SPICE
Internet-Draft
Intended status: Informational
Expires: 29 August 2026

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

25 February 2026

Cryptographically Verifiable Actor Chain for OAuth 2.0 Token Exchange
draft-mw-spice-actor-chain-00

Abstract

This document defines an extension to OAuth 2.0 Token Exchange [[RFC8693]] that addresses the problem of *Delegation Auditability Gaps* in multi-hop service environments. Current standards treat prior actors in a delegation chain as "informational only," providing no cryptographic proof of the actual delegation path. This document proposes a new actor_chain claim — a *Cryptographically Verifiable Actor Chain* — that replaces the informational-only nested act claim with a tamper-evident, ordered record of all actors. This solution enables high-assurance data-plane policy enforcement and forensic auditability, particularly for dynamic AI agent-to-agent workloads—where susceptibility to *prompt injection attacks* can lead to unauthorized delegation paths — the security posture of the entire delegation chain is critical for authorization decisions.

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 29 August 2026.

Table of Contents

1. Introduction

This document defines an extension to OAuth 2.0 Token Exchange [[RFC8693]] to support high-assurance *East-West* identity delegation through cryptographically verifiable actor chains.

In modern multi-service and AI-agent environments, a workload often delegates its authority to another agent, which may in turn delegate to others. While [[RFC8693]] provides the act (actor) claim to represent delegation, it explicitly restricts prior actors in a nested chain to be "informational only," excluding them from access control considerations. This creates a significant *Delegation Auditability Gap*: Resource Servers cannot verify the full path of authority, and attackers can potentially hide lateral movement or *prompt injection-induced hijacking* within unverified informational claims.

By providing a standardized *Cryptographically Verifiable Actor Chain*, this extension replaces the informal nested act structure with a policy-enforceable, ordered, and tamper-evident record of all participants. This establishes an *"East-West"* axis of accountability, ensuring that any service in a global delegation chain can be verified for identity, integrity, and (optionally) physical residency.

This solution addresses several critical gaps in [[RFC8693]]:

1. *Cryptographic Audit Trail*: Proves that each prior actor actually participated in the delegation chain and that the sequence has not been tampered with.
2. *Data-Plane Policy Enforcement*: Enables Resource Servers to write fine-grained authorization policies based on any actor in the path (e.g., "originating actor must be X").
3. *Dynamic AI Agent Topologies*: Provides a scalable architecture for the unpredictable and deep delegation chains common in autonomous agent networks.

4. *Data-Plane Efficiency*: Uses a flat, ordered array structure optimized for high-throughput parsing and indexing in cloud-native proxies (e.g., Envoy).

This extension is designed to be backward-compatible and format-agnostic, supporting both JSON/JWS (JWT [[RFC7519]]) and CBOR/COSE (CWT [[RFC8392]]) representations.

2. 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 leverages the terminology defined in OAuth 2.0 Token Exchange [[RFC8693]], the SPICE Architecture [[!I-D.ietf-spice-arch]], and the RATS Architecture [[RFC9334]].

Actor Chain: A Cryptographically Verifiable Actor Chain — an ordered sequence of Actor Chain Entries representing the complete delegation path from the originating actor to the current actor. The chain is integrity-protected either by the AS's JWT signature (AS-Attested Mode) or by per-actor cryptographic signatures (Self-Attested Mode).

Actor Chain Entry: A JSON object or CBOR map identifying a single actor in the delegation chain, including its identity claims and a cryptographic signature binding it to the chain state at the point of its participation.

Chain Digest: A cumulative cryptographic hash computed over the delegation history. For any entry at index N, the chain_digest is computed over the concatenation of the current entry's identity claims and the chain_digest of the preceding entry (at index N-1). For the entry at index 0, the digest is computed over that entry's identity claims alone. Each actor produces a per-actor signature over its chain_digest, cryptographically anchoring itself to the entire preceding history.

Chain Depth: The total number of Actor Chain Entries in an actor chain. Used by policy engines to enforce maximum delegation depth.

Proof of Residency (PoR): A cryptographic proof (as defined in [[!I-D.draft-mw-spice-transitive-attestation]]) binding a workload to a specific, verified local environment. When present in an Actor Chain Entry, it provides hardware-rooted assurance of the actor's execution context.

3. The Problem: RFC 8693 Actor Limitations

3.1. Single-Actor Semantics

[[RFC8693]] Section 4.1 defines the act claim as a JSON object identifying *the* current actor. While nesting is permitted to represent prior actors, the specification explicitly limits their utility. Only the outermost act claim—representing the current actor—is relevant for access control. All prior actors exist solely for informational purposes.

This design was appropriate for traditional web service delegation where chains are short (typically one or two hops) and the identity of the immediate caller is sufficient for authorization. It is insufficient for the emerging class of workloads described below.

3.2. AI Agent Delegation Chains

Modern AI systems increasingly operate as networks of specialized agents. A typical interaction may involve:

```
User -> Orchestrator Agent -> Planning Agent -> Tool Agent -> Data API
```

At each hop, the agent performs a token exchange ([[RFC8693]]) to obtain credentials appropriate for calling the next service. Under current [[RFC8693]] semantics, by the time the request reaches the Data API, only the Tool Agent is identified as the actor. The Orchestrator Agent and Planning Agent—which may have been manipulated via *prompt injection* into delegating authority they should not have—are invisible to policy enforcement.

This creates several concrete risks:

* *Lateral Movement*: A compromised agent deep in the chain can impersonate the authority of the originating actor without any cryptographic evidence of the actual delegation path.
* *Policy Bypass*: Fine-grained policies like "only allow data access when the orchestrator is a known, trusted entity" cannot be expressed because the orchestrator’s identity is not available for policy evaluation at the data plane.
* *Audit Gaps*: Post-incident forensic analysis cannot reliably reconstruct the delegation path because the nested act claims are self-reported and unsigned.

3.3. Structural Limitations of Nested act

Beyond the semantic restriction, the nested object structure of act in [[RFC8693]] has practical limitations:

1. *Parsing Complexity*: Each prior actor requires traversing one additional level of JSON nesting. In high-throughput data-plane proxies (e.g., Envoy, Istio sidecars), deep nesting imposes parsing overhead.
2. *Indexing*: It is not possible to efficiently query "the actor at position N" without recursively unwinding the nested structure.
3. *Size Predictability*: The depth of nesting is unbounded, making it difficult to predict token sizes and allocate parsing buffers.
4. *No Integrity*: Each nested act is a plain JSON object with no signature or hash binding. Any intermediary could insert, remove, or reorder prior actors without detection.

3.3.1. Selective Disclosure (SD-JWT)

4. The Solution: The Cryptographically Verifiable actor_chain Claim

4.1. Overview

This document defines a new claim, actor_chain, that provides a Cryptographically Verifiable Actor Chain. When used in a JWT, its value is a JSON array of Actor Chain Entries. When used in a CWT, its value is a CBOR array of Actor Chain Entries. The array is ordered chronologically: index 0 represents the originating actor, and the last index represents the current actor.

The actor_chain claim supports two operational modes:

1. *AS-Attested Mode*: The Authorization Server (AS) validates each actor at token exchange time and constructs the actor_chain. The AS's signature over the entire token (JWS or COSE) provides integrity protection for the chain. Per-actor chain_sig fields are omitted.

2. *Self-Attested Mode*: Each Actor Chain Entry additionally includes a chain_sig field: a cryptographic signature (compact JWS [[RFC7515]] or COSE_Sign1 [[RFC9052]]) computed by that actor over a Chain Digest of all preceding entries. This creates a hash-chain structure providing tamper evidence and non-repudiation independent of the AS.

4.2. Claim Definition

The actor_chain claim is a JSON array. Each element of the array is a JSON object (an Actor Chain Entry) with the following members. To support diverse privacy requirements, Selective Disclosure (SD-JWT) is configurable at two levels of granularity: - *Per-Actor*: An Authorization Server MAY choose to hide entire Actor Chain Entries or only specific actors in the chain. - *Per-Field*: Within a single

Actor Chain Entry, a subset of members (e.g., sub) MAY be hidden using an _sd claim while others (e.g., iat or por) remain in cleartext. These fields typically correspond to the identity claims present in the *Actor Token* or client credentials used during the token exchange flow.

sub: REQUIRED (or selectively disclosed). A string identifying the actor, as defined in [[RFC7519]] Section 4.1.2.

iss: REQUIRED (or selectively disclosed). A string identifying the issuer of the actor's identity, as defined in [[RFC7519]] Section 4.1.1.

iat: REQUIRED (or selectively disclosed). The time at which this actor was appended to the chain, represented as a NumericDate as defined in [[RFC7519]] Section 4.1.6.

por: OPTIONAL. A JSON object containing a Proof of Residency binding this actor to a verified execution environment. The structure of this object is defined in [[!I-D.draft-mw-spice-transitive-attestation]].

chain_digest: OPTIONAL. A Base64url-encoded cumulative cryptographic hash (SHA-256). For any entry at index N, the hash is computed over the canonical serialization of the union of the current entry's identity claims (e.g., sub, iss, iat, _sd hashes) and the chain_digest of the preceding entry (index N-1). For the entry at index 0, the hash is computed over its identity claims alone. This recursive structure ensures that a chain_sig at any point in the chain provides proof of participation for all prior actors. REQUIRED in Self-Attested Mode. MUST be omitted in AS-Attested Mode.

> [!IMPORTANT] When Selective Disclosure is used, the SD-JWT *Disclosure strings* (the cleartext salts/values) MUST NOT be included in the canonical serialization used for hashing. The chain_digest is computed exclusively over the Actor Chain Entry object, which contains the stable _sd hashes. This ensures the signature remains valid regardless of which disclosures are subsequently provided to different recipients.

chain_sig: OPTIONAL. A compact JWS [[RFC7515]] signature produced by this actor's private key over the chain_digest value. The JWS header MUST include the jwk or kid member to identify the signing key. REQUIRED in Self-Attested Mode. MUST be omitted in AS-Attested Mode.

chain_mode: OPTIONAL. A top-level string claim (sibling to actor_chain) indicating the operational mode. Values are as_attested or self_attested. If omitted, the mode is inferred from the presence or absence of chain_sig fields in the Actor Chain Entries.

4.3. Example Tokens

4.3.1. AS-Attested Mode

In this mode, the AS constructs the actor_chain array and the JWT's own signature provides integrity. No per-entry chain_sig or chain_digest fields are present. This is the simplest deployment model.

```
{
  "aud": "https://data-api.example.com",
  "iss": "https://auth.example.com",
  "exp": 1700000100,
  "nbf": 1700000000,
  "sub": "user@example.com",
  "actor_chain": [
    {
      "sub": "https://orchestrator.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000010,
      "por": {
        "wia_kid": "spiffe://example.com/wia/node-1",
        "env_hash": "sha256:abc123..."
      }
    },
    {
      "sub": "https://planner.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000030
    },
    {
      "sub": "https://tool-agent.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000050,
      "por": {
        "wia_kid": "spiffe://example.com/wia/node-3",
        "env_hash": "sha256:ghi789..."
      }
    }
  ]
}
```

The integrity of the chain rests on the AS's JWT signature. The Relying Party trusts the AS to have correctly validated each actor at the time of each token exchange.

4.3.2. Self-Attested Mode

In this mode, each Actor Chain Entry additionally carries a chain_digest and chain_sig, forming a hash chain with per-actor non-repudiation. This mode is appropriate for federated deployments, multi-AS environments, or when Relying Parties require cryptographic proof of each actor's participation independent of any single AS.

```
{
  "aud": "https://data-api.example.com",
  "iss": "https://auth.example.com",
  "exp": 1700000100,
  "nbf": 1700000000,
  "sub": "user@example.com",
  "actor_chain": [
    {
      "sub": "https://orchestrator.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000010,
      "por": {
        "wia_kid": "spiffe://example.com/wia/node-1",
        "env_hash": "sha256:abc123..."
      },
      "chain_digest": "sha256:mno345...",
      "chain_sig": "eyJhbGciOiJFUzI1NiIsImt..."
    },
    {
      "sub": "https://planner.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000030,
      "chain_digest": "sha256:def456...",
      "chain_sig": "eyJhbGciOiJFUzI1NiIsImt..."
    },
    {
      "sub": "https://tool-agent.example.com",
      "iss": "https://auth.example.com",
      "iat": 1700000050,
      "por": {
        "wia_kid": "spiffe://example.com/wia/node-3",
        "env_hash": "sha256:ghi789..."
      },
      "chain_digest": "sha256:jkl012...",
      "chain_sig": "eyJhbGciOiJFUzI1NiIsImt..."
    }
  ]
}
```

In the Self-Attested example:

* *Index 0* (Orchestrator): The originating actor. Its chain_digest is SHA-256(canonical_json({sub, iss, iat})) — a self-hash of its own identity claims. Its chain_sig is computed over this digest. It includes a PoR binding it to node-1.
* *Index 1* (Planning Agent): Its chain_digest is SHA-256(canonical_json(actor_chain[0])). Its chain_sig is produced by the Planning Agent's key over this digest. No PoR is present (the agent may be running in a non-TEE environment).
* *Index 2* (Tool Agent): Its chain_digest is SHA-256(canonical_json(actor_chain[0..1])). Its chain_sig is produced by the Tool Agent's key over this digest. It includes a PoR for node-3.

4.4. Token Exchange Flow

When an actor (Service B) receives a token containing an actor_chain and needs to call a downstream service (Service C), the following token exchange flow occurs:

1. *Service B* sends a token exchange request to the Authorization Server (AS) per [[RFC8693]] Section 2.1.
2. The subject_token contains the existing actor_chain.
3. The actor_token identifies Service B.
4. The AS validates the existing actor_chain:
    * In Self-Attested Mode: verifies each chain_sig against the corresponding actor's public key and each chain_digest against the hash of preceding entries.
    * In AS-Attested Mode: verifies the JWT signature and validates the actor identities through its own policy (e.g., client registration, mTLS certificate).
    * In both modes: enforces any max_chain_depth policy.
5. The AS constructs a new actor_chain for the issued token by:
    * Copying all existing Actor Chain Entries from the subject_token.
    * Appending a new Actor Chain Entry for Service B.
    * In Self-Attested Mode: the new entry's chain_digest is computed over all preceding entries, and its chain_sig is produced by Service B's key (provided via the actor_token or client credentials).
    * In AS-Attested Mode: the new entry contains only the identity claims (sub, iss, iat) and optional por.
6. The AS issues a new token with the extended actor_chain.

4.4.1. Disclosure Propagation in SD-JWT

When Selective Disclosure (SD-JWT) [[!I-D.ietf-oauth-selective-disclosure-jwt]] is used for any Actor Chain Entry, the Authorization Server (AS) acts as the Discloser during token exchange. The propagation of the underlying cleartext identities is managed as follows:

1. *Disclosure Inclusion*: When the AS issues a token for a downstream recipient (Service B), it determines based on policy which hidden identifiers in the actor_chain should be visible to that recipient. For each visible actor, the AS appends the corresponding SD-JWT Disclosure string (containing the salt and cleartext value) to the issued token.
2. *Hop-by-Hop Visibility*: In a chain a -> b -> c -> d -> e, if a is to be visible to b, c, and d, the AS includes a's disclosure string in the tokens issued to b, c, and d during their respective token exchanges.
3. *Selective Redaction*: When the final exchange occurs for recipient e (the end of the trust zone), the AS simply omits the disclosure string for a from the issued token.
4. *Validation*: Each recipient in the chain can independently verify the hash of any disclosed claim against the _sd array in the Actor Chain Entry. If a disclosure is missing, the recipient sees only the hash, but can still verify the entry's geographic and residency properties through other claims.

This mechanism ensures that the "key" to unlock a hidden identity is passed only to authorized actors in the chain, while the underlying cryptographically signed JWT structure remains identical for all recipients.

4.4.2. When Chain Signatures Are Produced

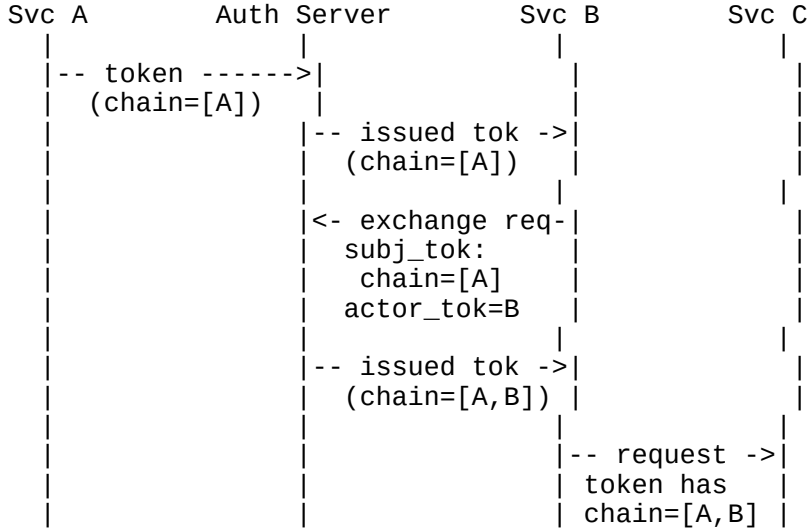
In Self-Attested Mode, a critical distinction is that each actor's chain_sig is produced *during the token exchange request*, before the final issued token exists. Consider a scenario where Service A calls Service B, and Service B later needs to call Service C:

1. Service A calls Service B, presenting a token containing actor_chain with chain [A].
2. Service B receives the token and performs its work.
3. *When Service B needs to call downstream Service C*, it initiates a token exchange with the AS. At this point — not before — Service B:
   * Computes chain_digest over the existing chain entries from the received token.
   * Signs this chain_digest with its own key, producing chain_sig.

   * Submits both values to the AS as part of the token exchange request, alongside the subject_token (containing chain [A]) and its own actor_token.

4. The AS validates the existing chain, then assembles the new actor_chain (prior entries + Service B's signed entry) into the issued JWT.
5. The AS signs the *entire JWT* (including the actor_chain) with the AS's own key and returns the token to Service B.
6. Service B uses this new token (containing chain [A, B]) to call Service C.

Consequently, each chain_sig covers only the *actor chain state* at that actor's point of participation — not the enclosing JWT's other claims such as aud, exp, or sub. This is by design: the actor chain and the token are concerns of different parties. The actor signs the chain to prove its participation; the AS signs the token to assert the token's validity. This separation allows the AS to set audience, expiry, and other claims independently without invalidating any actor's chain signature.



4.5. Data-Plane Policy Enforcement

Unlike the nested act claim in [[RFC8693]], the actor_chain claim is explicitly designed to be used in access control decisions. Resource Servers and data-plane proxies MAY apply authorization policies based on any entry in the actor chain.

4.5.1. Policy Examples

The following are illustrative examples of policies that become expressible with the actor_chain claim:

Origin-Based Policy: Allow access only if the originating actor (index 0) is a trusted orchestrator:

```
actor_chain[0].sub == "https://orchestrator.example.com"
```

Domain Restriction: Deny access if any actor in the chain belongs to an untrusted domain:

```
for_all(entry in actor_chain):
  entry.iss in ["https://auth.example.com",
                "https://auth.partner.com"]
```

Chain Depth Limit: Reject tokens with delegation chains longer than a configured maximum:

```
len(actor_chain) <= 5
```

Residency Requirement: Require that all actors in the chain have a valid Proof of Residency:

```
for_all(entry in actor_chain):
  entry.por is present AND entry.por is valid
```

Path-Based Policy: Allow access only through a specific delegation path:

```
actor_chain[0].sub == "https://orchestrator.example.com" AND
actor_chain[1].sub == "https://planner.example.com"
```

4.5.2. Integration with Data-Plane Proxies

The flat array structure of actor_chain is designed for efficient processing by data-plane proxies such as Envoy, Istio sidecars, and API gateways. Proxies can:

1. Extract the actor_chain array from the JWT payload with a single JSON path expression.
2. Iterate linearly over the entries without recursive descent.
3. Index specific entries by position (e.g., actor_chain[0] for the originator).
4. Compute len(actor_chain) for depth-based policies without parsing nested structures.

5. Emit structured log entries per Actor Chain Entry for distributed tracing and forensic analysis.

5. Chain Integrity Verification

A Relying Party receiving a token with the actor_chain claim MUST perform the following verification steps:

1. *JWT Signature Verification*: Verify the outer JWT signature per standard JWT processing rules. This is REQUIRED in both modes and provides baseline integrity for the entire token, including the actor_chain.

2. *Structural Validation*: Verify that actor_chain is a JSON array with at least one element. Verify that each element contains the required identity fields (sub, iss, iat).

3. *Per-Entry Signature Verification* (Self-Attested Mode only). For each entry at index i:

   * *Compute expected digest*: If i == 0, compute expected_digest = SHA-256(canonical_json({sub, iss, iat})) from the entry's own identity claims. If i > 0, compute expected_digest = SHA-256(canonical_json(actor_chain[0..i-1])).
   * *Verify chain_digest*: Confirm that the entry's chain_digest matches expected_digest.
   * *Verify chain_sig*: Verify chain_sig against chain_digest using the actor's public key.

4. *PoR Verification* (if present):

   * Verify each PoR assertion according to [[!I-D.draft-mw-spice-transitive-attestation]].

5. *Policy Evaluation*:

   * Apply local authorization policy against the verified actor chain.

If any verification step fails, the Relying Party MUST reject the token.

6. Relation to Other IETF Work

This proposal extends and complements several ongoing efforts:

Specification	Relationship
RFC 8693 [[RFC8693]]	This document extends [[RFC8693]] by defining actor_chain as a replacement for the informational-only nested act claim. The actor_chain claim is backward-compatible: an AS MAY populate both act (for legacy consumers) and actor_chain (for chain-aware consumers).
Transitive Attestation [[!I-D.draft-mw-spice-transitive-attestation]]	Provides the "North-South" residency proof (agent to local WIA) that complements the "East-West" delegation proof (agent to actor-chain) provided by this document.
SPICE Architecture [[!I-D.ietf-spice-arch]]	Defines the overarching workload identity architecture within which this extension operates.
WIMSE Architecture [[!I-D.ietf-wimse-arch]]	This proposal aligns with the WIMSE delegation and impersonation patterns for distributed microservices architectures.
Attestation-Based Auth [[!I-D.ietf-oauth-attestation-based-client-auth]]	Provides the client-to-AS attestation mechanism that can be leveraged to populate the hardware-rooted por claims in Actor Chain Entries.
SCITT [[!I-D.ietf-scitt-architecture]]	Verifiable actor chains can be recorded in SCITT transparency logs to provide long-term, tamper-proof auditability of delegation paths.
RATS [[RFC9334]]	Provides the attestation foundation for PoR assertions embedded in Actor Chain Entries.
DPoP [[RFC9449]]	actor_chain complements DPoP by providing delegation-chain context alongside proof-of-possession.

Table 1

6.1. East-West vs. North-South Security

This specification addresses the *East-West* axis of agent-to-agent communication, providing a cryptographically verifiable trail of identity delegation across a network of services. In contrast, Transitive Attestation [[!I-D.draft-mw-spice-transitive-attestation]] addresses the *North-South* axis of an agent's relationship with its local hosting environment (e.g., a Workload Identity Agent on a TEE-enabled node).

By embedding "North-South" Proofs of Residency (PoR) within "East-West" Actor Chain Entries, a Relying Party gains end-to-end assurance that every entity in a global delegation chain is both a recognized identity and is executing within a verified, secure environment.

6.2. Backward Compatibility with RFC 8693

An Authorization Server implementing this extension SHOULD populate both the act claim (per [[RFC8693]] Section 4.1) and the actor_chain claim in issued tokens. This ensures that:

* *Legacy consumers* that understand only act continue to function correctly, seeing the current actor in the top-level act claim.
* *Chain-aware consumers* can use actor_chain for fine-grained policy enforcement and audit.

The act claim, when present alongside actor_chain, MUST identify the same entity as the last entry in the actor_chain array.

7. Security Considerations

7.1. Token Signature vs. Per-Entry Chain Signatures

A natural question arises: since the JWT containing the actor_chain is already signed by the Authorization Server (AS), is per-entry chain_sig redundant?

The answer depends on what each signature proves and when it is produced:

* The *JWT outer signature* is produced by the AS when the final token is issued. It proves that the AS issued this specific token with this specific payload. It covers the entire token: sub, aud, exp, actor_chain, and all other claims.

* Each *chain_sig* is produced by an individual actor during token exchange, before the final token exists (see Section "When Chain Signatures Are Produced"). It covers only the chain_digest — the hash of the actor chain state at that actor's point of participation. It proves that this specific actor participated in this specific chain.

The JWT outer signature does NOT prove that each individual actor actually participated in the delegation. A compromised or malicious AS could construct any chain it desires. The two modes address different trust assumptions:

* *AS-Attested Mode* is appropriate when all participants trust the AS to faithfully record the chain. The AS validates each actor at exchange time, and its JWT signature provides integrity. This is the common single-organization deployment where the AS is a trusted internal service (e.g., a corporate identity provider). Per-actor signatures are omitted, keeping token sizes smaller and reducing cryptographic overhead.

* *Self-Attested Mode* is appropriate when the chain crosses trust boundaries or when stronger guarantees are needed:

  - *Federated/Multi-AS environments*: The token may be issued by AS-1 but consumed by a Relying Party that trusts AS-2. Per-actor signatures allow the RP to verify actor participation independently of the issuing AS.
  - *Non-repudiation*: Each actor's signature provides cryptographic evidence that it participated in the chain — evidence that the actor cannot deny and that the AS cannot fabricate.
  - *Zero-trust posture*: In adversarial environments, the Relying Party may not fully trust any single AS. Per-actor signatures provide defense-in-depth.
  - *Forensic analysis*: Post-breach investigations can verify the chain using each actor's public key, independent of the AS's continued availability or trustworthiness.

Deployments SHOULD select the mode that matches their trust model. An AS MAY enforce a specific mode via policy.

7.2. Chain Integrity

In AS-Attested Mode, chain integrity is provided by the JWT outer signature. The Relying Party trusts the AS to have correctly constructed the chain.

In Self-Attested Mode, the hash-chain structure provides additional tamper evidence. Insertion, deletion, or reordering of entries invalidates the chain_digest and chain_sig fields of all subsequent entries. An attacker who compromises a single actor in the chain cannot retroactively alter the entries of prior actors without possessing their signing keys.

7.3. Replay Protection

Each Actor Chain Entry includes an iat (issued-at) timestamp. Relying Parties SHOULD enforce a maximum age on Actor Chain Entries to prevent replay of stale chains. Additionally, the standard JWT claims exp and nbf on the enclosing token provide overall token-level freshness.

7.4. Chain Depth Limits

Unbounded actor chains pose a risk of token size explosion and processing overhead. Authorization Servers SHOULD enforce a configurable maximum chain depth (max_chain_depth). A RECOMMENDED default maximum is 10 entries. Relying Parties MAY independently enforce their own chain depth limits.

7.5. Key Management

Each actor in the chain signs its entry with its own key. The security of the entire chain depends on the security of each actor’s key material. Actors SHOULD use short-lived keys and/or hardware-protected keys (e.g., via the PoR mechanism). The use of PoR in Actor Chain Entries provides additional assurance that the signing key is bound to a verified execution environment.

7.6. Privacy of Prior Actors

The actor_chain expose the identities of all actors in the delegation path to every Relying Party that receives the token. While this is necessary for certain audit and policy requirements, it may conflict with privacy goals. For example, in a chain a -> b -> c -> d -> e, the originating actor a may require its identity to be hidden from e while still ensuring the integrity of the delegation.

Deployments SHOULD consider the following anonymization and privacy-preserving techniques:

7.6.1. Pseudonymous Identifiers

Instead of using globally unique or stable identifiers (like email addresses or client IDs), the Authorization Server (AS) can issue pairwise pseudonyms for actors. In the chain a -> b -> c -> d -> e: - The AS replaces sub: "a" with a pseudonym sub: "pseudo-xyz" that is only meaningful to the AS. - Relying Party e sees that the chain started with a verified actor, but does not know it was a. - The AS maintains a mapping to allow for forensic reconstruction if authorized.

7.6.2. Selective Disclosure (SD-JWT)

The actor_chain MAY be implemented using Selective Disclosure for JWTs [[!I-D.ietf-oauth-selective-disclosure-jwt]]. This allows individual Actor Chain Entries to be hashed with unique salts, enabling verification of the entry's integrity without revealing its cleartext content unless a corresponding disclosure is provided.

7.6.3. Identity Bridging for Anonymity

An Identity Bridge [[!I-D.ietf-spice-arch]] MAY act as an "Anonymizer" by performing a token exchange that replaces sensitive predecessor entries in the actor_chain with generic or pseudonymous identifiers, while still vouching for the chain's security properties.

7.6.4. Targeted Selective Disclosure

A more advanced privacy model involves disclosing actor identities only to trusted intermediate parties. In the chain a -> b -> c -> d -> e, actor a may be visible to b, c, and d, but hidden from e.

This is achieved through *Targeted Disclosure* using SD-JWT: - The AS generates the actor_chain with actor a's identity hashed and salted. - Alongside the JWT, the AS provides a "Disclosure" string (the salt and cleartext value) for actor a. - When the AS issues a token for b, c, or d, it includes this disclosure string. This allows these "internal" actors to "unlock" the hash and see a's true identity. - When the final token is exchanged for recipient e, the AS (or the last internal actor d) simply omits the disclosure string for actor a. - Actor e receives the same JWT but without the "key" to unlock a. e can verify the cryptographic integrity of the entire chain (proving that a verified, though anonymous, actor started it) without ever seeing a's identity.

Recipient	JWT Payload	Disclosure provided?	Interpretation
Actor b	_sd: [hash_a]	*Yes*	b sees sub: a
Actor e	_sd: [hash_a]	*No*	e sees sub: <hidden>

Table 2

7.6.5. Encryption (JWE)

The entire actor_chain claim can be encrypted using JWE [[RFC7516]] so that only the final intended audience e can decrypt it, preventing intermediate actors like b, c, and d from seeing the IDs of their predecessors. Alternatively, the chain can be nestedly encrypted for different parties in the path.

7.7. Confused Deputy Mitigation

In both modes, a confused deputy attack— where a legitimate actor is tricked into delegating to a malicious downstream— is detectable because the malicious downstream actor’s identity appears in the chain, providing forensic evidence of the attack path. In Self-Attested Mode, the malicious actor’s own cryptographic signature provides non-repudiable evidence of its participation.

8. IANA Considerations

8.1. JSON Web Token Claims Registration

This document requests registration of the following claim in the "JSON Web Token Claims" registry established by [[RFC7519]]:

* *Claim Name*: actor_chain
* *Claim Description*: A Cryptographically Verifiable Actor Chain — an ordered array of actor entries representing the complete delegation chain with optional per-entry cryptographic signatures.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]

8.2. CBOR Web Token Claims Registration

This document requests registration of the following claim in the "CBOR Web Token (CWT) Claims" registry established by [[RFC8392]]:

* *Claim Name*: actor_chain
* *Claim Description*: A Cryptographically Verifiable Actor Chain.
* *CBOR Key*: TBD (e.g., 40)
* *Claim Type*: array
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]

Authors' Addresses

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

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

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

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