RFC9449]) or x5t#S256 ([RFC8705]); WIMSE-defined binding mechanisms (for example, a future Workload Proof Token) can be added by a companion profile when those mechanisms reach deployment maturity.

4. Actor Token Grant Extension

This section defines a grant extension for carrying actor tokens at the OAuth token endpoint. It is independent of the client instance model defined later in this document: any profile that defines an actor_token_type, including profiles based on [ACTOR-PROFILE], can use this extension to present actor tokens on the grant types listed below. This document's client-instance-jwt actor token type (Section 7) is one such profile; its token-type-specific processing rules are defined in Section 8.

[RFC8693] provides a general-purpose way to present an actor token to the AS, but scopes actor_token and actor_token_type to the token-exchange grant. Actor identity is also useful when an AS issues tokens directly from other grants: for example, an agent may redeem an authorization code on behalf of a user, a service may request a self-acting client credentials token, a delegated actor may refresh a token while preserving actor identity, or a JWT bearer assertion may identify one principal while the request still identifies the actor performing the operation.

Defining new request parameters for each grant or actor-token profile would fragment the token endpoint surface. This extension instead reuses the existing actor_token and actor_token_type parameters as a common presentation mechanism. The extension does not define a universal actor-token validation model or access-token representation; those remain the responsibility of each actor_token_type profile.

4.1. Permitted Grant Types

[RFC8693] defines actor_token and actor_token_type only on the token-exchange grant. This document permits these parameters on the following additional token endpoint grant types:

* authorization_code ([RFC6749])

* client_credentials ([RFC6749])

* refresh_token ([RFC6749])

* urn:ietf:params:oauth:grant-type:jwt-bearer ([RFC7523])

Token-exchange ([RFC8693]) is also in scope; processing under [RFC8693] continues to apply unchanged for any actor_token_type the AS supports.

This actor token grant extension defines only where the actor_token and actor_token_type parameters may appear. Each actor_token_type profile defines how the AS validates that actor token, how it represents the actor in issued tokens, and how refresh-token requests are processed for that token type.

This document does not define behavior for the implicit grant or for the device authorization grant; specifying those is left to future work.

4.2. Other Actor Token Types

The actor token grant extension is independent of any specific actor_token_type. Any profile MAY register an actor_token_type value suitable for its actor model (for example, AI agents acting on behalf of a user, services acting under a delegation grant, or workload identities outside the OAuth client-class model). Such profiles can use this document’s actor token grant extension to present their actor tokens on the same set of grants without re-specifying the grant-level actor-token presentation interaction.

This document defines and registers urn:ietf:params:oauth:token-type:client-instance-jwt for asserting client instance identity (Section 7).

At the token endpoint, the AS dispatches by actor_token_type: tokens of type urn:ietf:params:oauth:token-type:client-instance-jwt are processed under this document's rules in Section 8, while tokens of other registered types are processed under their own specifications. An AS MAY support multiple actor_token_type values concurrently and SHOULD advertise them via actor_token_types_supported (Section 6.2).

A profile defining a new actor_token_type for use under the actor token grant extension MUST specify the validation, trust resolution, access-token representation, sender-constraint, refresh-token, and security rules for that token type. Such profiles can reference this actor token grant extension for the wire-level permission to carry actor_token and actor_token_type on the grant types listed in Section 4.1.

Some of the processing rules in Section 8 (notably Section 8.5, Section 8.6, and Section 8.7) describe behavior that is not specific to the client-instance-jwt token type. Other actor-token profiles may find them reusable as a starting point, but this document does not prescribe their reuse; each new actor_token_type profile defines its own validation, representation, and security rules.

5. Client Instance Model

A registered OAuth client commonly abstracts over many concrete runtimes (e.g., the Slack OAuth client across iOS, Android, web, and server-side; an agent platform across each running agent or session). This profile makes that class-and-instance relationship explicit so each runtime can be named, attested, and bound to access tokens individually. For agent platforms, a sub-agent spawned by an agent is represented as a nested actor via token-exchange (Section 8.6.4).

The remainder of this section covers the architecture (Section 5.1), registration models (Section 5.2), and trust delegation model (Section 5.3).

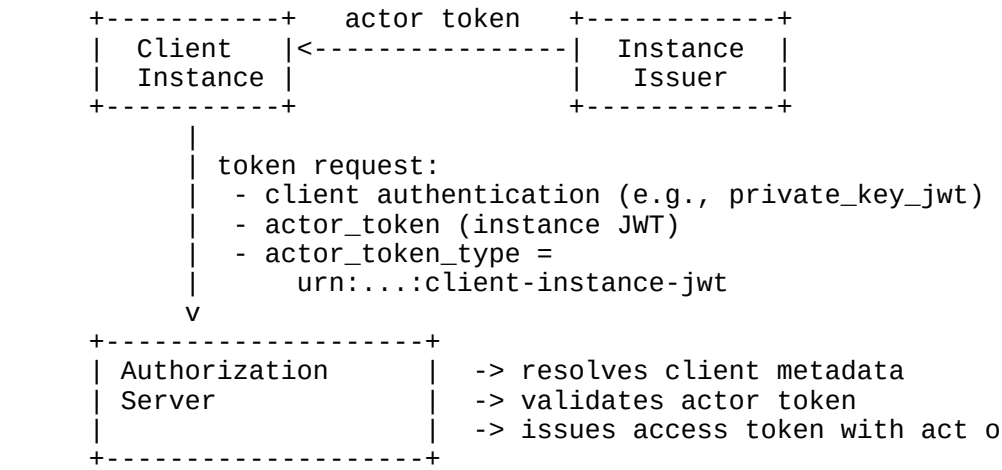
5.1. Architecture

Three roles cooperate to authenticate a client instance:

Role	Responsibility
OAuth Client	Logical OAuth client identified by a client_id (CIMD URL or RFC 7591-registered identifier). Publishes the list of trusted instance issuers in its registered client metadata.
Instance Issuer	Authenticates concrete runtime instances and issues short-lived JWT instance assertions describing them.
Authorization Server (AS)	Authenticates the client per its registered client authentication method; resolves the client metadata (via CIMD dereference or local registration storage); verifies the instance assertion against a trusted instance issuer; mints an access token whose act claim or top-level sub represents the instance.

Table 2

A high-level flow:



The client identifier (the CIMD URL or RFC 7591 client_id) is the OAuth client identifier; the instance issuer identifier is the JWT iss of the instance assertion. They are distinct trust anchors: the AS authenticates the client using its registered client authentication method (typically private_key_jwt with keys from the client's registered jwks_uri) and authenticates the instance through the instance assertion.

When the client registers token_endpoint_auth_method client_instance_jwt (Section 8.3), these two trust anchors collapse onto a single artifact: the instance assertion both authenticates the client (via the client metadata endorsement of its issuer) and identifies the instance. In that mode the request carries no separate client_assertion. See Section 8.3 for the token request and validation procedure.

5.2. Client Registration Models

This profile applies to OAuth clients regardless of how their metadata is registered with the AS. For most existing OAuth deployments, this means RFC 7591-style static registration administered by the AS operator (whether via the dynamic registration endpoint or pre-registered out of band); CIMD is the more recent option that adds public discovery and cross-organization auditability where those are required. The descriptor format and processing rules are identical in both cases.

Static registration ([RFC7591]): The client_id is opaque or AS-assigned. Metadata is registered via the dynamic registration endpoint or out of band, and stays internal to the AS. Updates take effect when the registration store is updated. This model integrates with existing RFC 7591-based registration toolchains and is the typical mode for closed or managed deployments where the trust relationship does not need to be publicly discoverable.

Client ID Metadata Document ([CIMD]): The client_id is an HTTPS URL that dereferences to a metadata document. The AS fetches the document on demand and caches it; updates take effect after the cache window expires (or upon explicit re-fetch). This model suits cross-organization deployments where the trust relationship between the OAuth client and its instance issuers must be auditable in a publicly resolvable document, and is operationally easier for cross-organization SPIFFE federation (Section 5.3.4) because the foreign organization can publicly publish its bundle endpoint and instance descriptors. Under static registration, equivalent federation requires manual coordination of registration between organizations.

The descriptor format (Section 6.1.1), the instance-assertion auth mode (Section 8.3), the SPIFFE compatibility features (Section 8.8), and the AS processing rules (Section 8.2) all apply uniformly to both models. The only deployment-time differences are how metadata updates propagate (cache TTL versus admin update) and whether the trust relationship is publicly discoverable.

5.3. Trust Delegation Model

This profile defines a three-party trust delegation between the client, the instance issuer, and the AS. The client _delegates_ attestation of its runtime instances to one or more instance issuers; the AS _relies on_ that delegation as expressed in the client's registered metadata.

5.3.1. Delegation by the OAuth Client

By listing an instance issuer in its instance_issuers metadata (Section 6.1.1), a client delegates to that issuer the authority to attest that a concrete runtime is an instance of the client. The descriptor bounds the delegation: trust_domain, subject_syntax, spiffe_id, and signing_alg_values_supported constrain what the issuer may assert and what the AS will accept (Section 6.1.1). The issuer's per-client minting obligations are in Section 7.1; client-side guidance on choosing issuers is in Section 12.1.

The per-client minting requirement (an issuer mints assertions naming a given client_id only for runtimes authorized as instances of that client) prevents cross-client instance impersonation when the same instance issuer is listed by multiple OAuth clients.

Because the instance_issuers listing endorses the issuer to mint tokens naming this client_id, an instance assertion signed by such an issuer is itself attributable to the client. This makes the assertion usable as the client credential under Section 8.3.

5.3.2. Authority of the Authorization Server

The AS treats the registered instance_issuers list as authoritative: it derives its trust in an instance assertion solely from the descriptor whose issuer member matches the instance assertion's iss claim. This document does not define out-of-band or AS-side configuration of additional instance issuers for a client_id; deployments requiring such configuration MUST do so via a separate specification.

5.3.3. Trust Lifecycle

The trust relationship between client and instance issuer is mutable. When the client's metadata changes (for example, an instance issuer is removed, its jwks_uri or jwks rotates, its trust_domain or spiffe_id is replaced, or its signing_alg_values_supported narrows), updates take effect according to the registration model: for CIMD, the AS applies the same freshness and re-fetch rules it applies to other CIMD-published trust material such as jwks_uri (see [CIMD]); for static registration, updates take effect when the AS's registration store is updated and re-read.

While the AS may continue to honor a stale descriptor within the propagation window, this profile imposes no additional revocation requirement on previously issued access tokens. AS treatment of access tokens whose validated instance identity is no longer endorsed after the update is governed by Section 12.4; sizing the resulting trust-withdrawal latency is in Section 12.2.

5.3.4. Cross-Organization Federation

A client MAY list instance issuers operated by a different organization, including cases where SPIFFE trust domains differ. The per-client minting rule (Section 5.3.1) applies unchanged. For SPIFFE cross-trust-domain deployments, the descriptor's spiffe_bundle_endpoint MUST be operated by the foreign organization or its delegate.

6. Metadata and Discovery

This section defines client metadata used to delegate instance attestation and authorization server metadata used by clients to discover support.

6.1. Client Metadata Extensions

This document defines client metadata parameters describing the trust relationship between a client and the instance issuers that authenticate its runtime instances. These parameters are registered in the OAuth Dynamic Client Registration Metadata registry (Section 13.2) and apply to clients regardless of how their metadata reaches the AS:

* For clients identified by a Client ID Metadata Document [CIMD], these parameters appear in the CIMD document and the AS resolves them by dereferencing the CIMD URL.

* For clients registered via OAuth Dynamic Client Registration [RFC7591] (or admin-registered with RFC 7591-shaped metadata), these parameters appear in the registered client metadata stored by the AS.

The descriptor format and processing rules are identical in both cases. Section 5.2 discusses the trade-offs between the two registration models.

6.1.1. instance_issuers

OPTIONAL. A non-empty JSON array of _instance issuer descriptor_ objects. Each descriptor declares an issuer that the client trusts to authenticate its instances. If this parameter is absent, or is present as an empty array, the AS MUST NOT accept instance assertions of type urn:ietf:params:oauth:token-type:client-instance-jwt for this client; the AS SHOULD treat an empty array as a metadata error and log it for the client operator.

An instance issuer descriptor has the following members:

issuer (REQUIRED): A StringOrURI [RFC7519] identifying the instance issuer. This value MUST exactly match the iss claim of accepted instance assertions and MUST be unique within the instance_issuers array.

For raw JWT-SVID compatibility (Section 8.8), this value is the SPIFFE JWT-SVID issuer for the trust domain. For re-minted Client Instance Assertions, this value identifies the OAuth-aware adapter or instance issuer that signed the assertion; trust_domain and spiffe_id then bound the SPIFFE subject space that issuer is allowed to assert.

A descriptor MUST contain exactly one of jwks_uri, jwks, and spiffe_bundle_endpoint. If two or more are present, or all are absent, the AS MUST reject the descriptor as invalid client metadata.

jwks_uri: An HTTPS URL of a JWK Set [RFC7517] containing the public keys used to verify signatures of instance assertions issued by this issuer.

jwks: An inline JWK Set serving the same purpose as jwks_uri.

spiffe_bundle_endpoint: An HTTPS URL of a SPIFFE trust bundle endpoint [SPIFFE] from which the AS resolves verification keys for instance assertions issued by this issuer. When present, subject_syntax MUST be "spiffe".

This descriptor field is intended for JWT-SVID validation and for other assertions signed with keys distributed in the SPIFFE bundle for the relevant trust domain. OAuth-aware adapters that sign re-minted Client Instance Assertions with separate OAuth signing keys use jwks_uri or jwks instead.

Bundle endpoint format and resolution rules are governed by SPIFFE; see [SPIFFE-CLIENT-AUTH] for the analogous use in client authentication.

signing_alg_values_supported (OPTIONAL): A JSON array of JSON Web Signature (JWS) [RFC7515] alg values the AS accepts for instance assertions issued by this issuer. If present, the AS MUST reject instance assertions whose alg is not listed. Issuers SHOULD publish only algorithms they actually use.

subject_syntax (OPTIONAL): A short identifier indicating the syntactic profile of the sub claim used by this issuer. This document defines two values: "uri" (default, arbitrary StringOrURI) and "spiffe" (a SPIFFE ID [SPIFFE]; see also [SPIFFE-CLIENT-AUTH] for the related SPIFFE-based client authentication profile). An AS that does not understand the value MUST reject instance assertions for that descriptor with invalid_grant.

trust_domain (OPTIONAL): When subject_syntax is "spiffe", a SPIFFE trust domain that the sub claim MUST belong to. The AS MUST reject any instance assertion whose sub does not lie within this trust domain.

A SPIFFE ID lies within a trust domain only when it parses as a valid SPIFFE ID whose trust-domain component exactly equals trust_domain; ASes MUST NOT use case folding, Unicode normalization, or percent-decoding to make a non-matching trust domain match.

trust_domain is meaningful only when subject_syntax is "spiffe"; an AS MUST ignore trust_domain in descriptors whose subject_syntax is any other value. A descriptor’s trust_domain is independent of any SPIFFE trust domain associated with the client itself under [SPIFFE-CLIENT-AUTH]; the two MAY differ.

spiffe_id (OPTIONAL): When subject_syntax is "spiffe", a SPIFFE ID that further bounds which workloads this issuer may attest as instances of this client. The value is a SPIFFE ID, optionally with a trailing "/*" wildcard.

Without "/_", the instance assertion's sub MUST equal this value exactly; with "/_", matching follows the spiffe_id matching rule of [SPIFFE-CLIENT-AUTH]. If both spiffe_id and trust_domain are present, the trust domain in spiffe_id MUST equal trust_domain.

When present, this member structurally binds a workload subtree to this client; see Section 8.8.1. If subject_syntax is "spiffe" and spiffe_id is absent, trust_domain MUST be present and the descriptor delegates the entire trust domain to this instance issuer.

Clients SHOULD include spiffe_id; omitting it is appropriate only when every workload in the SPIFFE trust domain is authorized to act as an instance of the client.

Example client metadata document with a SPIFFE instance issuer:

```
{
  "client_id": "https://openai.example.com/codex",
  "jwks_uri": "https://openai.example.com/codex/jwks.json",
  "token_endpoint_auth_method": "private_key_jwt",
  "instance_issuers": [
    {
      "issuer": "https://workload.openai.example.com",
      "jwks_uri": "https://workload.openai.example.com/jwks.json",
      "subject_syntax": "spiffe",
      "trust_domain": "openai.example.com",
      "spiffe_id": "spiffe://openai.example.com/codex/*",
      "signing_alg_values_supported": ["ES256"]
    }
  ]
}
```

6.2. Authorization Server Metadata

This document defines the following AS metadata parameters for [RFC8414] (see Section 13.4):

actor_token_types_supported: A JSON array of actor_token_type values

supported by the AS at the token endpoint. An AS implementing this profile MUST publish this parameter and include urn:ietf:params:oauth:token-type:client-instance-jwt in it. This is the only AS-side discovery signal for support of this profile; clients use it to decide whether to assemble token requests carrying an instance assertion. This signal is intentionally coarse: it does not describe grant-specific enablement, raw JWT-SVID support, accepted sender-constraint methods, refresh-token behavior, or client-specific registration policy. Clients may still need registration-time or deployment agreement with the AS for those details.

Other values MAY appear and are processed under their own specifications; their trust resolution is not via instance_issuers.

In addition, an AS that supports Section 8.3 MUST advertise client_instance_jwt in token_endpoint_auth_methods_supported ([RFC8414]).

Example AS metadata document (abridged):

```
{
  "issuer": "https://as.example.com",
  "token_endpoint": "https://as.example.com/token",
  "token_endpoint_auth_methods_supported": [
    "private_key_jwt",
    "client_instance_jwt"
  ],
  "actor_token_types_supported": [
    "urn:ietf:params:oauth:token-type:client-instance-jwt"
  ],
  "dpop_signing_alg_values_supported": ["ES256", "RS256"]
}
```

7. Client Instance Assertion Format

This section defines client-instance-jwt, a specific actor_token_type registered under this document for asserting client instance identity. Tokens of this type are presented under the actor token grant extension defined in Section 4; their AS-side processing is specified in Section 8.

A _Client Instance Assertion_ is a JWT [RFC7519] that asserts the identity of a client instance. It is presented as the actor_token parameter ([RFC8693]) with actor_token_type set to urn:ietf:params:oauth:token-type:client-instance-jwt (see Section 13.1). The assertion serves two purposes: it authenticates the runtime instance (workload identity), and it asserts that the

instance is a member of the named OAuth client. This document defines a single JWT carrying both, signed by the instance issuer. This matches the prevailing pattern in workload identity systems, which already issue audience-scoped, signed assertions of runtime identity (e.g., JWT-SVIDs in [SPIFFE]).

7.1. Issuer Obligations

The instance issuer is the trust authority for the assertion and MUST, before minting an instance assertion under this profile:

* Authenticate the runtime instance (e.g., via attestation, platform-level identity, or possession of an instance key); and

* Verify, under issuer-side policy, that the runtime is permitted to claim the client_id named in the token. This typically means the runtime is operationally part of the client's deployment. An instance issuer MUST refuse to mint an instance assertion whose client_id claim names a client for which the runtime has not been authorized, by issuer-side policy, as a member.

An instance issuer MUST NOT reassign an active or audit-relevant sub value to a different runtime. Issuers SHOULD use stable, non-reassigned subjects, or include sufficient generation or session uniqueness in sub to distinguish runtime incarnations. If subject reassignment is unavoidable, the client, issuer, and AS audit logs need enough lifecycle metadata to distinguish the old and new runtimes.

How the issuer internally authenticates the runtime is out of scope. Common deployment patterns (adapter, raw JWT-SVID compatibility, X.509-SVID binding) are described in Section 10.

7.2. JWT Claims

The following claims are defined for client instance assertions.

iss (REQUIRED): The instance issuer identifier. MUST exactly match an issuer member of an instance_issuers descriptor in the client's registered metadata.

sub (REQUIRED): The identifier of the client instance, in the syntax declared by the descriptor's subject_syntax (default: arbitrary StringOrURI). Sub-uniqueness considerations for self-acting tokens are addressed in Section 8.6.3.

aud (REQUIRED): The intended audience, identifying the AS. The AS

validates aud per [RFC7523] Section 3, accepting its own issuer identifier or token endpoint URL; if multiple values are present, at least one MUST match.

Each AS SHOULD specify a single canonical aud format (typically its issuer identifier) and document it; instance issuers SHOULD use that canonical form. Where instance assertions are scoped per AS, instance issuers SHOULD mint an AS-specific instance assertion rather than a multi-aud JWT, to limit the replay surface.

client_id (REQUIRED unless the SPIFFE compatibility conditions of Section 8.8.1 are met): The client_id of the client to which this instance belongs. This claim uses the JSON Web Token client_id claim registered by [RFC9068] Section 2.2 (which itself defers to [RFC8693] Section 4.3 for the underlying definition). Note that RFC 9068 defines client_id as the OAuth client to which a JWT access token was issued; in this profile, the claim instead names the OAuth client to which the asserted instance belongs.

The claim binds the actor token to a specific client and is not part of the actor's identity (per [ACTOR-PROFILE], client_id identifies an OAuth client, not an actor). When present, the AS MUST reject the token if this value does not exactly equal the client_id of the authenticated client.

When omitted under Section 8.8.1, the binding is established structurally by the matched descriptor's SPIFFE scope (spiffe_id when present, otherwise trust_domain) rather than by a JWT claim, and a SPIFFE JWT-SVID may be presented as the actor_token directly without re-minting.

exp (REQUIRED): Expiration time. Issuers SHOULD set short lifetimes (e.g., five minutes or less); see Section 12.5.

iat (REQUIRED): Issued-at time.

jti (REQUIRED): A unique identifier used for replay prevention; see Section 12.5.

sub_profile (RECOMMENDED): One or more OAuth Entity Profile names [ENTITY-PROFILES] classifying the actor. [ENTITY-PROFILES] defines this claim as OPTIONAL; this profile elevates it to RECOMMENDED so resource servers can apply actor-class-aware policy without bespoke configuration.

Its syntax (a space-delimited string of profile names) is the one defined by [ACTOR-PROFILE]. This document registers the value client_instance (Section 13.6). Issuers MAY include additional

values registered with the "Actor Profile" usage location in the OAuth Entity Profiles registry, or privately defined collision-resistant values, per [ACTOR-PROFILE].

cnf (REQUIRED unless the token is a raw JWT-SVID accepted under Section 8.8.1): A confirmation claim [RFC7800] carrying a key bound to this instance. The cnf value MUST contain exactly one of jkt (a JWK SHA-256 thumbprint per [RFC9449] Section 3.1) or x5t#S256 (an X.509 certificate SHA-256 thumbprint per [RFC8705] Section 3.1) as the binding member; other confirmation methods registered under [RFC7800] MAY appear alongside but are not the binding and do not change this profile's sender-constraint verification requirement.

The instance issuer MUST mint cnf from a key the named runtime instance demonstrably possesses (e.g., an instance-attested key, a per-instance workload key, or a Demonstration of Proof-of-Possession (DPoP, [RFC9449]) public key presented to the issuer at attestation time). Binding rules and AS verification are defined in Section 8.5.

Raw JWT-SVID compatibility is the only exception to this claim requirement, because the AS validates the SVID without re-minting; see Section 8.8.1 and Section 8.5.

nbf (OPTIONAL): Not-before time. If present, the AS MUST reject the token before this time.

When validating exp, nbf, and iat, ASes SHOULD permit a small clock skew tolerance, typically no more than 60 seconds, applied symmetrically. This bound is consistent with the short-lifetime recommendation in Section 12.5 and prevents brittle inter-clock failures across deployments.

A client instance assertion MUST NOT contain an act claim. The instance assertion is a direct identity assertion of a single party (the instance); per [ACTOR-PROFILE], an actor_token that carries an act claim represents a delegation chain rather than a direct identity, and the AS MUST reject such a token with invalid_grant (Section 8.6.4, Section 8.9).

Additional claims MAY be present and MUST be ignored if not understood, except where this document or [ACTOR-PROFILE] specifies processing rules. Future profiles requiring AS understanding of a new claim SHOULD use the JWS crit header parameter ([RFC7515] Section 4.1.11) to mark it must-understand; ASes MUST reject assertions whose crit header is malformed or includes claims they do not implement, per [RFC7515] Section 4.1.11.

7.3. Signing and JOSE Header

A Client Instance Assertion MUST be signed using an asymmetric JWS [RFC7515] algorithm; none and symmetric (HMAC-based) algorithms (HS256, HS384, HS512) MUST NOT be used and ASes MUST reject assertions signed with them. The descriptor's signing_alg_values_supported (Section 6.1.1), when present, MUST contain only asymmetric algorithm identifiers. Implementations SHOULD follow the JWT BCP guidance in [RFC8725]. For interoperability, ASes SHOULD support at least RS256 and ES256. Issuers SHOULD include a kid in the JWS protected header; ASes SHOULD use kid for key selection.

Issuers minting a Client Instance Assertion under this profile MUST set the JWS typ (type) protected header parameter to client-instance+jwt per [RFC8725] Section 3.11, and ASes MUST reject such assertions whose typ is anything else. Explicit typing prevents JWT confusion attacks where a token of a different type (for example, a WIMSE workload identity credential [WIMSE-CREDS], a JWT-SVID outside the SPIFFE compatibility mode, or an OAuth JWT access token [RFC9068]) is mistaken for a Client Instance Assertion.

The only exception is the SPIFFE compatibility mode in Section 8.8.1, where a raw JWT-SVID is intentionally presented without re-minting. In that mode, the AS MUST validate the token as a JWT-SVID according to [SPIFFE-CLIENT-AUTH] and Section 8.8.2, and MUST NOT require the JWS typ header to be client-instance+jwt.

Verification keys are obtained from the descriptor's jwks_uri, jwks, or spiffe_bundle_endpoint for the issuer that matches the iss claim; the AS MUST verify alg against signing_alg_values_supported when present.

7.4. Example Assertion

A decoded re-minted Client Instance Assertion (JWS protected header and JWT payload):

```
{ "alg": "ES256", "kid": "4vC8agycHu6rnkE...", "typ": "client-instance+jwt" }
```

```
{
  "iss":         "https://workload.openai.example.com",
  "sub":         "spiffe://openai.example.com/codex/session-abc",
  "aud":         "https://as.example.com",
  "client_id":   "https://openai.example.com/codex",
  "sub_profile": "client_instance",
  "iat":         1770000000,
  "exp":         1770000300,
  "jti":         "1a2b3c4d-5e6f",
  "cnf":         { "jkt": "0ZcOCORZNYy...iguA4I" }
}
```

8. Token Endpoint Processing

This section specifies AS-side processing for token requests carrying an actor_token of type urn:ietf:params:oauth:token-type:client-instance-jwt. It builds on the actor token grant extension in Section 4 by defining the validation, authorization-time consistency, sender- constraint, representation, refresh, client authentication, SPIFFE compatibility, and error rules specific to the client-instance-jwt token type.

This profile applies whether the AS issues JWT access tokens ([RFC9068]) or opaque (reference) access tokens. The representation rules in Section 8.6 describe the _claims_ an issued access token carries (act, sub, client_id, cnf, sub_profile); for JWT access tokens these appear directly in the token payload, while for opaque access tokens they MUST be reflected in introspection responses ([RFC7662], Section 12.4.3). The sender-constraint binding in Section 8.5 applies to both formats; the binding key is verified at presentation regardless of whether the access token is self-contained or requires introspection.

8.1. Token Request

A client presents a client instance assertion at the token endpoint by adding the actor_token and actor_token_type parameters defined by [RFC8693] to a token request of any grant type listed in Section 4.

The following example shows a client credentials grant carrying a client instance assertion. The client authenticates with private_key_jwt; line breaks are for readability:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: <DPoP proof bound to the instance's key>

grant_type=client_credentials
&scope=repo.write
&client_id=https%3A%2F%2Fopenai.example.com%2Fcodex
&client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJFUzI1NiIsImtpZCI6...
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

8.2. Authorization Server Processing

When evaluating a token request for this profile, an AS implementing this document MUST perform the following checks and steps in addition to grant-type-specific processing.

Before the steps below, the AS MUST reject the request with invalid_request if any of the following pre-conditions hold:

* exactly one of actor_token and actor_token_type is present;

* actor_token_type is urn:ietf:params:oauth:token-type:client-instance-jwt but actor_token is absent;

* actor_token_type is urn:ietf:params:oauth:token-type:client-instance-jwt and actor_token is present but is not a syntactically valid JWT.

1. *Authenticate the client.* Authenticate the client using its registered token_endpoint_auth_method per [RFC6749] and, if applicable, [RFC7523]. The client_id identifies the OAuth client. When the registered method is client_instance_jwt, follow Section 8.3 instead of presenting a separate client-controlled credential.

2. *Match the token type.* If actor_token_type is not urn:ietf:params:oauth:token-type:client-instance-jwt, processing under this document does not apply. If the AS does not support the supplied actor_token_type for the requested grant, it MUST reject the request with unsupported_token_type (Section 8.9). Other registered actor_token_type values MAY be processed under their own specifications (Section 4.2).

3. *Resolve client metadata.* Retrieve the client metadata for the authenticated client_id. For clients identified by [CIMD], dereference the CIMD document subject to its caching rules. For clients registered via [RFC7591] or pre-registered with the AS, read the stored metadata. The remaining steps operate on the resolved metadata regardless of source.

4. *Locate the instance issuer descriptor.* Parse the actor_token as a JWT and read its iss claim. Find the descriptor in instance_issuers whose issuer member exactly equals iss. If no descriptor is found, or instance_issuers is absent, reject the request with invalid_grant (Section 8.9).

5. *Verify the signature.* Using the descriptor's jwks_uri, jwks, or spiffe_bundle_endpoint, verify the JWS signature per [RFC7515], Section 7.3, and, when applicable, Section 8.8.2.

6. *Validate JWT claims.* Validate iss, sub, aud, exp, iat, nbf, and jti per Section 7.2 and [RFC7523] Section 3, subject to the raw JWT-SVID exceptions in Section 8.8.1. Enforce subject_syntax, trust_domain, spiffe_id, and signing_alg_values_supported when present in the descriptor. If subject_syntax is "spiffe" and spiffe_id is absent, require trust_domain and treat the descriptor as delegating the whole trust domain. Validate the JWS typ per Section 7.3 and reject unrecognized crit header parameters per Section 7.2.

7. *Verify client_id binding.* If the instance assertion contains a client_id claim, it MUST exactly equal the authenticated client_id; reject with invalid_grant otherwise. If the instance assertion has no client_id claim, the AS MUST verify that the matched descriptor satisfies the SPIFFE compatibility conditions (Section 8.8.1); if not, reject with invalid_grant. When the descriptor satisfies those conditions, the AS MUST verify that the actor_token's sub falls under the descriptor's SPIFFE scope: spiffe_id when present, otherwise the descriptor's trust_domain; if not, reject with invalid_grant. After client_id binding succeeds, apply replay checking per Section 12.5; reject with invalid_grant if a previously seen tuple is found. For raw JWT-SVIDs, this replay check applies only when jti is present. The replay check follows client_id binding so that an attacker cannot burn a legitimate client's jti by presenting the assertion under a mismatched client_id.

8. *Enforce delegation policy.* Apply the AS's local maximum delegation depth per [ACTOR-PROFILE].

9. *Check authorization-time consistency.* For grants that originate from a prior authorization step (notably authorization_code), apply the rules of Section 8.4.

10. *Bind the instance.* If issuance succeeds, represent the instance in the access token per Section 8.6, applying Section 8.5 for token binding. Reflect any prior actor chain present in input tokens by nesting per [ACTOR-PROFILE]; chain merging rules are given in Section 8.6.4.

If validation succeeds, the AS issues an access token (and optionally a refresh token) per the requested grant.

8.3. Client Authentication via Instance Assertion

A client MAY register the token_endpoint_auth_method value client_instance_jwt in its registered metadata (whether published as a CIMD document or stored at the AS) to indicate that the AS authenticates the client implicitly from a presented instance assertion, without requiring a separate client_assertion or other credential controlled by the client itself.

This mode is the natural choice for workload-only deployments (for example, agentic services, autoscaled microservices, or ephemeral functions) where there is no human user authorizing operations and the team operating the runtime is also the natural owner of the workload identity provider. For these deployments, requiring a separate client-level credential typically means provisioning a private key into every pod alongside the instance-attested actor_token, which (per Section 12.7) does not meaningfully improve defense against runtime compromise. The mode is also appropriate where the client identifier is a logical CIMD URL with client-key custody centralized away from the runtime, or where the workload identity provider trusted to attest instances is the only authority the client wishes to publish.

The trust chain to the client is preserved: the client’s listing of the instance issuer in its registered metadata is itself the endorsement, and a token signed by such an issuer naming this client_id is attributable to the client. When token_endpoint_auth_method is client_instance_jwt, every accepted instance issuer for the client is also a client authentication trust root. Clients MUST NOT enable this method unless each listed issuer is authorized for that role.

8.3.1. Request Format

A request using this auth method MUST include the client_id form parameter, the actor_token, and actor_token_type. It MUST NOT carry client_assertion or any other client authentication credential. The client_id form parameter is required so the AS can resolve client metadata before validating the assertion; the assertion's client_id claim (Section 7.2) is then matched against this value. Example, using the client_credentials grant:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&scope=repo.write
&client_id=https%3A%2F%2Fopenai.example.com%2Fcodex
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

8.3.2. Validation Procedure

When the registered token_endpoint_auth_method for the client_id is client_instance_jwt, the pre-conditions of Section 8.2 (rejecting mismatched or malformed actor_token/actor_token_type) still apply, and the AS replaces step 1 of Section 8.2 ("Authenticate the client") with the following procedure:

1. Resolve client metadata for client_id (per the registration model: dereference the CIMD URL or read stored registration data).

2. Validate the presented actor_token using the token-type check, instance issuer descriptor lookup, signature verification, and JWT claim validation rules in Section 8.2.

3. Verify that the actor_token's client_id claim exactly equals the request's client_id parameter.

4. Apply the replay check in Section 12.5 after client_id binding succeeds.

5. Verify proof-of-possession of the actor_token at presentation per Section 8.5. In this mode the actor_token serves as the sole client authentication credential, so the bearer-replay considerations in Section 12.5 apply with no fallback credential; ASes MUST reject requests in this mode whose actor_token lacks a cnf claim, and MUST verify possession of the cnf key.

6. Reject the request with invalid_client if any of the above fails.

7. Treat the client as authenticated. The validated actor_token also satisfies the actor_token requirement of this profile and is used for instance representation per Section 8.6.

Because the actor_token is the client authentication credential in this mode, validation failures that would otherwise be returned as invalid_grant under Section 8.2 are returned as invalid_client.

The actor_token's aud claim serves both purposes (the [RFC7523] client-assertion audience and this profile's actor-token audience). A single value identifying the AS satisfies both.

The SPIFFE client_id claim omission mode (Section 8.8.1) does not apply to client_instance_jwt client authentication. Because the same JWT is the sole client authentication credential, the assertion MUST contain the client_id claim and the AS MUST verify it exactly as described above.

After this procedure completes, processing continues with step 8 of Section 8.2 ("Enforce delegation policy") and onward, reusing the validated instance assertion. The AS MUST NOT re-apply the token-type check, descriptor lookup, signature verification, claim validation, or client_id binding checks to the same actor_token in a way that would cause the request to fail replay detection for its own presentation.

8.4. Authorization-Time Consistency

When a token request is made under the authorization_code grant ([RFC6749] Section 4.1), the user has authorized the _client_ identified by client_id, not any specific client instance. The AS MUST ensure that the instance introduced at the token endpoint is consistent with that authorization:

* The client_id authenticated at the token endpoint MUST match the client_id that received the authorization_code ([RFC6749] Section 4.1.3). Combined with the client_id-binding requirement in Section 8.2, this prevents an instance assertion from another client from being attached to a code.

* The AS MUST NOT permit the instance identity to bypass standard authorization-code controls (single-use redemption, redirect URI matching, and any code challenge bound to the original authorization request).

* If the AS has any authorization-time policy that depends on the instance (for example, a per-instance allow-list), the AS MUST evaluate that policy against the instance assertion presented at /token and reject inconsistent requests with invalid_grant.

When the issued access token is to be DPoP-bound, clients SHOULD include the dpop_jkt parameter ([RFC9449] Section 10) on the authorization request, naming the same public key whose thumbprint will appear in the instance assertion's cnf.jkt at the token endpoint. When dpop_jkt is present, the AS binds the authorization code to that key per [RFC9449], and at the token endpoint MUST verify that the DPoP proof, the authorization code's dpop_jkt, and the instance assertion's cnf.jkt all reference the same key. This provides cryptographic continuity from /authorize to /token bound to a specific instance: only the instance that holds the named DPoP private key can redeem the resulting code, even if the code is intercepted or transferred to another runtime under the same client.

If dpop_jkt is absent, DPoP still sender-constrains the issued access token at the token endpoint, but this profile does not provide cryptographic continuity between the authorization endpoint and the token endpoint for that authorization code.

For Mutual-TLS-bound access tokens ([RFC8705]), authorization-code continuity is deployment-specific. If the AS binds the authorization request or authorization code to a client certificate seen at the authorization endpoint, then at the token endpoint the AS MUST verify that the certificate used to redeem the code and the instance assertion's cnf.x5t#S256 match the certificate bound at authorization time. Otherwise, mTLS sender-constraint is established at the token endpoint and does not by itself provide authorization- to-token endpoint continuity.

User consent under this profile applies to the client as a whole; consent thereby covers all instances attested by listed instance issuers. The key-bound continuity above adds cryptographic guarantees about _which_ instance redeems a code, but does not by itself constitute per-instance consent.

An AS MAY require per-instance or per-key authorization policy when the authorization request includes a sender-constraining key such as dpop_jkt. Such policy is deployment-specific: dpop_jkt identifies a key, not an instance, unless the AS has an authorization-time mapping

from that key to an instance identity. In those deployments, the AS can require user or administrator approval for the specific instance or key and then verify at the token endpoint that the DPoP proof, authorization code binding, and instance assertion cnf.jkt all reference the approved key.

ASes that record consent SHOULD record the descriptor scope under which consent was granted (in particular, the descriptor's issuer and trust_domain), and MAY refuse access tokens for the same client issued under a different descriptor scope than the one consented. This matters for clients deployed across multiple trust domains (for example, "production" vs. "staging" SPIFFE trust domains, or distinct PaaS environments) where the user's consent to one is not necessarily consent to another.

This document does not define a general authorization endpoint mechanism for presenting instance identity. Deployments requiring standardized per-instance consent without an authorization-time key mapping need a separate extension.

8.5. Sender-Constrained Access Tokens

When the AS issues an access token under this profile, whether the instance is represented in act (delegation case; Section 8.6.2) or in sub (self-acting case; Section 8.6.3), the AS MUST issue a sender-constrained access token bound to a key the instance possesses. Established mechanisms include DPoP [RFC9449] and Mutual-TLS-bound access tokens [RFC8705].

The AS MUST NOT issue a bearer access token under this profile. Sender-constraint is a structural prerequisite, not a preference: per-instance non-repudiation depends on binding the access token to a key the validated instance possesses. Deployments adopting this profile therefore require AS and RS support for DPoP ([RFC9449]), Mutual-TLS-bound access tokens ([RFC8705]), or both, and SHOULD verify implementation support before committing. Adoption guidance for DPoP or mTLS rollout is outside the scope of this document; deployments unable to deploy either mechanism within their adoption timeline should defer adopting this profile.

If the instance assertion includes a cnf claim (Section 7.2), the AS MUST:

* bind the issued access token to the same key by setting the access token's top-level cnf to the instance assertion's cnf value;

* verify possession of the cnf key at the token endpoint, matching the binding member used in cnf per [RFC7800]. For cnf.jkt, the JWK thumbprint of the DPoP proof's public key [RFC9449] MUST equal cnf.jkt. For cnf.x5t#S256, the certificate authenticated at the TLS layer [RFC8705] MUST match cnf.x5t#S256. Other confirmation methods present in cnf are not binding members for this profile and MAY be ignored unless local policy or their defining specifications require additional processing;

* reject the request with invalid_request if verification fails.

This protects the instance assertion from bearer-style replay within its validity window (Section 12.5); without it, the instance assertion would be a bearer credential whose replay is bounded only by exp and the jti cache.

The binding key MUST be specific to the validated client instance. A credential shared by the client as a whole, such as the client-level mTLS certificate authenticated under [RFC8705], the client's private_key_jwt key, or any other client-controlled key not provisioned per-instance, is not sufficient.

A re-minted Client Instance Assertion MUST contain cnf so that the binding key is supplied by the same authority that named the instance. The only profile-defined case where cnf can be absent is raw-JWT-SVID compatibility, where the AS establishes an instance-specific binding through a channel independent of the SVID; rules are in Section 8.8.3.

Deployments combining client-level Mutual-TLS-bound client authentication ([RFC8705]) with this profile MUST establish instance binding through a separate, instance-specific key. The typical configuration uses the client's mTLS certificate at the TLS layer for client authentication and a cnf.jkt in the instance assertion paired with DPoP [RFC9449] at the token endpoint for instance binding. Per-instance mTLS certificates issued by the instance issuer (or otherwise bound to instance attestation) are an alternative; in that case the same TLS certificate satisfies both client authentication and instance binding only if the AS treats it as belonging to the instance for binding purposes.

8.6. Access Token Representation

A client instance may be acting on behalf of another principal (_delegation case_; e.g., a user authorized the request through an authorization_code grant) or acting as itself with no other principal involved (_self-acting case_; e.g., a client_credentials grant). The AS MUST classify each request as delegation or self-acting before populating the issued access token's claims. In both cases the access token's client_id remains the client, the access token MUST be sender-constrained per Section 8.5, and any upstream actor chain MUST be preserved by nesting per [ACTOR-PROFILE] (merge rules in Section 8.6.4).

8.6.1. Classification

The AS classifies the request based on whether the grant produces a principal distinct from the client instance presenting the actor_token:

Grant	Principal	Classification
authorization_code ([RFC6749])	the user who authorized the code	delegation
client_credentials ([RFC6749])	none	self-acting
refresh_token ([RFC6749])	inherited from the original grant	inherited
jwt-bearer ([RFC7523])	the assertion's sub	delegation
token-exchange ([RFC8693])	the subject_token's subject	delegation

Table 3

The jwt-bearer and token-exchange rows always classify as delegation under this profile. [RFC7523] requires a JWT-bearer assertion that identifies a principal, and [RFC8693] Section 2.1 requires a subject_token; in both cases another party is present and named, so the issued access token's sub is that party and the actor appears in act. ASes MUST NOT classify these grants as self-acting based on heuristic matching of subject identifiers; see Section 12.8. This rule applies even when the subject_token was itself a self-acting

access token whose sub named the same instance now presenting the actor_token (e.g., a client-credentials token from an upstream AS exchanged at a downstream AS): the resulting access token has sub and act.sub naming the same instance. This is benign chain self-reference and is not an error; the AS MUST NOT collapse the two into a self- acting representation.

When neither delegation nor self-acting cleanly applies (for example, custom or experimental grants), the AS MUST refuse to issue the access token rather than guess; reject with invalid_grant (Section 8.9).

8.6.2. Delegation Case

When the request is classified as delegation, the AS MUST populate the issued access token's act claim per [ACTOR-PROFILE] from the validated client instance assertion:

* act.iss = actor_token's iss

* act.sub = actor_token's sub

* act.sub_profile = actor_token's sub_profile (if present); the value client_instance SHOULD be included.

* act.cnf = actor_token's cnf, if present.

The access token's sub MUST be the principal identified by the grant (e.g., the authenticated user). Sender-constraint binding (top-level cnf and PoP verification) is governed by Section 8.5. Note that the instance assertion's aud, client_id, exp, iat, and jti are validated and consumed by the AS but do not appear in the issued access token (client_id appears at the top level as a property of the token, not as an actor claim).

For a worked example see Appendix "Authorization Code with User Delegation"; for a nested actor chain (token-exchange whose subject_token already carries an act chain), see Appendix "Token Exchange with Prior Delegation Chain (Agent Spawns Sub-Agent)".

8.6.3. Self-Acting Case

When the request is classified as self-acting, the instance is the principal and there is no other party on whose behalf it acts. The AS MUST populate the issued access token from the validated instance assertion:

* sub = actor_token’s sub (optionally with AS-applied namespacing, see below)

* sub_profile = actor_token’s sub_profile (if present); the value client_instance SHOULD be included

* cnf is set per Section 8.5

* act MUST be omitted

The instance issuer’s identifier (the assertion’s iss) is not represented as a standard access-token claim in the self-acting case; trust in the issuer is structural via the descriptor (Section 6.1.1). The AS MUST nevertheless retain the validated issuer with its token state when needed for revocation, introspection, audit, or issuer-aware resource-server policy. For JWT access tokens consumed without introspection, if resource servers need issuer context and the client lists multiple issuers with potentially colliding subject spaces, the AS SHOULD either apply AS-scoped namespacing to sub as described below or expose issuer context using a deployment-specific claim understood by the resource server. For raw JWT-SVIDs that do not carry sub_profile, the AS SHOULD set the access token’s sub_profile to client_instance after successful validation, unless local policy intentionally suppresses that signal.

A client that lists multiple instance issuers MUST ensure those issuers’ sub spaces do not collide (for example, by using disjoint naming conventions, prefixes, or a SPIFFE trust-domain split); when the client cannot guarantee disjointness, the AS SHOULD apply namespacing (e.g., prefixing sub with the descriptor’s issuer such as <issuer>#<sub>) to prevent a compromised issuer from spoofing another’s sub. Such namespacing is a deployment-side choice and does not affect the wire format of the actor_token. AS-applied namespacing produces an AS-scoped subject identifier; resource-server policy and audit tooling need to treat it as AS-issued rather than issuer-native.

For a worked example see Appendix "Client Credentials (Self-Acting)".

8.6.4. Actor Chain Merging

The AS constructs the issued access token’s act chain per [ACTOR-PROFILE]’s Delegation Chain Validation and Construction algorithm: the validated client instance assertion is the new outermost actor, and any subject_token act chain (only applicable to token-exchange) is preserved verbatim under it. Depth limits and rejection on overflow follow [ACTOR-PROFILE].

In the self-acting case (Section 8.6.3) the act claim is omitted.

8.7. Refresh Tokens

When an access token is refreshed ([RFC6749] Section 6), the AS reuses the classification (Section 8.6.1) of the original grant to shape the refreshed access token; the original classification is _inherited_ and is not re-derived from the refresh request itself.

Refresh tokens issued under this profile MUST be sender-constrained to the originating instance's cnf key, by the same mechanism used to sender-constrain the access token (Section 8.5). Only the originating instance can present the refresh token. A refresh request MUST NOT introduce a client instance identity if the refresh token was not originally issued under this profile.

A client MAY include actor_token and actor_token_type (Section 8.1) on a refresh request to supply a fresh instance assertion (for example, to rotate the underlying assertion before its exp). When present, the actor_token MUST:

* be bound to the same cnf key as the refresh token;

* have (iss, sub) matching those recorded at the refresh token's original issuance; and

* pass the token-type check, instance issuer descriptor lookup, signature verification, JWT claim validation, and client_id binding checks defined in Section 8.2.

If the presented actor_token is a raw JWT-SVID without cnf, the AS MUST establish the binding key per Section 8.8.3 and verify that the established binding key matches the refresh token's binding. The AS MUST reject with invalid_grant any refresh request whose presented actor_token is not bound to the same cnf key, or whose (iss, sub) differ from those recorded at issuance.

Because the refresh token is bound to the originating instance, it is implicitly invalidated when that instance terminates. This keeps act.sub (delegation) or sub (self-acting) stable across the refresh chain, matching the expectation of audit pipelines that a token's actor identity does not change after issuance.

For SPIFFE deployments, the cnf binding key SHOULD outlive the JWT-SVID rotation cycle (typically 5-10 minutes in default SPIRE) when refresh tokens are issued. Deployments that bind cnf to a per-instance DPoP or mTLS key held by the workload satisfy this naturally; deployments that attempt to bind cnf to the SVID's signing key directly will lose refresh-token continuity at every rotation and SHOULD NOT use that pattern.

Deployments requiring cross-instance session continuity (for example, agent platforms whose runtime is recycled but whose session should continue) handle this at the deployment layer rather than through refresh-token semantics. Suitable mechanisms include:

* re-authorizing the client through a fresh authorization grant (e.g., a new authorization_code flow with passive sign-in);

* using token-exchange ([RFC8693]) to mint a new instance-bound token chain in the successor instance from a long-lived parent token held by the platform; or

* persisting session state outside the refresh token (e.g., in a shared store keyed by the user-and-client pair) and obtaining new tokens through a fresh grant.

This profile does not extend refresh-token semantics to cross-instance succession; doing so would break the per-instance audit-stability invariant the profile is designed to provide.

8.8. SPIFFE Compatibility

A SPIFFE workload typically obtains a JWT-SVID from the SPIFFE Workload API. JWT-SVIDs carry iss (the trust domain), sub (the SPIFFE ID), aud, exp, and a signature, and may carry additional registered claims such as iat and jti; they do not carry an OAuth client_id claim. To allow such SVIDs to be presented as actor_token without re-minting, this profile defines an optional SPIFFE compatibility mode driven entirely by descriptor configuration. An AS is not required to support raw JWT-SVID compatibility in order to support re-minted Client Instance Assertions with subject_syntax = "spiffe".

This profile uses JWT-format actor tokens. X.509-SVIDs are not presented as actor_token; SPIFFE deployments using X.509-SVIDs authenticate at the TLS layer (per [RFC8705]) and obtain a JWT-SVID separately for actor_token presentation. The X.509-SVID certificate thumbprint MAY serve as cnf.x5t#S256 in either the re-minted assertion or the issued access token's binding.

The AS MUST select exactly one validation mode before accepting the assertion and MUST apply that mode's rules exclusively:

Mode	Selected when	Key source	Claim requirements	Binding
Raw JWT-SVID mode	The token has no client_id claim and satisfies Section 8.8.1	spiffe_bundle_endpoint	SPIFFE JWT-SVID claims; client_id, typ, cnf, and jti are not required	Established separately per Section 8.8.3
Re-minted assertion mode	The token contains client_id, or raw JWT-SVID mode does not apply	jwks_uri, jwks, or spiffe_bundle_endpoint when the signing key is distributed in the SPIFFE bundle	Full Client Instance Assertion claims and typ per Section 7.2 and Section 7.3	The assertion's cnf drives binding per Section 8.5

Table 4

In raw JWT-SVID mode, the AS MUST:

1. match the descriptor by exact comparison of the JWT-SVID iss to the descriptor's issuer;

2. require subject_syntax = "spiffe";

3. validate the token as a JWT-SVID using SPIFFE JWT-SVID validation rules and the descriptor's spiffe_bundle_endpoint;

4. validate the SPIFFE JWT-SVID claims required by SPIFFE and ignore unrecognized claims unless local policy rejects them;

5. validate aud and exp;

6. validate iat if present;

7. validate nbf if present;

8. enforce the descriptor's signing_alg_values_supported when present;

9. apply the replay-cache rule in Section 12.5 if jti is present;

10. validate sub as a SPIFFE ID and enforce the descriptor's spiffe_id, or trust_domain when spiffe_id is absent; and

11. establish an instance-specific sender-constraint binding per Section 8.8.3.

In raw JWT-SVID mode, the JWT-SVID's iss claim MUST identify the SPIFFE JWT-SVID issuer for the trust domain and MUST exactly match the descriptor's issuer member. Because raw JWT-SVIDs do not require jti, an AS that accepts a raw JWT-SVID without jti MUST rely on sender-constraint and short SVID lifetimes for replay protection.

In re-minted assertion mode, the AS MUST:

1. match the descriptor by exact comparison of the assertion's iss to the descriptor's issuer;

2. verify the JWS signature using the descriptor key source;

3. validate typ = client-instance+jwt per Section 7.3;

4. require and validate the claims defined in Section 7.2, including client_id, exp, iat, jti, and cnf;

5. verify that client_id equals the authenticated client;

6. apply the replay-cache rule in Section 12.5;

7. if subject_syntax is "spiffe", validate sub as a SPIFFE ID and enforce spiffe_id, or trust_domain when spiffe_id is absent; and

8. verify possession of the cnf key per Section 8.5.

A deployment that re-mints an SVID into a Client Instance Assertion MUST include the claims required by Section 7.2 and MUST use typ = client-instance+jwt per Section 7.3.

8.8.1. Client ID Claim Omission

Raw JWT-SVID mode applies when the actor_token has no client_id claim and all of the following hold for the descriptor that matches the actor_token's iss:

* subject_syntax is "spiffe";

* either (a) a spiffe_id member is present and the actor_token's sub satisfies the spiffe_id matching rule of Section 6.1.1 (exact match, or with "/*", path-segment prefix match per [SPIFFE-CLIENT-AUTH]), or (b) spiffe_id is absent, trust_domain is present, and the actor_token's sub falls within that trust domain;

the AS MUST treat the descriptor as the per-client binding for the raw JWT-SVID. In this mode:

* The AS MUST verify that the actor_token's sub satisfies the descriptor's SPIFFE scope: spiffe_id when present, otherwise the descriptor's whole trust_domain.

* All other JWT claims and validation rules of Section 7.2 continue to apply unchanged, except that a raw JWT-SVID is not required to carry iat or jti; if either claim is present, the AS MUST validate it per Section 7.2 and Section 12.5.

A token that contains a client_id claim is processed as a re-minted Client Instance Assertion, not under this omission mode; that claim MUST equal the request's client_id parameter (Section 8.2).

The security rationale is that the descriptor's SPIFFE scope, present in the client's registered metadata (whether published as a CIMD document or stored at the AS), is itself the per-client binding: a workload's SPIFFE ID is bound to a client by the client explicitly listing the prefix that contains it, or by omitting spiffe_id and thereby delegating the whole trust domain. This is the same model [SPIFFE-CLIENT-AUTH] uses for client authentication, applied here to actor identity. Clients SHOULD include spiffe_id unless whole-domain delegation is intentional.

8.8.2. SPIFFE Trust Bundle Resolution

When a descriptor specifies spiffe_bundle_endpoint instead of jwks_uri or jwks, the AS resolves verification keys via the SPIFFE trust bundle endpoint. The AS MUST validate the bundle's freshness and applicability to the trust domain in the descriptor's trust_domain (or the trust domain implied by spiffe_id). The AS MUST verify JWT signatures with JWT authority keys from the bundle for the relevant trust domain, and MUST separately require the assertion's iss and sub to satisfy the descriptor's issuer and SPIFFE scope constraints. The bundle endpoint format, freshness, rotation rules, and TLS authentication (WebPKI) follow [SPIFFE-CLIENT-AUTH], the same handling used for client authentication. When the AS uses an X.509-SVID at the TLS layer for sender-constraint binding under raw-JWT-SVID compatibility (Section 8.8.3), the X.509-SVID is validated against the X.509 trust anchors served by the same SPIFFE bundle.

8.8.3. Sender-Constraint Binding for Raw JWT-SVIDs

A raw JWT-SVID accepted under Section 8.8.1 does not include a cnf claim. The AS MUST establish an instance-specific binding through some other means whose key is attributable to the validated instance. The AS MUST reject the request unless the presented DPoP key or mTLS certificate is bound, by the AS's local policy, to the same runtime named by the JWT-SVID's sub. The binding mechanism MUST establish key custody through a channel independent of the JWT-SVID itself (for example, issuer-provisioned per-instance credentials or a workload attestation channel that names the same sub); accepting a DPoP key solely because it accompanied a valid JWT-SVID is not a binding and reduces this mode to bearer-with-aud.

Acceptable binding mechanisms include:

* a per-instance mTLS client certificate provisioned by the instance issuer (or otherwise tied to instance attestation) and presented under [RFC8705]; when the certificate is an X.509-SVID, the AS MUST verify that its SAN URI exactly equals the JWT-SVID's sub; or

* a DPoP key [RFC9449] that the AS confirms, through deployment-specific attestation or out-of-band binding to the instance issuer, represents the same runtime named by the instance assertion's sub.

In raw-JWT-SVID mode, the AS MUST set the issued access token's top-level cnf to a confirmation member identifying the binding key established above (cnf.x5t#S256 for an X.509-SVID, cnf.jkt for a DPoP key). Access tokens bound via cnf.x5t#S256 to a rotating X.509-SVID are usable only while the workload holds that specific certificate; deployments SHOULD size access-token TTL with the SVID rotation cycle in mind.

The AS SHOULD record which mechanism established the binding and which key or certificate was bound to the instance, to support incident response and per-instance revocation (Section 12.4.2). If the AS cannot establish an instance-specific binding, it MUST reject the request with invalid_request (Section 8.9).

8.9. Error Responses

Errors are returned per [RFC6749] Section 5.2 and [RFC8693] Section 2.2.2. This profile uses the existing OAuth error codes:

* invalid_request: pre-condition and request-shape failures, including missing or mismatched actor_token/actor_token_type, malformed JWT, JWS typ mismatch (Section 7.3), sender-constraint binding failures (Section 8.5), and chain depth exceeding the AS local maximum ([ACTOR-PROFILE]).

* invalid_grant: failures of instance-assertion validation, including signature, JWT claim validation, descriptor lookup or shape, client_id binding, SPIFFE compatibility conditions, classification ambiguity, and actor_token carrying an act claim.

* unsupported_token_type ([RFC8693]): an unrecognized actor_token_type required for the grant.

* invalid_client: when the actor_token is the client authentication credential under Section 8.3, validation failures that would otherwise be returned as invalid_grant are returned as invalid_client.

The AS MAY return additional information via the error_description parameter; deployments MUST NOT include sensitive instance details (e.g., raw SPIFFE IDs of unrelated workloads) in error responses. To help client-developer debugging, AS implementations SHOULD include non-sensitive diagnostic context such as which validation step failed (for example, "issuer not in instance_issuers" or "cnf possession failed").

9. Resource Server Processing

Resource servers consuming access tokens issued under this profile follow the resource server processing rules defined in [ACTOR-PROFILE] (its "Resource Server Processing" section) for delegated access tokens, including actor authorization, JWT access token validation, sender-constraint validation against the top-level cnf, error responses (including the actor_unauthorized error code), and introspection. This section adds only the considerations specific to this profile.

When the access token carries an act claim (Section 8.6.2), the resource server processes it per [ACTOR-PROFILE]; this profile adds no new RS-side requirements for the delegation case. Resource servers MAY use act.sub_profile = client_instance as a signal that the actor is a concrete runtime of an OAuth client, which can inform authorization policy. Resource servers MUST NOT rely on act.iss for proof of possession; per [ACTOR-PROFILE], the top-level cnf is the binding for the current presenter.

Under this profile, the access token's top-level cnf is the _instance's_ key, not the principal's key (the principal in sub is typically a user, who does not present the token). The instance, named in act.sub, is the bearer; sender-constraint validation authenticates that instance. This is a deliberate consequence of binding the access token to the instance that holds the authorization, not to the principal who delegated to it.

Resource-server authorization policies SHOULD evaluate the logical client (client_id), the instance identity (act.sub in delegation tokens or sub in self-acting tokens), actor/subject profile signals such as sub_profile, granted scopes, and issuer context where it is available. Policies that authorize solely on client_id lose the instance-level distinction this profile is designed to provide.

In the self-acting case (Section 8.6.3), the access token carries no act claim. From the resource server's perspective, the access token is a non-delegated token whose top-level claims have the following semantics under this profile:

* sub names the _client instance_ (typically a SPIFFE ID or other workload identifier), not a human user.

* sub_profile = client_instance signals that the subject is a runtime instance and resource servers SHOULD apply policy appropriate to workload identities (for example, scope-restricted machine-to- machine policies, rate limits, or workload-specific audit).

* client_id names the _client_ (the logical OAuth client to which the instance belongs), not the instance. Resource servers MUST NOT treat client_id as the actor identifier in the self- acting case; the actor identifier is sub. (When delegation is also present, this distinction is moot; client_id remains the OAuth client identifier and act.sub names the actor.)

* cnf binds the token to the instance's key; PoP validation follows the access token's binding mechanism (DPoP [RFC9449] or Mutual-TLS-bound [RFC8705]).

Resource servers that distinguish "human-delegated" from "workload-self-acting" requests SHOULD make the determination based on the presence or absence of act, not on the format of sub. In both cases the top-level client_id names the client, not the instance; when the instance is a SPIFFE workload, the SPIFFE ID is conveyed via act.sub (delegation) or sub (self-acting), never via client_id.

10. Adoption and Migration

This profile is designed for incremental adoption. Existing client metadata (CIMD documents or static registrations) that does not declare instance_issuers continues to work unchanged, and existing access tokens in circulation when a client adds instance_issuers remain valid for their original lifetime.

An AS MAY implement this profile while continuing to serve clients that do not use it. Token requests are dispatched on actor_token_type, with urn:ietf:params:oauth:token-type:client-instance-jwt triggering this profile's processing (Section 8.2) and other (or absent) values processed under their own specifications.

ASes SHOULD advertise support via actor_token_types_supported (Section 6.2). Clients SHOULD verify that urn:ietf:params:oauth:token-type:client-instance-jwt is present in the AS's actor_token_types_supported before sending an actor_token on a token request, since RFC 6749 permits ASes that do not implement this extension to silently ignore unrecognized parameters and issue an unbound access token. The actor_token_types_supported value is a coarse capability signal; clients may still need registration-time or deployment agreement for grant-specific use, raw JWT-SVID compatibility, accepted sender-constraint methods, and refresh-token behavior.

A client MAY add instance_issuers at any time. A client that wants to mandate instance assertions for every issued access token can register token_endpoint_auth_method = client_instance_jwt (Section 8.3), which intrinsically requires the actor_token.

Re-minted Client Instance Assertions require cnf (Section 7.2). A deployment whose workload identity system does not yet emit per-instance keys has three options:

* *Adapter pattern*: an OAuth-aware adapter wraps an existing workload identity system (Kubernetes projected service-account tokens, AWS IMDS, GCP metadata server, Azure managed identity, a SPIFFE control plane, etc.) and re-mints a Client Instance Assertion with cnf from the underlying credential, signing with a key registered in the issuer's descriptor. From the AS's perspective, the adapter is the instance issuer (Section 7.1); the adapter enforces the per-client authorization rule, since underlying workload-identity systems typically do not know about OAuth clients or their class-and-instance relationship. The adapter holds the workload-identifier → client_id mapping, the OAuth signing keys, and re-issues at the underlying credential's rotation cadence. Recommended for non-SPIFFE deployments and for SPIFFE deployments that can run an adapter.

* *Raw JWT-SVID compatibility*: the SVID is presented as actor_token without re-minting and the AS establishes binding through a channel independent of the SVID (Section 8.8.1, Section 8.8.3). This is the SPIFFE-native path. For workloads with X.509-SVIDs, the X.509-SVID can serve as the binding key in two shapes:

  - *Re-minted with cnf*: an adapter mints a Client Instance Assertion whose cnf.x5t#S256 is the SHA-256 thumbprint of the workload's X.509-SVID certificate. The workload presents the assertion as actor_token and the same X.509-SVID at TLS under [RFC8705]; the AS verifies that the TLS certificate thumbprint matches cnf.x5t#S256. Use when an OAuth-aware adapter is available.

  - *Raw JWT-SVID with X.509 binding*: the workload presents its JWT-SVID directly as actor_token (no re-mint, no cnf) and its X.509-SVID at TLS; the AS treats the X.509-SVID as the sender-constraint binding under Section 8.8.3. Use when the workload runs against unmodified SPIFFE infrastructure.

  Both ground the binding in the SPIFFE trust domain that issued the SVID and avoid a separate DPoP key. Worked example in Appendix "SPIFFE Workload (Self-Acting, JWT-SVID Reuse with X.509-SVID Binding)".

* *Defer adoption* until the workload identity system can emit per-instance keys directly.

ASes and OAuth client operators SHOULD NOT enable the client_instance_jwt authentication method (Section 8.3) without cnf: that mode has no fallback client credential, so a cnf-less assertion is fully bearer at presentation (Section 12.7).

11. Conformance

An AS conforms to this document by implementing the actor token grant extension (Section 4) and processing for urn:ietf:params:oauth:token-type:client-instance-jwt (Section 8, Section 8.6, Section 8.9, plus [ACTOR-PROFILE]). Raw JWT-SVID compatibility (Section 8.8) and the client_instance_jwt authentication method (Section 8.3) are optional capabilities; an AS that supports either MUST conform to its respective section. A client using this profile MUST publish or register instance_issuers metadata (Section 6.1.1) and MUST ensure each listed issuer is authorized to attest its instances. An instance issuer MUST mint assertions per Section 7.2, Section 7.3, and Section 5.3.1. A resource server MUST process delegated tokens per [ACTOR-PROFILE] and apply the self-acting semantics in Section 9 when act is absent.

12. Security Considerations

This document inherits the security considerations of [RFC6749], [RFC7519], [RFC7523], [RFC8693], [RFC8725], [CIMD], and [ACTOR-PROFILE].

12.1. Trust Model

The normative trust model for this profile is in Section 5.3. This subsection summarizes the security implications.

A client delegates the authentication of its instances to one or more instance issuers. A compromised or misconfigured instance issuer can mint instance assertions that the AS will accept as legitimate instances of the named client. Clients SHOULD list only instance issuers under their own administrative control (or contractually equivalent), and SHOULD set spiffe_id, trust_domain, and signing_alg_values_supported to bound what each issuer is allowed to assert. Clients using SPIFFE SHOULD include spiffe_id; omitting it delegates the whole SPIFFE trust domain and is appropriate only when every workload in that trust domain is authorized to act as an instance of the client. After a client detaches a compromised issuer, tokens minted under the prior trust may continue to validate up to the trust-withdrawal latency bound in Section 12.2; operators SHOULD plan incident response around this window.

Client metadata is itself trust-affecting: an attacker who can modify it can add a new instance issuer under their control. Adding an instance_issuers entry, widening a descriptor’s spiffe_id or trust_domain, changing a descriptor’s key source, or enabling client_instance_jwt is equivalent to adding or expanding a credential issuer for the client and SHOULD require high-assurance change

control by the client operator and AS. Clients publishing CIMD metadata MUST protect the publication channel (per [CIMD]'s requirement of HTTPS) and the storage backing it; deployments using static registration MUST protect the registration store and any administrative API used to update it. ASes resolving CIMD documents inherit [CIMD]'s security considerations covering transport, intermediary caches, and DNS.

12.2. Trust-Withdrawal Latency

The trust-withdrawal latency, that is, the worst-case time from a client metadata change to all derived access tokens having expired, is approximately the sum of the metadata refresh interval (for CIMD, the cache TTL; for static registration, the expected lag between an admin update and AS-side propagation), the instance assertion's exp window, the AS's JWKS or SPIFFE-bundle cache TTL for the issuer, and the access token TTL. ASes SHOULD size these components together so that the resulting latency matches their incident-response target.

Recommended defaults: instance assertion exp of 5 minutes or less (Section 12.5); access-token TTL of 30 minutes or less paired with a metadata refresh interval of 30 minutes or less, giving a trust-withdrawal latency of approximately 65 minutes. Tighter deployments (5-minute access tokens with 5-minute caches, approximately 15 minutes) and looser deployments (60-minute tokens with 1-hour caches, approximately 2 hours) are both common; looser deployments SHOULD support active revocation (Section 12.4) and introspection-based status checks at the resource server. Operators SHOULD treat this latency as a deployment-time SLO matching their incident-response requirements.

12.3. Instance Lifecycle

Client instances are short-lived in many deployments (containers, function invocations, agent sessions). This profile relies on three mechanisms to keep actor identity current:

Rotation: Instance issuers SHOULD mint short-lived instance assertions (Section 12.5). New tokens are issued continuously as instances start, restart, or rotate keys.

Revocation within the validity window: Within an instance assertion's exp window, the AS prevents reuse via the jti replay rule (Section 12.5). A specific issued access token, or all tokens associated with an instance, can be revoked only via the AS's own revocation mechanisms (Section 12.4); this profile does not define a standardized revocation endpoint or instance revocation list format.

Trust withdrawal: To stop accepting instance assertions from an issuer (e.g., after a workload identity compromise), the client removes the issuer from instance_issuers, replaces or removes trust_domain, or rotates jwks at the issuer level. The AS's response is governed by Section 5.3.3: subsequent uses of access tokens whose act references the withdrawn scope are treated as no longer endorsed.

Refresh windows are a particular concern: an access token refreshed without a new instance assertion may carry stale instance identity long after the original instance has terminated. ASes SHOULD prefer requiring a fresh instance assertion on refresh (Section 8.7), or set short refresh intervals when instance identity is present.

12.4. Token Revocation

This profile supports two complementary revocation models for access tokens issued under it. Both build on [RFC7009]; deployments MAY support either, both, or neither.

After the AS has adopted updated client metadata (Section 5.3.3), the AS SHOULD treat further use of access tokens whose validated instance identity is no longer endorsed by the client as invalid:

* for delegation tokens, when the act claim names a removed instance issuer or falls outside the descriptor's updated scope;

* for self-acting tokens, when the instance issuer recorded by the AS at issuance time has been removed, or when the access token's sub falls outside the descriptor's updated scope.

Where the deployment supports it, this is naturally enforced by introspection (Section 12.4.3) and short access-token lifetimes; AS implementations MAY additionally revoke such tokens per [RFC7009], including via the per-instance mechanism in Section 12.4.2. ASes MAY apply the same policy to changes in a descriptor's jwks_uri, jwks, or spiffe_bundle_endpoint keys that [CIMD] permits for changes in client-level keys.

12.4.1. Per-Token Revocation

An AS that supports [RFC7009] revocation MAY accept the access token (or its associated refresh token) as the token parameter and revoke that specific issued token. This works unchanged for tokens issued under this profile; no profile-specific extensions to the revocation endpoint are required.

12.4.2. Per-Instance Revocation

When an instance is compromised or otherwise needs to be quarantined, a deployment may need to invalidate all access tokens whose validated instance issuer and instance subject identify that instance, without enumerating every issued token. ASes implementing this profile SHOULD support a per-instance revocation mode keyed by the pair (instance issuer, instance subject):

* for delegation tokens, the key is (act.iss, act.sub);
* for self-acting tokens, the key is the instance issuer recorded by the AS at issuance time together with the access token's sub;
* invalidates all currently-active access tokens matching that key, with descriptor scope as an optional additional filter;
* prevents issuance of new access tokens with that instance issuer-and-subject pair as actor or principal until a follow-up condition is met (for example, expiration of an internal blocklist entry, or removal of the instance from the workload identity system).

The mechanism for triggering per-instance revocation is deployment-specific and out of scope for this document.

12.4.3. Introspection Behavior on Revocation

When an AS supports introspection ([RFC7662]), introspection responses for access tokens issued under this profile MUST honor both per-token revocation and per-instance revocation: an introspection response MUST return active = false for any access token that has been revoked under either model. Introspection MUST also honor the trust update rules in Section 5.3.3: when the AS has adopted updated client metadata that removes an instance issuer or narrows a descriptor's scope so that an issued access token's act (or, for self-acting tokens, sub) is no longer endorsed, introspection responses for that access token MUST return active = false once the AS has applied the update. Active introspection responses SHOULD include the access token's top-level cnf so that introspection-based PoP has the binding key (delegation case follows [ACTOR-PROFILE]); for self-acting tokens, sub_profile and cnf SHOULD also be returned.

12.4.4. Revocation and Refresh Tokens

When a refresh token is sender-constrained to the originating instance (Section 8.7), per-instance revocation MUST also revoke the refresh token (and prevent any further access tokens it would mint). This profile does not define successor-instance refresh; deployments that need cross-instance session continuity use the separate mechanisms described in Section 8.7.

12.5. Replay

Actor tokens MUST include jti, exp, and iat (Section 7.2), except for raw JWT-SVIDs accepted under Section 8.8.1.

After identifying the issuer and validating the signature, the AS MUST reject a token whose (iss, jti) pair has already been seen within the token's validity window, and MUST retain replay-cache entries at least until the token's exp plus any allowed clock skew. For raw JWT-SVIDs, this check applies only when jti is present; otherwise replay is bounded by sender-constraint, short SVID lifetimes, and audience restriction.

An AS MAY skip the replay check for cnf-bound assertions that have been PoP-verified at presentation, treating them as reusable within their exp window, provided the deployment documents this behavior and applies rate limits, monitoring, and audit logging. This shifts the blast radius of cnf-key compromise from one access token per assertion to the rate-limit ceiling within exp; the rate limit is therefore the bound on that threat. The MUST applies unconditionally to assertions without a verified cnf.

Issuers SHOULD use short lifetimes (five minutes or less). On refresh (Section 8.7), AS implementations SHOULD prefer requiring a fresh instance assertion. Distributed AS deployments MUST share the replay cache or coordinate to prevent cross-replica replay, except on cnf-bound paths in reusable mode where cnf+PoP verification is correctness-preserving across replicas.

A compromised instance-issuer signing key creates a denial-of-service surface: an attacker can mint validly-signed assertions with arbitrary jti values. ASes SHOULD apply per-issuer rate limits and bounded cache caps; a sustained high rate of distinct jti values from a single issuer is a signal of compromise.

12.6. Audience and Confused Deputy

The aud claim binds the instance assertion to a specific AS, preventing one AS from replaying it against another ([RFC7523] Section 3). The client_id claim, which this document treats as a binding (not as actor identity), prevents an instance assertion issued for one client from being presented under a different client's authentication.

12.7. Trust-Root Collapse

The client_instance_jwt authentication method (Section 8.3) collapses two trust roots (client credential and instance issuer) into one. Compromise of any listed instance issuer is sufficient to mint tokens that authenticate as the client. Modes such as private_key_jwt require an attacker to possess both the instance issuer's signing key and the client's private key; that two-key property holds only when the two keys live in different security domains (e.g., the client's key in an HSM or signing service distinct from the runtime). Where genuine separation is not feasible, deployments SHOULD use Section 8.3 explicitly rather than rely on nominal two-key authentication that does not actually separate custody.

When client_instance_jwt is used, clients SHOULD constrain each instance issuer's authority through spiffe_id, trust_domain, and signing_alg_values_supported, and SHOULD list only the minimum set of instance_issuers necessary. Trust withdrawal under this auth method has immediate consequences: removing an instance issuer from instance_issuers (or narrowing its descriptor scope) invalidates client authentications that depended on that issuer's endorsement, and the AS MUST stop accepting client_instance_jwt authentications via the removed or narrowed issuer once it has applied the metadata update (Section 5.3.3).

12.8. Mode-Switch Between Delegation and Self-Acting

Whether an issued access token represents delegation or self-acting (Section 8.6.1) determines whether the instance is exposed to resource servers as act or as sub. An adversary that can influence classification could escalate privileges, for example by inducing the AS to drop a sub belonging to a user and re-anchor the token on the instance's sub. The classification rule in Section 8.6.1 is determined by the grant type, not by comparison of attacker-influenceable subject strings; ASes MUST NOT employ heuristic or fuzzy matching of assertion contents to override the table. In particular, ASes MUST NOT normalize either side of any comparison they perform on subject identifiers (no Unicode normalization, no case folding, no percent-decoding beyond what [RFC7519] requires for

JSON parsing). When classification is ambiguous (for example, custom grants not listed in the table), the AS MUST refuse rather than guess.

12.9. Sender-Constraint Requirement

Without sender-constraint, an act claim is an assertion about who acted, not a binding enforced at the resource server: any party in possession of the access token can present it as the named actor. Section 8.5 therefore requires sender-constrained access tokens and forbids bearer issuance under this profile. Per [ACTOR-PROFILE], the resource server validates proof of possession against the access token's top-level cnf only; confirmation members inside an act object are actor context for audit and correlation, not a binding the RS independently verifies.

12.10. Delegation Control

Unbounded delegation chains permit privilege amplification across boundaries. AS implementations MUST enforce a local maximum delegation depth ([ACTOR-PROFILE]). [ACTOR-PROFILE] recommends supporting at least depth 4 for cross-domain interop; deployments imposing lower ceilings should weigh interoperability against the privilege-amplification surface they are willing to allow.

12.11. Privacy

A client instance assertion reveals fine-grained workload identity to the AS and, after issuance, to resource servers via the act claim (delegation case) or the access token's top-level sub (self-acting case). Exposing per-instance identity to resource servers is the deliberate purpose of this profile (it is what enables instance-level audit, authorization, and binding downstream), but it has privacy and operational consequences:

* Resource servers gain visibility into the deploying organization's internal workload structure, including (depending on sub) cluster names, namespaces, function instance IDs, or session identifiers. Resource server operators SHOULD treat this information with the same care as any other identity attribute received from an AS, and SHOULD NOT log or propagate it more broadly than necessary.

* Naming conventions in sub may inadvertently encode sensitive details. Issuers and clients SHOULD avoid encoding identifiers of human users, secret material, or internal infrastructure topology in sub, and SHOULD prefer opaque or hierarchical identifiers (e.g., a SPIFFE path) whose minimum granularity matches the auditing need.

The error response guidance in Section 8.9 extends to logs and audit trails: instance assertion contents SHOULD be logged at a level commensurate with the sensitivity of the workload identity they convey.

For incident response, per-instance revocation (Section 12.4.2), and operational audit, ASes issuing access tokens under this profile SHOULD log enough information at issuance time to support per-instance revocation (Section 12.4.2) and incident response, subject to the sensitivity guidance above.

13. IANA Considerations

13.1. OAuth Token Type

IANA is requested to register the following value in the "OAuth URI" registry established by [RFC6755] (and used by [RFC8693] for actor_token_type values):

URN: urn:ietf:params:oauth:token-type:client-instance-jwt

Common Name: OAuth 2.0 Client Instance Assertion

Change Controller: IETF

Specification Document(s): This document

13.2. OAuth Dynamic Client Registration Metadata

IANA is requested to register the following parameters in the "OAuth Dynamic Client Registration Metadata" registry established by [RFC7591]. The Change Controller for each entry is IETF.

13.2.1. instance_issuers

Client Metadata Name: instance_issuers

Client Metadata Description: Trusted issuers of client instance assertions for this client.

Specification Document(s): Section 6.1.1 of this document

13.3. OAuth Token Endpoint Authentication Method

IANA is requested to register the following value in the "OAuth Token Endpoint Authentication Methods" registry established by [RFC8414]:

Token Endpoint Authentication Method Name: client_instance_jwt

Change Controller: IETF

Specification Document(s): Section 8.3 of this document

13.4. OAuth Authorization Server Metadata

IANA is requested to register the following parameters in the "OAuth Authorization Server Metadata" registry established by [RFC8414]. The Change Controller for each entry is IETF.

13.4.1. actor_token_types_supported

Metadata Name: actor_token_types_supported

Metadata Description: JSON array of actor_token_type values supported at the token endpoint.

Specification Document(s): Section 6.2 of this document

13.5. Media Type

IANA is requested to register the following media type in the "Media Types" registry:

Type name: application

Subtype name: client-instance+jwt

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary; A Client Instance Assertion is a JWT; JWT values are encoded as a series of base64url-encoded values separated by period characters.

Security considerations: See Section 12 of this document.

Interoperability considerations: N/A

Published specification: This document.

Applications that use this media type: OAuth 2.0 authorization servers and clients implementing this profile.

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A; File extension(s): N/

A; Macintosh file type code(s): N/A

Person & email address to contact for further information: Karl McGuinness public@karlmcguinness.com (mailto:public@karlmcguinness.com)

Intended usage: COMMON

Restrictions on usage: none

Author: Karl McGuinness

Change controller: IETF

This media type, when used as a value of the typ JWS protected header parameter ([RFC7515] Section 4.1.9), MUST be client-instance+jwt (per [RFC8725] Section 3.11; the application/ prefix is omitted).

13.6. OAuth Entity Profile

IANA is requested to register the following value in the "OAuth Entity Profiles" registry established by [ENTITY-PROFILES]. This registration is contingent on the establishment of that registry.

Profile Name: client_instance

Profile Description: A concrete runtime instance of an OAuth client identified by a client_id.

Profile Usage Location: Actor Profile

Change Controller: IETF

Specification Document(s): This document

14. References

14.1. Normative References

[ACTOR-PROFILE]
McGuinness, K., "OAuth Actor Profile for Delegation", Work in Progress, Internet-Draft, draft-mcguinness-oauth-actor-profile-00, 30 April 2026, <https://datatracker.ietf.org/doc/html/draft-mcguinness-oauth-actor-profile-00>.

[ENTITY-PROFILES]
Mora, S. C., Dingle, P., and K. McGuinness, "OAuth 2.0 Entity Profiles", Work in Progress, Internet-Draft, draft-mora-oauth-entity-profiles-01, 15 April 2026, <https://datatracker.ietf.org/doc/html/draft-mora-oauth-entity-profiles-01>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/rfc/rfc6749>.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/rfc/rfc6755>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/rfc/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/rfc/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/rfc/rfc7519>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/rfc/rfc7523>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/rfc/rfc7591>.

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <https://www.rfc-editor.org/rfc/rfc7662>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/rfc/rfc7800>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/rfc/rfc8414>.

[RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/rfc/rfc8693>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/rfc/rfc8705>.

[RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/rfc/rfc8725>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/rfc/rfc9068>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/rfc/rfc9449>.

[SPIFFE-CLIENT-AUTH] Schwenkschuster, A., Kasselman, P., Rose, S., and S. Thorgersen, "OAuth SPIFFE Client Authentication", Work in Progress, Internet-Draft, draft-ietf-oauth-spiffe-client-auth-01, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-spiffe-client-auth-01>.

14.2. Informative References

[CIMD] Parecki, A. and E. Smith, "OAuth Client ID Metadata Document", Work in Progress, Internet-Draft, draft-ietf-oauth-client-id-metadata-document-01, 1 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-client-id-metadata-document-01>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/rfc/rfc7009>.

[SPIFFE] SPIFFE, "SPIFFE: Secure Production Identity Framework For Everyone", 2024, <https://spiffe.io/docs/latest/spiffe-about/spiffe-concepts/>.

[WIMSE-ARCH] Salowey, J. A., Rosomakho, Y., and H. Tschofenig, "Workload Identity in a Multi System Environment (WIMSE) Architecture", Work in Progress, Internet-Draft, draft-ietf-wimse-arch-07, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-wimse-arch-07>.

[WIMSE-CREDS] Campbell, B., Salowey, J. A., Schwenkschuster, A., Sheffer, Y., and Y. Rosomakho, "WIMSE Workload Credentials", Work in Progress, Internet-Draft, draft-ietf-wimse-workload-creds-01, 5 May 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-wimse-workload-creds-01>.

Design Rationale

This appendix records design choices that motivated the normative text.

Why not a client_instance request parameter?

A new top-level client_instance parameter would have to flow through the authorization request, the token request, introspection, the access token, and several existing extensions. Each is a separate specification touch-point and a deployment cliff. Reusing actor_token keeps the protocol surface unchanged.

Why not a dedicated client_instance_assertion request parameter?

A dedicated client_instance_assertion parameter would name what this profile carries on the wire more directly than a typed actor_token. This document reuses actor_token for three reasons.

First, this document also defines the actor token grant extension (Section 4): a wire mechanism for any actor_token_type to appear on grants beyond token-exchange. A separate client_instance_assertion parameter would not eliminate actor_token; it would parallel it for one use case, doubling the wire surface.

Second, in delegated flows the instance is the actor, and actor_token is already the established wire mechanism for naming the actor (required by [RFC8693] on token-exchange). Introducing client_instance_assertion alongside actor_token would carry the same JWT in two parameter slots for delegated flows. Using actor_token for both keeps the request side canonical with the token side, where the validated artifact populates act (delegation case) or sub (self-acting case) per [ACTOR-PROFILE].

Third, an actor_token_type URN is a well-known IANA-registered extension point; clients with existing actor_token plumbing pick up client-instance-jwt without parser changes.

Profiles that prefer a dedicated request parameter can be defined later as a wire-syntax sibling without changing the token shape defined here.

Why extend actor_token to non-token-exchange grants?

[RFC8693] already defines request parameters for presenting an actor token and identifying its token type. Those parameters are not specific to token exchange as a protocol concept; token exchange is only the grant on which [RFC8693] defines them. Other grants also need a way to identify the actor performing the request. Reusing the existing parameter machinery avoids grant-specific actor parameters and gives actor-token profiles a single token endpoint presentation model.

Client instance identity is one motivating example: the instance acts on behalf of the subject (the human user or service principal) under the authority of the client. Other actor delegation use cases share the same presentation need. The only normative move in Section 4 is permitting actor_token and actor_token_type on additional grants; validation and access-token representation remain token-type-specific.

The actor token grant extension is intentionally separable from the specific actor_token_type defined here. By isolating the actor token grant extension (Section 4), this document provides future profiles a defined wire mechanism to build on, rather than each profile re-litigating the question of whether actor_token may appear on authorization_code or client_credentials. See Section 4.2.

If working group consensus prefers to progress the grant extension independently from the client instance assertion profile, the extension can be split into a companion specification without changing the client-instance-jwt token type. In that structure, this profile would depend on the companion grant-extension specification for the wire-level permission to carry actor_token and actor_token_type on non-token-exchange grants.

Why client metadata as the trust anchor for instance issuers?

The trust relationship between a client and its instance issuers is published in the client's registered metadata (either in a CIMD document or in the AS's registration store). This keeps the trust relationship auditable alongside other client metadata and reuses existing freshness, caching, and key-rotation mechanisms. Locating it elsewhere (in AS-side static configuration unrelated to the client, or in a separate trust-relationship registry) would have fragmented the surface and reduced the auditability of who trusts whom.

Why a dedicated actor_token_type URN?

The general-purpose JWT token type does not signal that the AS should look up trust via the instance_issuers client metadata, nor that client_id binding applies. A dedicated URN lets ASes route processing unambiguously and lets clients advertise support via actor_token_types_supported.

Why a token_endpoint_auth_method rather than a client_assertion_type?

[SPIFFE-CLIENT-AUTH] models its workload-identity-as-client-auth mechanism as a client_assertion_type. The natural question is why Section 8.3 does not.

The two cases differ in what the JWT names. A SPIFFE JWT-SVID presented as a client_assertion under [SPIFFE-CLIENT-AUTH] names _the client_ (its sub is the spiffe_id of the workload acting as the client). The client-metadata listing of spiffe_id, including the permitted "/*" path-segment wildcard, turns the SVID into a credential for the client. There is no separate notion of "instance" on the wire.

A client instance assertion under this profile names _the instance_: its sub is the instance identifier and its client_id claim names the client. The same JWT is required to do double duty only when the client chooses token_endpoint_auth_method = client_instance_jwt; in every other auth method, a separate client credential authenticates the client and the instance assertion names the instance.

Modeling the dual-use case as a client_assertion_type would have required either (a) inventing a second token type identical to client-instance-jwt to be the assertion, doubling the wire surface, or (b) overloading client_assertion_type with the actor-token URN, which conflicts with that URN's role as actor_token_type. Modeling it as a token_endpoint_auth_method captures what is actually happening, namely that the AS authenticates the client implicitly from its client-metadata endorsement of the instance assertion's issuer, while keeping client_assertion and actor_token semantically distinct.

Why reuse actor_token in the self-acting case?

In the self-acting case (Section 8.6.3) the issued access token's principal is the instance itself, not a separate party. RFC 8693's "actor" framing literally describes the actor as the party acting on behalf of the subject; with no other subject present, the framing is technically a stretch.

Two considerations specific to the self-acting case led to reuse rather than introducing a parallel "subject_assertion" parameter (the general argument against a new request parameter is in Appendix "Why not a dedicated client_instance_assertion request parameter?"):

1. _Classification belongs to the grant._ Whether the issued access token represents delegation or self-acting is determined by the grant (Section 8.6.1), not by the instance assertion. The same instance assertion can correctly produce either shape depending on the grant it accompanies.

2. _Deployment fit._ Workload identity systems already issue exactly this artifact for both purposes. Requiring deployments to re-mint the same JWT under a different parameter name to satisfy an academic distinction would not improve security.

The cost is that [RFC8693]'s "actor" terminology must be read with this profile's classification rules in mind. Implementations and specification readers should treat actor_token in this profile as "validated instance identity assertion," with the understanding that its placement in the issued access token (act vs. sub) is governed by Section 8.6.1.

Worked Examples

This appendix gives end-to-end worked examples for each grant type that interacts with this profile. Examples are non-normative and omit unrelated headers or grant-specific details that do not affect actor processing. Decoded actor_token blocks show only the JWT payload; re-minted Client Instance Assertions also carry a JWS protected header with typ set to client-instance+jwt per Section 7.3 (see the full example in Section 7.4). Timestamps and lifetimes in these examples are illustrative and do not override the lifetime guidance in Section 12.2 and Section 12.5.

The examples use a CIMD-style client_id for clarity; the same flows apply identically to static-registration deployments (Section 5.2), where the client_id is opaque or AS-assigned and the metadata is read from the AS's registration store rather than dereferenced.

The examples share a common deployment:

* OAuth client: https://app.example.com/agent
* Client metadata declares one instance issuer https://workload.app.example.com (subject_syntax "uri", signing_alg_values_supported containing ES256).
* AS: https://as.example.com.
* Resource server: https://api.example.com.
* All access tokens are DPoP-bound; the instance assertion's cnf carries the instance's DPoP key thumbprint.

Authorization Code with User Delegation

Alice authorizes the agent (client) at the AS through a standard authorization_code flow. The agent runs as instance inst-01, which presents an instance assertion at the token endpoint to identify itself.

Authorization request from the agent (abridged). dpop_jkt carries the thumbprint of inst-01's DPoP key (matching the instance assertion's cnf.jkt) to establish key-bound continuity per Section 8.4:

```
GET /authorize?response_type=code
  &client_id=https%3A%2F%2Fapp.example.com%2Fagent
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Fcb
  &scope=repo.write
  &state=xyz
  &code_challenge=...
  &code_challenge_method=S256
  &dpop_jkt=0ZcOCORZNYy...iguA4I HTTP/1.1
Host: as.example.com
```

The AS authenticates Alice, displays consent for the client (per Section 8.4 consent applies to the client as a whole, not per-instance), binds the authorization code to the dpop_jkt value per [RFC9449], and redirects with an authorization_code.

Token request from instance inst-01:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: <DPoP proof bound to inst-01's key>

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fapp.example.com%2Fcb
&client_id=https%3A%2F%2Fapp.example.com%2Fagent
&code_verifier=...
&client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJFUzI1NiIs... (private_key_jwt)
&actor_token=eyJhbGciOiJFUzI1NiIs...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

Decoded actor_token:

```
{
  "iss":         "https://workload.app.example.com",
  "sub":         "https://workload.app.example.com/inst-01",
  "aud":         "https://as.example.com",
  "client_id":   "https://app.example.com/agent",
  "sub_profile": "client_instance",
  "iat":         1770000000,
  "exp":         1770000300,
  "jti":         "ac-1a2b3c",
  "cnf":         { "jkt": "0ZcOCORZNYy...iguA4I" }
}
```

AS validation:

1. Authenticates the client (private_key_jwt).

2. Recognizes actor_token_type.

3. Resolves CIMD for the agent.

4. Locates the descriptor by matching iss.

5. Verifies signature via descriptor’s jwks_uri.

6. Validates JWT claims.

7. Verifies actor_token.client_id == request client_id and applies the (iss, jti) replay check.

8. Applies AS-local maximum delegation depth.

9. Section 8.4: the request’s client_id matches the client_id that received the code, and the DPoP proof’s thumbprint matches both the code’s dpop_jkt and the actor token’s cnf.jkt.

10. Classifies as delegation; issues sender-constrained access token.

Issued access token:

```
{
  "iss":       "https://as.example.com",
  "aud":       "https://api.example.com",
  "sub":       "user:alice@example.com",
  "client_id": "https://app.example.com/agent",
  "scope":     "repo.write",
  "iat":       1770000005,
  "exp":       1770001805,
  "cnf":       { "jkt": "0ZcOCORZNYy...iguA4I" },
  "act": {
    "iss":         "https://workload.app.example.com",
    "sub":         "https://workload.app.example.com/inst-01",
    "sub_profile": "client_instance",
    "cnf":         { "jkt": "0ZcOCORZNYy...iguA4I" }
  }
}
```

The RS authorizes the request based on (alice, inst-01) per Section 9.

Client Credentials (Self-Acting)

A workload makes a client_credentials request with no human user involved.

Token request:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: <DPoP proof bound to inst-02's key>

grant_type=client_credentials
&scope=repo.read
&client_id=https%3A%2F%2Fapp.example.com%2Fagent
&client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJFUzI1NiIs...
&actor_token=eyJhbGciOiJFUzI1NiIs...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

Decoded actor_token:

```
{
  "iss":         "https://workload.app.example.com",
  "sub":         "https://workload.app.example.com/inst-02",
  "aud":         "https://as.example.com",
  "client_id":   "https://app.example.com/agent",
  "sub_profile": "client_instance",
  "iat":         1770000000,
  "exp":         1770000300,
  "jti":         "cc-2b3c4d",
  "cnf":         { "jkt": "PqR...XyZ" }
}
```

Issued access token (self-acting; no user, instance is the principal):

```
{
  "iss":         "https://as.example.com",
  "aud":         "https://api.example.com",
  "sub":         "https://workload.app.example.com/inst-02",
  "sub_profile": "client_instance",
  "client_id":   "https://app.example.com/agent",
  "scope":       "repo.read",
  "iat":         1770000005,
  "exp":         1770001805,
  "cnf":         { "jkt": "PqR...XyZ" }
}
```

The RS treats sub as the workload identity (not a human user) per Section 9.

Token Exchange with Prior Delegation Chain (Agent Spawns Sub-Agent)

A parent agent’s user-delegated access token is exchanged at the AS for a sub-agent’s downstream-resource-scoped token. The sub-agent runtime presents an instance assertion; the inbound subject_token already carries an act chain naming the parent agent. The subject_token was issued by upstream.example.com, which as.example.com trusts as a token issuer under local policy.

Inbound subject_token (decoded; issued earlier when a parent agent "agent-orchestrator-alpha" obtained access on the user’s behalf):

```
{
  "iss":       "https://upstream.example.com",
  "aud":       "https://app.example.com/agent",
  "sub":       "user:alice@example.com",
  "scope":     "repo.write",
  "act": {
    "iss":         "https://platform.example.com",
    "sub":         "agent:orchestrator-alpha",
    "sub_profile": "client_instance"
  }
}
```

Token request:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: <DPoP proof bound to inst-03's key>

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=https%3A%2F%2Fapi.example.com
&subject_token=eyJhbGciOiJFUzI1NiIs...
&subject_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
&client_id=https%3A%2F%2Fapp.example.com%2Fagent
&client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJFUzI1NiIs...
&actor_token=eyJhbGciOiJFUzI1NiIs...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

The actor_token is the sub-agent runtime inst-03, with no act of its own (per [ACTOR-PROFILE], actor_token MUST NOT carry act). Decoded actor_token:

```
{
  "iss":         "https://workload.app.example.com",
  "sub":         "https://workload.app.example.com/inst-03",
  "aud":         "https://as.example.com",
  "client_id":   "https://app.example.com/agent",
  "sub_profile": "client_instance",
  "iat":         1770000005,
  "exp":         1770000305,
  "jti":         "tx-3c4d5e",
  "cnf":         { "jkt": "AbC...123" }
}
```

Chain construction per Section 8.6.4:

* Outermost: the validated actor_token (sub-agent inst-03).

* Inner: the subject_token's act chain, preserved (agent-orchestrator-alpha).

Issued access token:

```
{
  "iss":       "https://as.example.com",
  "aud":       "https://api.example.com",
  "sub":       "user:alice@example.com",
  "client_id": "https://app.example.com/agent",
  "scope":     "repo.write",
  "iat":       1770000010,
  "exp":       1770001810,
  "cnf":       { "jkt": "AbC...123" },
  "act": {
    "iss":         "https://workload.app.example.com",
    "sub":         "https://workload.app.example.com/inst-03",
    "sub_profile": "client_instance",
    "cnf":         { "jkt": "AbC...123" },
    "act": {
      "iss":         "https://platform.example.com",
      "sub":         "agent:orchestrator-alpha",
      "sub_profile": "client_instance"
    }
  }
}
```

Resulting chain depth is 2, well within typical AS-local maximums. The chain reads outward-in as: sub-agent inst-03 acted on behalf of the parent agent agent-orchestrator-alpha, which acted on behalf of user alice. Each act layer names a distinct runtime; resource servers and audit pipelines can attribute the request to the specific sub-agent that performed it.

SPIFFE Workload (Self-Acting, JWT-SVID Reuse with X.509-SVID Binding)

A SPIFFE workload makes a client_credentials request. The workload holds both a JWT-SVID and an X.509-SVID issued by its SPIFFE trust domain. The workload presents the JWT-SVID directly as actor_token under raw JWT-SVID compatibility (Section 8.8.1) and presents the X.509-SVID at TLS, so the certificate supplies the sender-constraint binding ([RFC8705]). Because the actor_token is a raw JWT-SVID, its JOSE header follows SPIFFE conventions and is not required to carry typ=client-instance+jwt.

The descriptor uses subject_syntax="spiffe" and the access token is mTLS-bound, in place of the preamble’s URI-syntax descriptor and DPoP binding.

Instance issuer descriptor:

```
{
  "issuer": "spiffe://example.com",
  "spiffe_bundle_endpoint": "https://example.com/spiffe/bundle",
  "subject_syntax": "spiffe",
  "trust_domain": "example.com",
  "spiffe_id": "spiffe://example.com/agent/*",
  "signing_alg_values_supported": ["ES256"]
}
```

Token request (TLS presents the workload's X.509-SVID; client authentication uses [SPIFFE-CLIENT-AUTH] X.509 mode, so client_assertion is omitted):

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&scope=repo.read
&client_id=https%3A%2F%2Fapp.example.com%2Fagent
&actor_token=eyJhbGciOiJFUzI1NiIs...
&actor_token_type=
  urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aclient-instance-jwt
```

Decoded actor_token:

```
{
  "iss":         "spiffe://example.com",
  "sub":         "spiffe://example.com/agent/inst-04",
  "aud":         "https://as.example.com",
  "iat":         1770000000,
  "exp":         1770000300
}
```

The AS verifies the JWT-SVID signature against the trust bundle, that sub falls under the descriptor's spiffe_id prefix, that the TLS-presented X.509-SVID has SAN URI exactly equal to the JWT-SVID's sub, and binds the issued access token via cnf.x5t#S256 set to the certificate's SHA-256 thumbprint. Issued access token (self-acting; TTL kept short relative to the X.509-SVID rotation cycle):

```
{
  "iss":         "https://as.example.com",
  "aud":         "https://api.example.com",
  "sub":         "spiffe://example.com/agent/inst-04",
  "sub_profile": "client_instance",
  "client_id":   "https://app.example.com/agent",
  "scope":       "repo.read",
  "iat":         1770000005,
  "exp":         1770001805,
  "cnf":         { "x5t#S256": "AbCdE...xyz" }
}
```

At the resource server, the workload connects over mTLS with the same X.509-SVID; the RS verifies that the certificate thumbprint matches the access token’s cnf.x5t#S256.

Acknowledgments

The author thanks participants in the OAuth Working Group for discussions on client instance identity, workload identity, and actor-based delegation that informed this document.

Author’s Address

Karl McGuinness
Independent
Email: public@karlmcguinness.com