

OAuth  
Internet-Draft  
Intended status: Standards Track  
Expires: 11 October 2026

R. Krishnan  
JPMorgan Chase & Co  
A. Prasad  
Oracle  
D. Lopez  
Telefonica  
S. Addepalli  
Aryaka  
9 April 2026

TLS-Session-Bound Access Tokens for OAuth 2.0  
draft-mw-oauth-tls-session-bound-tokens-04

Abstract

This document defines a mechanism for binding OAuth 2.0 access tokens to a specific mutual TLS (mTLS) connection. The binding is achieved through a proof token that incorporates the TLS Exporter value [RFC5705] derived from the current connection and an access token hash, signed by the client's private key corresponding to its mTLS certificate. This mechanism prevents stolen bearer tokens from being replayed on a different TLS connection. The proof is constructed once per (token, connection) pair and reused across all requests on that connection, delivering session binding with no per-request signing overhead and no additional key management beyond what mTLS already provides. The mechanism is applicable to TLS 1.2, TLS 1.3, and QUIC transports. While applicable to any OAuth 2.0 access token presented over mTLS, this specification is primarily motivated by the Token Exchange protocol [RFC8693], where multi-hop delegation chains in autonomous, agent-driven architectures create elevated replay risk.

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 11 October 2026.

## Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. The Bearer Token Replay Problem . . . . .	3
1.2. The Agentic AI Amplifier . . . . .	4
1.3. Conventions and Terminology . . . . .	5
2. TLS Session Binding for Access Tokens . . . . .	5
2.1. Overview . . . . .	5
2.2. TLS Exporter Derivation . . . . .	6
2.3. Session-Binding Proof Format . . . . .	7
2.3.1. Header . . . . .	7
2.3.2. Payload . . . . .	7
2.3.3. Signature . . . . .	8
2.4. Token Confirmation Claim . . . . .	8
3. Protocol Flow . . . . .	9
3.1. Token Issuance with Session Binding . . . . .	9
3.2. Authorization Server Behavior . . . . .	9
3.3. Resource Access with Session-Binding Proof . . . . .	10
3.3.1. Performance Characteristics . . . . .	11
3.3.2. Mid-Session Token Binding . . . . .	11
3.4. Token Introspection Considerations . . . . .	13
3.5. Error Responses . . . . .	13
4. Integration with Existing Mechanisms . . . . .	14
4.1. Relationship to RFC 8705 (mTLS-Bound Tokens) . . . . .	14
4.2. Relationship to RFC 9449 (DPoP) . . . . .	14
4.3. Relationship to WIMSE WIT/WPT . . . . .	15
4.4. Relationship to Transitive Attestation . . . . .	15
4.5. Relationship to RFC 8693 (Token Exchange) . . . . .	15
5. TLS Proxy Considerations . . . . .	16
5.1. Pass-Through mTLS . . . . .	16
5.2. TLS Termination at Proxy . . . . .	16
5.3. QUIC and HTTP/3 . . . . .	16

6.	Security Considerations . . . . .	17
6.1.	Threat Model . . . . .	17
6.1.1.	T1: Cross-Connection Token Replay . . . . .	17
6.1.2.	T2: Cross-Host Token Replay . . . . .	17
6.1.3.	T3: Token Exfiltration via LLM Prompt Injection . . . . .	18
6.1.4.	T4: Token + Proof Exfiltration (Both Stolen) . . . . .	18
6.1.5.	T5: Multi-Hop Delegation Chain Compromise . . . . .	18
6.2.	Residual Risks . . . . .	19
6.2.1.	Intra-Connection Replay . . . . .	19
6.2.2.	Compromised Private Key . . . . .	19
6.2.3.	TLS Proxy Trust . . . . .	19
7.	IANA Considerations . . . . .	19
7.1.	OAuth Token Confirmation Methods . . . . .	19
7.2.	OAuth Dynamic Client Registration Metadata . . . . .	20
7.3.	HTTP Header Fields . . . . .	20
7.3.1.	Session-Binding-Proof . . . . .	20
7.3.2.	TLS-Exporter . . . . .	20
7.4.	TLS Exporter Label . . . . .	20
8.	References . . . . .	20
8.1.	Normative References . . . . .	20
8.2.	Informative References . . . . .	21
Appendix A.	Contributors . . . . .	22
Appendix B.	Sidecar Deployment for Agentic AI . . . . .	22
B.1.	Architecture . . . . .	22
B.2.	Request Flow . . . . .	23
B.3.	HTTP/2 Multi-User Multiplexing . . . . .	25
B.4.	Security Benefits . . . . .	26
B.5.	Relationship to Existing Infrastructure . . . . .	26
B.6.	Implementation Considerations . . . . .	26
B.6.1.	Connection Lifecycle Management . . . . .	27
Authors' Addresses	. . . . .	27

## 1. Introduction

### 1.1. The Bearer Token Replay Problem

OAuth 2.0 access tokens are typically *\*bearer tokens\**: any party in possession of the token can use it to access protected resources, regardless of the presenter's identity or the communication channel. This is a known risk addressed by the OAuth 2.0 Security Best Current Practice [I-D.ietf-oauth-security-topics].

The Token Exchange protocol [RFC8693] amplifies this risk by enabling chained delegation across service boundaries. Each exchange produces a new bearer token, and a compromise at any point in the chain exposes downstream tokens.

Existing mitigations address parts of this problem:

- \* **\*RFC 8705 (mTLS Certificate-Bound Tokens)\* [RFC8705]:** Binds the token to the client's X.509 certificate thumbprint. However, the binding is to the `_certificate identity_`, not the `_TLS connection_`. If the same certificate is used across connections, or if the certificate and token are both exfiltrated, the token remains replayable.
- \* **\*RFC 9449 (DPoP)\* [RFC9449]:** Provides application-layer proof-of-possession using ephemeral, application-managed keys. Applicable to both public and confidential clients, but binds to the key, not to the TLS channel, and requires generating and managing a separate key pair.
- \* **\*Token Binding (RFC 8471-8473)\*:** Proposed direct TLS session binding but required a new TLS extension, was never specified for Token Exchange, encountered adoption barriers in browsers and TLS 1.3 transitions, and was ultimately abandoned.

None of these mechanisms provide **\*TLS-connection-level binding\*** for OAuth 2.0 access tokens in mTLS environments. This specification fills that gap by reusing the existing mTLS key pair (no additional key generation) and amortizing the proof to once per (token, connection) pair rather than once per request — delivering stronger binding than DPoP at lower per-request cost. The binding mechanism relies on TLS Exporter values, which are available in TLS 1.2, TLS 1.3, and QUIC (via its integrated TLS 1.3 handshake), making the specification transport-agnostic across modern encrypted transports.

## 1.2. The Agentic AI Amplifier

The rise of autonomous AI agents dramatically amplifies the bearer token replay risk. Unlike traditional OAuth flows where a human initiates discrete requests, agentic AI systems:

- \* **\*Autonomously chain API calls\*:** An agent may invoke hundreds of API calls across multiple services without human intervention, each potentially involving RFC 8693 token exchanges.
- \* **\*Perform multi-hop delegation\*:** Agent A exchanges its token for a delegated token to call Agent B, which exchanges again for Agent C. Each hop produces a new bearer token. A single compromise anywhere in this chain exposes all downstream tokens.
- \* **\*Run for extended periods\*:** Agents operate autonomously for hours or days, providing a broad window for token interception and replay.
- \* **\*Generate opaque traffic\*:** Agent-to-agent API calls are fully automated. A replayed token produces legitimate-looking traffic that is extremely difficult to distinguish from genuine requests—there is no human in the loop to detect anomalies.

- \* \*Are susceptible to prompt injection\*: A compromised or prompt-injected LLM agent can exfiltrate bearer tokens via tool calls, side channels, or log leakage. These tokens are immediately usable from any connection.

These characteristics make bearer token replay a *\*first-order threat\** in agentic AI architectures. As autonomous agents increasingly drive API-to-API traffic volumes that exceed human-initiated requests, per-request cryptographic overhead becomes a significant scaling concern. This document addresses this gap by binding access tokens to the mTLS connection on which they are presented, with proof amortization that scales to high-volume agent traffic.

### 1.3. Conventions and Terminology

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:

**TLS Exporter Value:** A value derived from the TLS handshake using the mechanism defined in [RFC5705] (for TLS 1.2) or Section 7.5 of [RFC8446] (for TLS 1.3). The exporter value is unique to the specific TLS connection and is available to both endpoints.

**Session-Binding Proof:** A signed JWT presented alongside the access token that cryptographically binds the token to the current mTLS connection via the TLS Exporter value.

**Token Exchange:** The protocol defined in [RFC8693] for exchanging one security token for another at an authorization server.

## 2. TLS Session Binding for Access Tokens

### 2.1. Overview

This specification defines a proof-of-possession mechanism that binds OAuth 2.0 access tokens to the mTLS connection on which they are presented. While applicable to any OAuth 2.0 access token, it is primarily designed for tokens issued via the Token Exchange protocol [RFC8693], where multi-hop delegation creates elevated replay risk. The mechanism operates as follows:

1. The client and resource server establish an mTLS connection. Both sides derive a TLS Exporter value unique to this connection.

2. When presenting an access token on a new connection, the client constructs a *\*Session-Binding Proof\**: a JWT containing the hash of the access token and the TLS Exporter value.
3. The client signs this JWT with the private key corresponding to its mTLS client certificate. The same proof MAY be reused for all subsequent requests using that token on the same connection.
4. The resource server verifies the proof by checking the signature against the client certificate's public key, confirming the exporter value matches the current connection, confirming the token hash matches the presented access token, and confirming the issuance time is within an acceptable window. The server MUST perform these checks on every request but MAY use a per-connection cache to reduce subsequent verifications to a cache lookup.

A token that requires session binding includes a confirmation method claim (`tls_exp`) containing the TLS Exporter label, which signals to the resource server that the Session-Binding Proof MUST be presented and verified.

## 2.2. TLS Exporter Derivation

Both the client and resource server MUST derive the TLS Exporter value using the following parameters:

- \* *\*Label\**: `EXPORTER-oauth-tls-session-bound`
- \* *\*Context\**: Empty (zero-length)
- \* *\*Length\**: 32 octets

For TLS 1.3, the exporter is derived as specified in Section 7.5 of [RFC8446]. For TLS 1.2, the exporter is derived as specified in [RFC5705]. TLS 1.3 is RECOMMENDED because TLS 1.2 abbreviated handshakes (session resumption) may reuse the same master secret across connections, weakening the binding.

Note: By binding to the TLS Exporter rather than the application traffic keys, the binding remains valid across TLS 1.3 KeyUpdate operations. Standard key rotation refreshes traffic keys but does not change the exporter master secret, avoiding unnecessary re-proof cycles while maintaining strong connection binding.

Note: This mechanism requires both the client and resource server to access the TLS Exporter derivation function from their TLS library. Most TLS libraries (OpenSSL, BoringSSL, Go crypto/tls) provide this API (e.g., `SSL_export_keying_material()`). However, some application frameworks and proxy platforms may not yet surface this function to their HTTP or extension layer. Implementors should verify that their TLS integration exposes exporter derivation to the layer responsible for constructing or verifying the Session-Binding Proof.

### 2.3. Session-Binding Proof Format

The Session-Binding Proof is a JWT with the following structure:

#### 2.3.1. Header

```
{
  "typ": "tls-binding-proof+jwt",
  "alg": "ES256",
  "x5t#S256": "<base64url SHA-256 thumbprint of client certificate>"
}
```

The `alg` value MUST match the key type of the client's mTLS certificate. The `x5t#S256` value MUST be the base64url-encoded SHA-256 thumbprint of the DER-encoded client certificate, as defined in [RFC8705].

#### 2.3.2. Payload

The payload contains REQUIRED connection-level claims that prove session binding, and OPTIONAL per-request claims for intra-session hardening.

```
{
  "ath": "<base64url SHA-256 hash of the access token>",
  "ekm": "<base64url TLS Exporter value>",
  "iat": 1710820000,
  "jti": "<unique identifier>",
  "htm": "POST",
  "htu": "/api/resource"
}
```

The payload claims are defined as follows:

##### 2.3.2.1. Connection-Level Claims (REQUIRED)

`ath`: REQUIRED. The base64url-encoded SHA-256 hash of the ASCII encoding of the associated access token value.

`ekm`: REQUIRED. The base64url-encoded TLS Exporter value derived as

specified in Section 2.2.

iat: REQUIRED. The time at which the proof was issued, as a NumericDate (seconds since the Unix epoch). The resource server MUST verify that this value is within an acceptable window.

#### 2.3.2.2. Per-Request Claims (OPTIONAL)

The following claims provide intra-session request-level binding. They are OPTIONAL and intended for deployments that require defense-in-depth against intra-connection replay. When omitted, the proof can be reused across all requests on the same connection for the same token.

jti: OPTIONAL. A unique identifier for the proof. When present, the resource server MUST verify that this value has not been seen before within the token's validity period.

htm: OPTIONAL. The HTTP method of the request to which the proof is attached (e.g., "GET", "POST"). When present, the resource server MUST verify that it matches the actual request method.

htu: OPTIONAL. The HTTP target URI of the request to which the proof is attached, without query and fragment parts. When present, the resource server MUST verify that it matches the actual request URI.

#### 2.3.3. Signature

The JWT MUST be signed using the private key corresponding to the client's mTLS certificate. The signature algorithm MUST match the alg header parameter.

#### 2.4. Token Confirmation Claim

An authorization server that supports TLS-session-bound access tokens MUST include a cnf (confirmation) claim in the issued access token (when the token is a JWT) or in the token introspection response. The cnf claim MUST contain:

```
{
  "cnf": {
    "x5t#S256": "<cert-thumbprint>",
    "tls_exp": "EXPORTER-oauth-tls-session-bound"
  }
}
```

x5t#S256: REQUIRED. The certificate thumbprint as defined in [RFC8705].

tls\_exp: REQUIRED. A string value containing the TLS Exporter label



that the client and resource server MUST use to derive the session-binding value. The presence of this claim signals that the resource server MUST require and verify a Session-Binding Proof whenever this token is presented. This follows the pattern of existing cnf members which carry key/binding material rather than boolean flags (see [RFC7800]).

### 3. Protocol Flow

#### 3.1. Token Issuance with Session Binding

The mechanism for requesting and issuing TLS-session-bound tokens is as follows:

1. The client authenticates to the authorization server using mTLS and requests a token. This MAY be a token exchange per [RFC8693], a client credentials grant, or any other OAuth 2.0 grant type.
2. The authorization server issues a new access token.
3. If the authorization server policy requires TLS session binding for this client, it includes the cnf claim with `tls_exp` set to the exporter label in the issued token.
4. The client receives the token and notes the `tls_exp` requirement.

#### 3.2. Authorization Server Behavior

The authorization server determines whether to issue TLS-session-bound tokens based on per-client configuration. The following client registration metadata parameter is defined:

`tls_session_bound_access_tokens`: OPTIONAL. A boolean value indicating that the authorization server MUST issue TLS-session-bound access tokens for this client. When set to true, the authorization server includes the `tls_exp` confirmation method in all access tokens issued to this client. Defaults to false.

The authorization server MAY also apply session binding based on:

- \* Token exchange policy: When the `subject_token` or `actor_token` in an RFC 8693 exchange is itself session-bound, the resulting token SHOULD also be session-bound to maintain the security property across the delegation chain.
- \* Resource server requirements: When a resource server's metadata indicates that it requires session-bound tokens.
- \* Risk-based policy: When the requested scope, audience, or delegation depth exceeds a policy threshold.

### 3.3. Resource Access with Session-Binding Proof

The following diagram illustrates the complete flow:

Client (with cert C)	Resource Server
<pre> === mTLS handshake (client cert C) =====&gt; </pre>	
<pre> [ONCE PER CONNECTION] Both sides derive:   EKM = TLS-Exporter(     "EXPORTER-oauth-tls-session-bound",     "", 32)  [ONCE PER (TOKEN, CONNECTION)] Client constructs proof JWT:   header = { typ, alg, x5t#S256 }   payload = {     ath: SHA256(access_token),     ekm: EKM,     iat: &lt;unix_timestamp&gt;   }   sig = Sign(C.privateKey, header    payload) </pre>	
<pre> --- HTTP Request 1 -----&gt;   Authorization: Bearer &lt;access_token&gt;   Session-Binding-Proof: &lt;proof_jwt&gt; </pre>	
<pre> Server verifies:   1. sig matches C.publicKey from mTLS   2. ath matches SHA256(access_token)   3. ekm matches server-derived EKM   4. iat within acceptable window   (caches: this token is bound to this conn) </pre>	
<pre> &lt;-- 200 OK ----- </pre>	
<pre> --- HTTP Request 2 (same token, same proof) -----&gt;   Authorization: Bearer &lt;access_token&gt;   Session-Binding-Proof: &lt;proof_jwt&gt; (reused) </pre>	
<pre> Server verifies (per request):   - looks up (conn_id, ath) in cache   - cache hit: binding verified for THIS conn </pre>	
<pre> &lt;-- 200 OK ----- </pre>	

### 3.3.1. Performance Characteristics

Because the Session-Binding Proof contains only connection-level claims (ekm, ath, iat), the client constructs and signs the proof \*once per (token, connection) pair\* and reuses it for all subsequent requests on that connection. The iat value reflects the time of proof construction, not the time of any particular request. This provides a significant advantage over DPoP:

- \* \*Proof construction (client-side)\*: One JWT signature per connection per token. DPoP requires one JWT signature per request.
- \* \*Proof verification (server-side)\*: Full JWT verification on first presentation; the server MAY cache the verified (connection, ath) binding and reduce subsequent verifications to a cache lookup. DPoP requires full JWT verification per request.
- \* \*TLS Exporter derivation\*: Once per connection, cached by both sides.
- \* \*No separate key management\*: Unlike DPoP, which requires generating, storing, and rotating an ephemeral key pair independent of TLS, this specification reuses the mTLS key pair. This eliminates an entire key lifecycle from the implementation.

When OPTIONAL per-request claims (jti, htm, htu) are included, the proof must be constructed per request (similar to DPoP). Deployments choose between per-connection efficiency and per-request intra-session hardening based on their threat model.

This amortization benefit assumes that mTLS connections are \*long-lived\* relative to the number of requests. If a deployment creates a new mTLS connection for every HTTP request-response cycle, the proof must be constructed on every request and the per-request cost becomes equivalent to DPoP. Deployments SHOULD use persistent connections (e.g., HTTP/2 or HTTP/1.1 with keep-alive) so that a single mTLS connection carries many requests. This is already standard practice in service mesh and workload-to-workload environments where mTLS is terminated at a sidecar or gateway.

### 3.3.2. Mid-Session Token Binding

A client MAY bind multiple tokens to a single mTLS connection concurrently. Each token requires its own Session-Binding Proof with a distinct ath (token hash) and the same ekm (TLS Exporter value) from the current connection.

This is common in workload-to-workload scenarios: a single mTLS connection between two microservices may carry requests on behalf of many different users, each with a distinct access token obtained via

separate token exchanges. For example, 100 users interacting with an AI agentic service that delegates to a downstream service — each user's token is bound to the same mTLS connection via a separate proof. Similarly, when a token is refreshed or rotated mid-session, the client constructs a fresh proof for the new token without requiring a new mTLS connection.

The resource server MAY cache verified (connection\_id, ath) bindings to avoid re-verifying proofs on subsequent requests. Each cache entry is small (a connection identifier plus a 32-byte hash), so even connections carrying hundreds of tokens have modest memory requirements. Caching is an optimization, not a requirement: if the server does not cache (or evicts entries under memory pressure), it simply falls back to full JWT signature verification on every request — equivalent to the per-request cost of DPOP.

When the client presents the access token to a resource server:

1. The client establishes an mTLS connection with the resource server.
2. The client derives the TLS Exporter value for the current connection (computed once and cached).
3. The client constructs a Session-Binding Proof JWT as specified in Section 2.3. If the proof does not contain per-request claims, the client MAY reuse the same proof for all subsequent requests using this token on this connection.
4. The client sends the HTTP request with:
  - \* The access token in the Authorization header: Authorization: Bearer <access\_token>
  - \* The Session-Binding Proof in the Session-Binding-Proof header: Session-Binding-Proof: <proof\_jwt>
5. The resource server performs the following verifications:
  - a. *\*Standard mTLS verification\**: Verifies the client certificate as part of the TLS handshake.
  - b. *\*Token validation\**: Validates the access token (signature, expiration, audience, etc.).
  - c. *\*Certificate binding\**: Verifies that the x5t#S256 in the token's cnf claim matches the presented client certificate.
  - d. *\*TLS binding required\**: Checks that cnf.tls\_exp is present and a Session-Binding Proof is present.
  - e. *\*Proof signature\**: Verifies the proof JWT signature against the public key in the client certificate.
  - f. *\*Exporter match\**: Compares the ekm claim in the proof against the TLS Exporter value for the current connection

and confirms they match. g. *\*Token hash\**: Computes SHA-256 of the presented access token and confirms it matches the ath claim. h. *\*Timestamp\**: Confirms iat is within an acceptable window. i. *\*Per-request claims\** (when present): Confirms htm and htu match the actual request, and jti has not been seen before.

The resource server MUST ensure that steps (a) through (i) are satisfied for every request. This is critical: an attacker who obtains a stolen token and proof may attempt to inject them on their own mTLS connection to the same resource server. Verification ensures the proof signature is checked against the certificate from `_this_` connection's handshake, which will not match the attacker's certificate.

To satisfy this requirement efficiently, the resource server MAY cache verified bindings keyed on (`connection_id`, `ath`). On subsequent requests with the same token on the same connection, a cache hit confirms the binding was already verified; a cache miss (different connection, different token, or first presentation) triggers full verification. This cache is inherently per-connection, so a stolen proof presented on a different connection will always miss and fail full verification. When a TLS connection terminates, the resource server SHOULD purge all cache entries associated with that connection, as the bindings are no longer valid.

6. If all verifications succeed, the resource server processes the request. If any verification fails, the resource server MUST reject the request as specified in Section 3.5.

### 3.4. Token Introspection Considerations

When token introspection [RFC7662] is used, the introspection response MUST include the `cnf` claim with the `tls_exp` field. This allows resource servers that do not have direct access to the token's claims (e.g., opaque tokens) to determine whether session binding is required.

### 3.5. Error Responses

When verification of the Session-Binding Proof fails or the proof is missing, the resource server MUST respond with HTTP 401 and include a WWW-Authenticate header with the appropriate error code:

```
WWW-Authenticate: Bearer error="invalid_proof",  
  error_description="Session-Binding Proof verification failed:  
  exporter mismatch"
```

```
WWW-Authenticate: Bearer error="use_session_binding",  
  error_description="This token requires a Session-Binding Proof"
```

The following error code values are defined:

`invalid_proof`: The Session-Binding Proof is malformed or failed verification. This includes signature verification failure, exporter mismatch, stale iat, jti reuse, ath mismatch, and htm/htu mismatch.

`use_session_binding`: The access token requires a Session-Binding Proof (the `cnf.tls_exp` claim is present) but no Session-Binding-Proof header was provided. This error signals to the client that it must construct and present a proof.

The resource server **SHOULD** include an `error_description` parameter with a human-readable explanation of the specific verification failure.

## 4. Integration with Existing Mechanisms

### 4.1. Relationship to RFC 8705 (mTLS-Bound Tokens)

This specification extends RFC 8705 by adding session-level binding on top of certificate binding. The `x5t#S256` claim from RFC 8705 is reused. Deployments **MAY** support both mechanisms simultaneously: RFC 8705 provides certificate binding, while this specification adds per-connection session binding.

### 4.2. Relationship to RFC 9449 (DPoP)

DPoP and this specification address similar goals (proof-of-possession) but differ in binding targets, key management, and per-request cost:

- \* **\*Binding target\***: DPoP binds tokens to an ephemeral application-layer key. This specification binds tokens to both the client identity (X.509 certificate) and the specific TLS connection (Exporter value). An attacker who exfiltrates a DPoP-bound token and the associated DPoP key can replay the token from any network location; an attacker who exfiltrates a session-bound token cannot, because reproducing the TLS Exporter value requires being on the same TLS connection.
- \* **\*Key management\***: DPoP requires generating, storing, and rotating an ephemeral key pair independent of TLS. This specification reuses the mTLS key pair already established during the handshake, eliminating the entire DPoP key lifecycle.

- \* **\*Per-request overhead\***: DPoP requires constructing and signing a new proof JWT for every HTTP request. This specification constructs the proof *\*once per (token, connection)\** and reuses it across all requests on that connection (when per-request claims are omitted). This is a fundamental efficiency advantage in high-throughput workload-to-workload scenarios.
- \* **\*Applicability\***: DPoP is particularly valuable for *\*public clients\** that cannot use mTLS. This specification is designed for *\*confidential clients and workloads\** that already use mTLS. The two mechanisms are complementary rather than competing.

In summary, for mTLS-capable environments, this specification provides stronger security (connection-level binding vs. key-level binding), lower per-request cost (amortized proof vs. per-request proof), and simpler implementation (no separate key management).

#### 4.3. Relationship to WIMSE WIT/WPT

The WIMSE Workload Identity Token (WIT) and Workload Proof Token (WPT) defined in [I-D.ietf-wimse-s2s-protocol] provide a similar proof-of-possession mechanism for workload-to-workload communication. This specification is compatible with WIMSE and can be used in conjunction with WIT/WPT when mTLS-based session binding is required for tokens obtained via RFC 8693 exchange.

#### 4.4. Relationship to Transitive Attestation

The Transitive Attestation profile [I-D.draft-mw-wimse-transitive-attestation] addresses a complementary problem: binding an identity to a verified execution environment ("Proof of Residency"). While this specification binds tokens to a TLS connection to prevent network-level replay, Transitive Attestation binds identities to a hardware-rooted host to prevent credential export. In high-assurance deployments, both mechanisms MAY be combined: Transitive Attestation ensures the token is used from the correct host, and TLS session binding ensures it is used on the correct connection.

#### 4.5. Relationship to RFC 8693 (Token Exchange)

This specification does not modify the token exchange protocol itself. The authorization server's token exchange endpoint continues to operate as specified in [RFC8693]. The session binding is applied to the `_resulting_` access token through the `cnf` claim. While this specification is applicable to any OAuth 2.0 access token, RFC 8693 Token Exchange is a primary motivator: each hop in a delegation chain produces a new bearer token, and session binding contains the blast

radius of any single token compromise to the specific TLS connection on which it is presented.

## 5. TLS Proxy Considerations

### 5.1. Pass-Through mTLS

In deployments where TLS is terminated at the application server (pass-through mode), this specification works without modification. Both the client and server have direct access to the TLS Exporter value.

### 5.2. TLS Termination at Proxy

When a TLS-terminating proxy (e.g., a load balancer or API gateway) sits between the client and the resource server, the proxy MUST forward the following information to the backend:

1. The client certificate or its thumbprint, as specified in [RFC9440].
2. The TLS Exporter value derived from the client-to-proxy mTLS session.

A new HTTP header is defined for conveying the exporter value:

TLS-Exporter: <base64url-encoded exporter value>

The proxy MUST derive the exporter using the label and parameters specified in Section 2.2 from the client-facing TLS session, and forward it in this header over a trusted, integrity-protected connection to the backend.

The backend resource server MUST use the forwarded exporter value (instead of its own locally-derived value) when verifying the Session-Binding Proof.

**\*Security Warning:** The TLS-Exporter header contains security-sensitive material. The connection between the proxy and backend MUST be integrity-protected (e.g., via a separate mTLS connection or a trusted network). The backend MUST NOT accept this header from untrusted sources.

### 5.3. QUIC and HTTP/3

This specification is directly applicable to QUIC ([RFC9000]) and HTTP/3 transports. QUIC integrates TLS 1.3 into its handshake ([RFC9001]), and TLS Exporter values are available for use with QUIC connections as specified in Section 4.2 of [RFC9001].



The following considerations apply when using this specification over QUIC:

- \* **\*Exporter derivation\***: The TLS Exporter MUST be derived using the 1-RTT exporter keys (post-handshake). Proofs MUST NOT be constructed using 0-RTT exporter values, as these are derived from pre-shared keys and provide weaker binding guarantees.
- \* **\*Connection migration\***: QUIC connections may migrate across network paths (different IP addresses and ports) without a new TLS handshake. The TLS state — and therefore the EKM — persists across migrations. Session-Binding Proofs remain valid after connection migration.
- \* **\*Client authentication\***: In TLS 1.3 over QUIC, post-handshake client authentication is not supported (Section 4.4 of [RFC9001]). Client certificates MUST be presented during the initial handshake. This simplifies the binding model: the client identity is fixed at connection establishment.
- \* **\*Long-lived connections\***: QUIC connections are designed to be long-lived and resilient to network changes, aligning naturally with the long-lived connection guidance in Section 3.3.1.

## 6. Security Considerations

This section addresses security considerations in addition to those described in the OAuth 2.0 Security Best Current Practice [I-D.ietf-oauth-security-topics].

### 6.1. Threat Model

The following table summarizes the threats addressed by this specification, the specific mechanism that mitigates each threat, and whether DPoP provides equivalent protection.

#### 6.1.1. T1: Cross-Connection Token Replay

- \* **\*Threat\***: An attacker intercepts a bearer token and presents it on a different TLS connection.
- \* **\*Mitigation\***: The ekm claim in the proof is derived from the TLS handshake transcript and is unique per connection. The resource server compares it against its own locally-derived EKM. A replayed proof will fail the EKM comparison.
- \* **\*DPoP equivalent\***: No. DPoP binds to an application-layer key, not to the TLS connection. If the DPoP key is also exfiltrated, replay succeeds.

#### 6.1.2. T2: Cross-Host Token Replay

- \* **\*Threat\***: An attacker on a different host obtains a bearer token and attempts to use it.
- \* **\*Mitigation\***: The proof is signed with the client's mTLS private key. The resource server verifies the signature against the public key from the current mTLS handshake. A different host presents a different certificate, so signature verification fails.
- \* **\*DPoP equivalent\***: Partial. DPoP binds to an ephemeral key, which may not be hardware-protected.

#### 6.1.3. T3: Token Exfiltration via LLM Prompt Injection

- \* **\*Threat\***: A compromised or prompt-injected AI agent exfiltrates a bearer token via tool calls, side channels, or log leakage.
- \* **\*Mitigation\***: The attacker obtains the token but cannot produce a valid proof. The mTLS private key resides in a separate process (e.g., a security sidecar, see Appendix B) or hardware module, inaccessible to the agent's LLM runtime. Even if the attacker also obtains the proof, the EKM will not match on a different connection.
- \* **\*DPoP equivalent\***: No. DPoP keys are application-layer and typically reside in the same process as the agent, making co-exfiltration with the token likely.

#### 6.1.4. T4: Token + Proof Exfiltration (Both Stolen)

- \* **\*Threat\***: An attacker obtains both the access token and the Session-Binding Proof, and attempts to use them on their own mTLS connection to the same resource server.
- \* **\*Mitigation\***: Two independent protections apply: (1) the attacker's mTLS certificate differs from the original client's, so the proof signature does not match the public key from the attacker's handshake — the resource server verifies the proof against the certificate presented on `_this_` connection; (2) even if the attacker somehow presents the same certificate, a different TLS connection produces a different EKM, so the ekm claim does not match the server's locally-derived value.
- \* **\*DPoP equivalent\***: No. If both the DPoP proof and the DPoP key are exfiltrated, the attacker can replay from any connection.

#### 6.1.5. T5: Multi-Hop Delegation Chain Compromise

- \* **\*Threat\***: A token is stolen at one hop in a delegation chain ( $A \rightarrow B \rightarrow C \rightarrow D$ ) and replayed at a different hop.
- \* **\*Mitigation\***: Each hop uses a distinct mTLS connection with a distinct EKM. A token bound to one hop's connection cannot be replayed on another.
- \* **\*DPoP equivalent\***: Partial. DPoP binds per key, but the key is not inherently tied to a specific hop's connection.

## 6.2. Residual Risks

### 6.2.1. Intra-Connection Replay

Within the same TLS connection, an attacker with access to the channel (e.g., a compromised middleware component) could observe and replay requests. This risk is mitigated by including the OPTIONAL per-request claims:

- \* The jti claim, which provides per-proof uniqueness when the server maintains a replay cache.
- \* The htm and htu claims, which bind the proof to a specific HTTP method and URI.
- \* Short iat validity windows that limit the temporal scope of any replay.

Deployments that require intra-connection replay protection SHOULD include per-request claims.

### 6.2.2. Compromised Private Key

If the client's mTLS private key is compromised, the attacker can produce valid proofs. This risk is mitigated by:

- \* Hardware-backed key storage (TPM, HSM, TEE).
- \* Short-lived certificates (e.g., SPIFFE SVIDs with 1-hour expiry).
- \* Transitive Attestation [I-D.draft-mw-wimse-transitive-attestation] for binding identity to a verified execution context.

### 6.2.3. TLS Proxy Trust

The TLS-Exporter header introduces a trust dependency on the proxy. A compromised proxy could forge exporter values. Mitigations include mutual authentication between proxy and backend, and restricting the header to trusted network segments.

## 7. IANA Considerations

### 7.1. OAuth Token Confirmation Methods

This specification registers the following confirmation method in the IANA "OAuth Token Confirmation Methods" registry established by [RFC7800]:

- \* \*Confirmation Method Value\*: tls\_exp
- \* \*Confirmation Method Description\*: TLS Exporter Session Binding
- \* \*Change Controller\*: IETF
- \* \*Reference\*: [this document]

## 7.2. OAuth Dynamic Client Registration Metadata

This specification registers the following client metadata value:

- \* \*Client Metadata Name\*: tls\_session\_bound\_access\_tokens
- \* \*Client Metadata Description\*: Boolean indicating the client requires TLS-session-bound access tokens
- \* \*Change Controller\*: IETF
- \* \*Reference\*: [this document]

## 7.3. HTTP Header Fields

This specification registers the following HTTP header fields:

### 7.3.1. Session-Binding-Proof

- \* \*Header Field Name\*: Session-Binding-Proof
- \* \*Status\*: permanent
- \* \*Reference\*: [this document]

### 7.3.2. TLS-Exporter

- \* \*Header Field Name\*: TLS-Exporter
- \* \*Status\*: permanent
- \* \*Reference\*: [this document]

## 7.4. TLS Exporter Label

This specification registers the following TLS Exporter label in the IANA "TLS Exporter Labels" registry:

- \* \*Value\*: EXPORTER-oauth-tls-session-bound
- \* \*DTLS-OK\*: N
- \* \*Recommended\*: Y
- \* \*Reference\*: [this document]

## 8. References

### 8.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", March 2010, <<https://www.rfc-editor.org/rfc/rfc5705>>.

- [RFC7662] Richer, J., "OAuth 2.0 Token Introspection", 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)", April 2016, <<https://www.rfc-editor.org/rfc/rfc7800>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC8693] Jones, M., Nadalin, A., Campbell, B., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", 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", February 2020, <<https://www.rfc-editor.org/rfc/rfc8705>>.
- [RFC9440] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", July 2023, <<https://www.rfc-editor.org/rfc/rfc9440>>.
- [RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", September 2023, <<https://www.rfc-editor.org/rfc/rfc9449>>.

## 8.2. Informative References

- [I-D.draft-mw-wimse-transitive-attestation]  
Krishnan, R., "Transitive Attestation for Sovereign Workloads: A WIMSE Profile", March 2026, <<https://datatracker.ietf.org/doc/html/draft-mw-wimse-transitive-attestation>>.
- [I-D.ietf-oauth-security-topics]  
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", April 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics>>.

[I-D.ietf-wimse-s2s-protocol]

Howard, P., "WIMSE Service to Service Authentication", 21 October 2024, <<https://datatracker.ietf.org/doc/html/draft-ietf-wimse-s2s-protocol>>.

[RFC9000] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.

[RFC9001] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", May 2021, <<https://www.rfc-editor.org/rfc/rfc9001>>.

## Appendix A. Contributors

The following individuals have contributed to this document:

Roman Zabicki  
JPMorgan Chase & Co  
Email: [roman.zabicki@chase.com](mailto:roman.zabicki@chase.com)

Jack McCallie  
JPMorgan Chase & Co  
Email: [jack.mccallie@chase.com](mailto:jack.mccallie@chase.com)

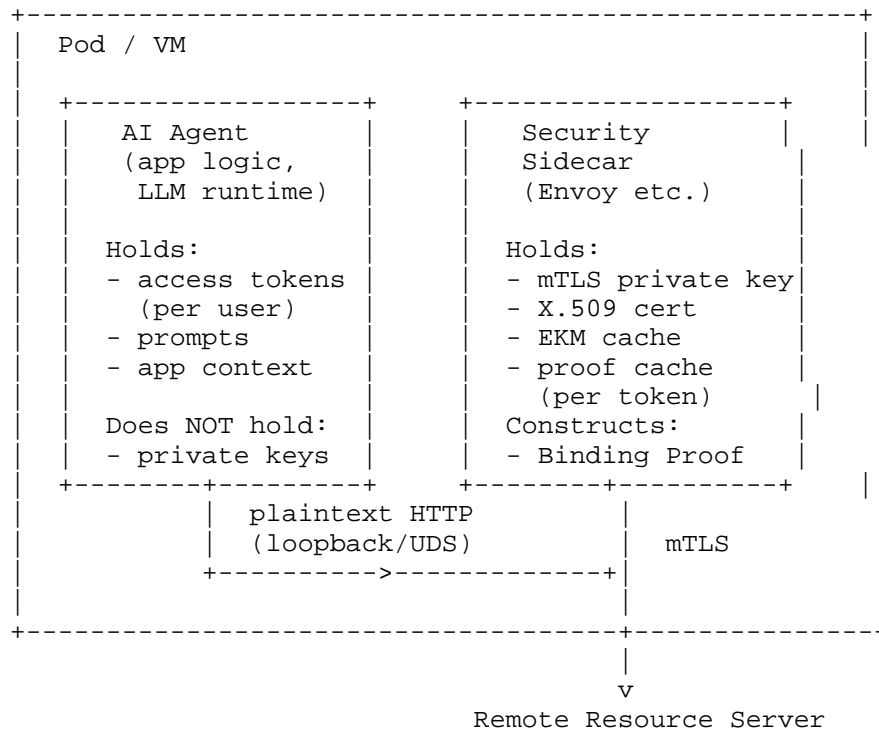
Jonian Musa JPMorgan Chase & Co  
Email: [jonian.musa@chase.com](mailto:jonian.musa@chase.com)

## Appendix B. Sidecar Deployment for Agentic AI

In agentic AI architectures, a common deployment pattern places a security sidecar (e.g., Envoy, Istio proxy, or a purpose-built agent gateway) alongside each AI agent workload. This appendix describes how TLS-session-bound tokens integrate with this pattern and the resulting security benefits.

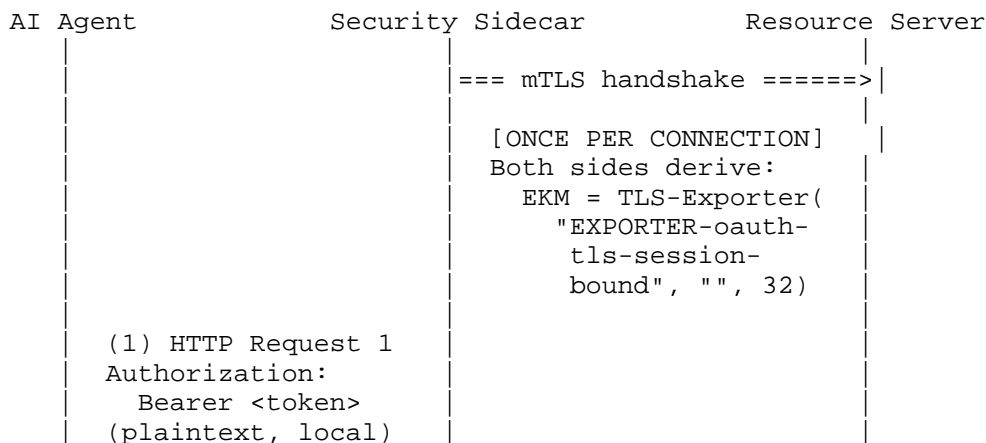
### B.1. Architecture

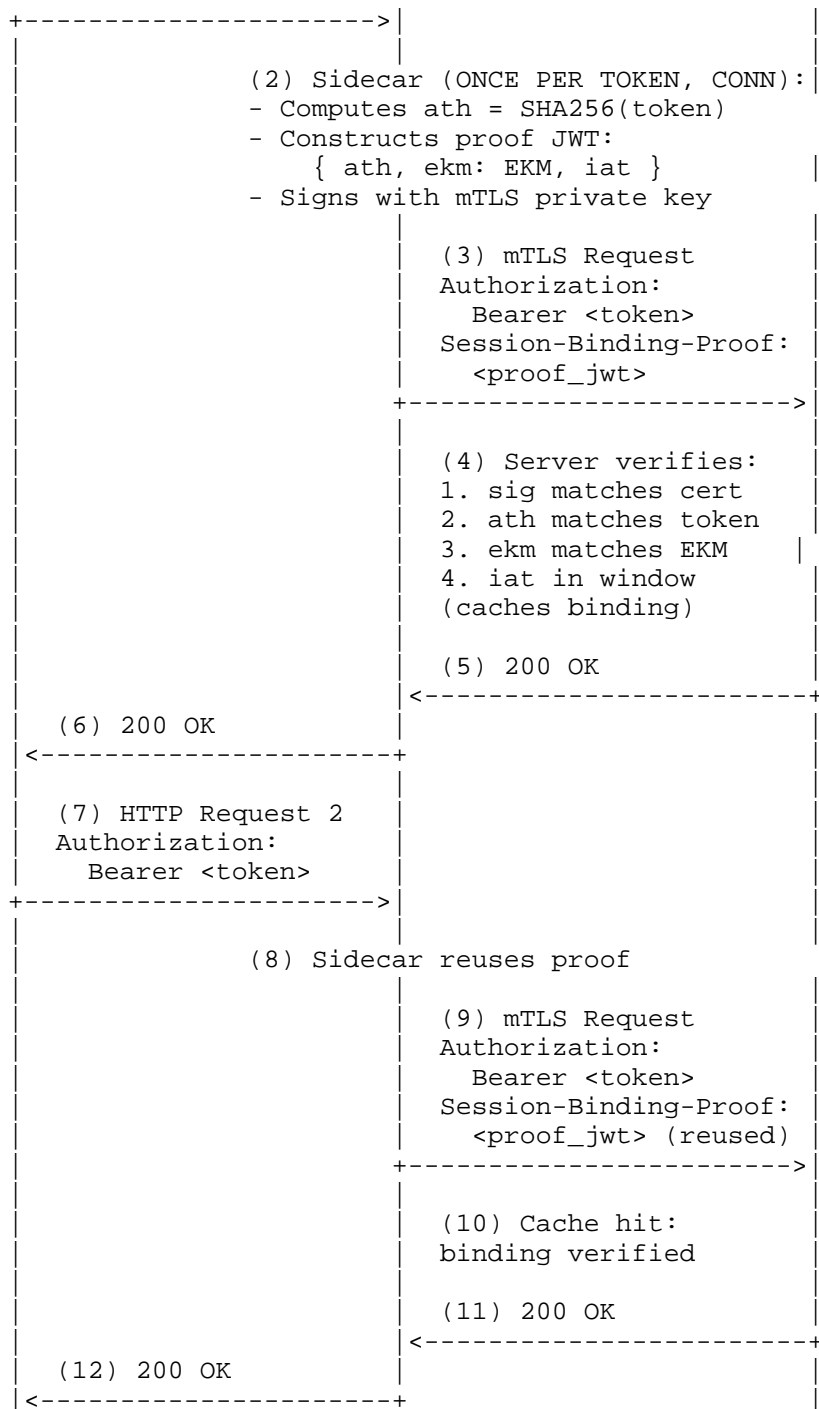
The following diagram shows the deployment layout. The AI agent and security sidecar run in the same pod or VM. The sidecar holds the mTLS private key, manages all authenticated outbound connections, and maintains a cache of signed proofs. A single mTLS connection to a remote resource server may carry requests on behalf of many different users, each with a distinct access token; the sidecar constructs and signs a proof once per (token, connection) pair and reuses it for subsequent requests with the same token.



## B.2. Request Flow

The following diagram shows how requests flow from the AI agent through the sidecar to the remote resource server. The sidecar transparently adds the Session-Binding Proof and reuses it for subsequent requests with the same token.

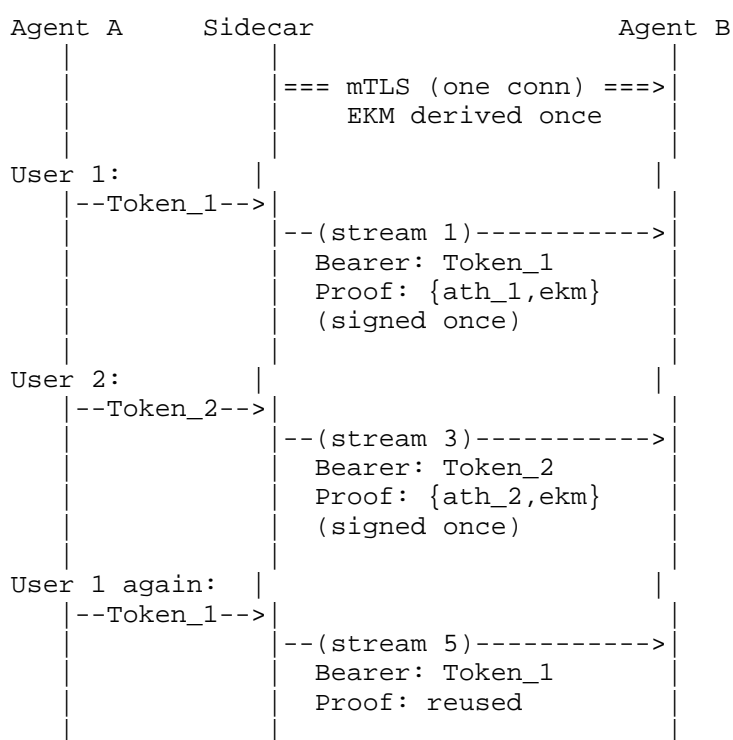






### B.3. HTTP/2 Multi-User Multiplexing

In a typical agentic deployment, Agent A serves multiple users concurrently. Each user's request triggers an on-behalf-of (OBO) token exchange, producing a distinct access token. All outbound requests to Agent B share a single HTTP/2 mTLS connection, multiplexed across streams:



Each token is signed *once* when first seen on this connection. Subsequent requests with the same token reuse the cached proof. With N users and M requests per user, the total signing cost is N (one per token) rather than N×M (one per request as with DPoP).

Note: Industry measurements indicate that AI-driven bot and agent traffic is growing rapidly and may exceed human-initiated web traffic within the next few years. In such environments, the per-request signing cost of mechanisms like DPoP becomes a significant scaling bottleneck. The per-connection amortization provided by this specification is designed to remain efficient at these traffic volumes.

#### B.4. Security Benefits

This architecture provides defense-in-depth against agentic AI threat vectors:

- \* **\*Prompt injection token exfiltration\***: Even if a compromised LLM exfiltrates an access token via tool calls, log leakage, or side channels, the attacker cannot produce a valid Session-Binding Proof. The mTLS private key resides exclusively in the sidecar process, which is not accessible to the agent's application logic or LLM runtime.
- \* **\*Key isolation\***: The agent never has access to the signing key. The sidecar can enforce hardware-backed key storage (TPM, HSM) independently of the agent's runtime environment.
- \* **\*Transparent integration\***: The agent application code requires no modifications beyond standard OAuth token handling. The session binding is entirely transparent—the sidecar intercepts outbound requests and adds the proof.
- \* **\*Centralized policy enforcement\***: The sidecar can apply additional policy checks (token scoping, rate limiting, destination allowlisting) before constructing the proof, providing a security control plane for agent traffic.
- \* **\*Audit boundary\***: All authenticated outbound traffic passes through the sidecar, providing a natural audit point for logging which tokens were used, to which destinations, and when.

#### B.5. Relationship to Existing Infrastructure

This deployment model aligns with the service mesh architecture used in SPIFFE/SPIRE environments, where the sidecar already manages workload identity certificates. When combined with Transitive Attestation [I-D.draft-mw-wimse-transitive-attestation], the sidecar can additionally attest that the agent is running in a verified execution environment while simultaneously binding all tokens to the active TLS connection.

#### B.6. Implementation Considerations

The sidecar must access the TLS Exporter value from the mTLS connection it terminates in order to construct the Session-Binding Proof. See Section 2.2 for the general implementation note on TLS Exporter API availability across TLS libraries and frameworks.

### B.6.1. Connection Lifecycle Management

Sidecars that implement this specification should handle TLS connection lifecycle events to manage the EKM and proof caches correctly. A typical implementation uses a layered architecture:

- \* *Connection-level layer*\* (e.g., a network filter or transport callback): Derives the EKM when a new mTLS connection is established and stores it in connection-scoped state. Registers a callback for connection close events.
- \* *Request-level layer*\* (e.g., an HTTP filter): Reads the cached EKM from the connection-scoped state, constructs or retrieves the cached proof for each token, and injects the Session-Binding-Proof header into outbound requests.

When a TLS connection terminates, all connection-scoped state — including the EKM and all cached proofs for that connection — SHOULD be purged automatically. Frameworks that support connection-scoped storage (e.g., per-connection filter state) provide this cleanup naturally without requiring explicit invalidation logic.

When a connection to the upstream resource server is lost and a new one is established, the sidecar derives a fresh EKM from the new handshake and constructs new proofs. Previously cached proofs are invalid because they contain the old connection's EKM.

#### Authors' Addresses

Ram Krishnan  
JPMorgan Chase & Co  
Email: ramkri123@gmail.com

A Prasad  
Oracle  
Email: a.prasad@oracle.com

Diego R. Lopez  
Telefonica  
Email: diego.r.lopez@telefonica.com

Srinivasa Addepalli  
Aryaka  
Email: srinivasa.addepalli@aryaka.com