Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 29 October 2026

J. Miller

27 April 2026

Zero-Trust Intent Protocol (ZTIP)
draft-miller-ztip-00

Abstract

The Zero-Trust Intent Protocol (ZTIP) defines three primitives for verifiable delegation, intent binding, and behavioral attestation in multi-agent systems:

1. *Delegation Chain Attestation* — a nested-JWS structure carrying signed delegation from a root principal (a human user, an orchestrator, or a trusted root) through intermediary agents to a downstream agent or tool, with cryptographic enforcement of scope monotonicity.

2. *Intent-Scoped Authorization* — extends any signed authorization token (OAuth 2.0 access tokens, GNAP grants, ZTNP Permits, or vendor-defined tokens) with a hash of the originator's signed intent, so that an agent operating under that token cannot use it for actions inconsistent with the original authorization. This addresses the prompt-injection-induced confused-deputy attack.

3. *Behavioral Claim Extensions* — a claim shape conveying behavioral safety properties (prompt-injection-tested, tool-call-audit-logged, output-validated, human-in-loop policy, etc.) that can be carried in any signed credential about an agent.

ZTIP is composable. It can be deployed standalone, composed with the Zero-Trust Negotiation Protocol (ZTNP) Core [I-D.miller-ztnp] for posture-aware deployments, composed with OAuth 2.0 / JWT for token-bound deployments, composed with GNAP [RFC9635], or composed with bespoke authorization systems. Section 6 specifies concrete composition profiles.

This draft is an individual submission. The appropriate IETF venue for progressing this work is an open question; the author welcomes community guidance.

Status of This Memo

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

This Internet-Draft will expire on 29 October 2026.

Copyright Notice

Table of Contents

1. Introduction

In multi-principal delegation chains, an action is rarely the work of a single principal. A human user instructs an orchestrator; the orchestrator delegates a subtask to a sub-agent; the sub-agent invokes a tool. This chain of delegation introduces three classes of security problem that existing protocols do not address:

1. *Delegation integrity.* When a downstream tool receives a call, can it verify that the call chain leading to it was authorized end-to-end? Per-hop authorization tokens (OAuth 2.0 access tokens, JWTs, ZTNP Permits) only show that the immediately upstream party was trusted by the receiver — not that the entire chain back to the originating principal was authorized.

2. *Intent binding.* An authorization token issued for "summarize email" can be used for any action within the token's stated scope. If a sub-agent is prompt-injected mid-task into "send all email to attacker.example", the token's signature is still valid; only the intent has been substituted. This is a confused-deputy attack mediated by language-model context manipulation.

3. *Behavioral transparency.* A receiver applying policy to a language-model-driven agent needs more than identity and scope. Has the agent been tested against prompt injection? Are its tool calls audited? What is its human-in-loop policy? These are behavioral posture properties that don't fit the generic claim schemas of identity and authorization protocols.

ZTIP addresses these three problems with three primitives, specified in Sections 3, 4, and 5. Each primitive is independently useful; deployments MAY adopt one without the others.

ZTIP does not require any specific authorization or identity protocol. It defines its primitives in the abstract and provides composition profiles (Section 6) for the most common authorization systems: OAuth 2.0 / JWT, GNAP, ZTNP, and vendor-defined tokens. This pattern follows DPoP [RFC9449], which is technically standalone but most often composed with OAuth 2.0.

1.1. Goals

ZTIP is designed to:

1. Provide cryptographic evidence of delegation authority from a root principal to a downstream agent, with verifiable scope monotonicity at each layer.

2. Bind authorization to specific user intent, preventing prompt-injection-induced authorization expansion.

3. Convey behavioral posture claims that inform policy decisions about which capabilities an agent may access.

4. Compose with multiple authorization systems without requiring any specific one.

1.2. Non-Goals

ZTIP does not define:

* Identity or authentication.

* The runtime envelope for inter-agent communication.

* The tool capability protocol.

* Posture assessment or framework semantics (see ZTNP [I-D.miller-ztnp]).

* A new authorization-token format. ZTIP extends existing authorization tokens; it does not define a new one.

* A specific Key Source mechanism. ZTIP is agnostic between JWKS endpoints, X.509 chains, ZTNP's IKS, and DID resolution.

1.3. Relationship to Existing Token Chaining Mechanisms

Reviewers familiar with nested JWTs, OAuth 2.0 Token Exchange [RFC8693], or GNAP continuation [RFC9635] may reasonably ask why ZTIP defines new primitives rather than reusing those mechanisms. The structures look superficially similar: a chain of signed credentials, each issued in response to a prior one. ZTIP's primitives differ on three properties that none of those mechanisms provide.

1. *Monotonic scope enforcement at every receiver.* Token Exchange permits a client to request a downstream token whose scope is determined by the AS's policy; the issued token's scope is not constrained by protocol to be a subset of the inbound token's scope. There is no protocol-level guarantee that "this downstream token grants only what the upstream token already granted." ZTIP's Section 3.4 makes scope reduction a normative property of the chain itself, verifiable at any receiver without recourse to the AS.

2. *End-to-end chain auditability.* Nested JWTs and chained Token Exchange responses do not preserve the identity and signature of every intermediary. A receiver of a token-exchanged JWT typically sees only the AS's signature; the chain of clients that requested and re-requested the token is opaque to the resource server. ZTIP's nested-JWS structure preserves each delegator's signature, so the receiver can verify "who authorized whom" for every step of the chain.

3. *Originator intent binding.* No existing token-chaining mechanism binds an issued token to a structured originator intent. A token-exchanged token is bound to its scope; ZTIP additionally binds it to the specific intent (intent_hash) the originator signed. This is what makes Section 4 a defense against prompt-injection-induced confused-deputy attacks: an injected instruction can request operations within the token's scope, but those operations will not match the bound intent.

In short: existing token-chaining mechanisms answer "may this client present this token?" ZTIP additionally answers "did the original principal authorize this entire chain to perform this specific operation?" Deployments needing only the former should continue to use existing mechanisms; ZTIP is for deployments that need both.

1.4. Document Structure

Sections 2 and 3 define terminology and the Delegation Chain Attestation primitive. Section 4 defines Intent-Scoped Authorization. Section 5 defines Behavioral Claim Extensions. Section 6 defines composition profiles for OAuth 2.0, ZTNP, GNAP, and custom authorization systems. Sections 7 and 8 contain Security and Privacy Considerations. Section 9 contains IANA Considerations. Appendix A is a worked example using the ZTNP composition profile. Appendix B defines the conformance profile.

1.5. Requirements Language

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.

2. Terminology

Term	Definition
Originator	The root principal of a Delegation Chain — a user, an orchestrator, or a trusted root system that initiated an action.
Signed Intent	A JWS produced by an Originator describing the originally authorized action and its scope.
Delegation Chain	A nested-JWS structure rooted at a Signed Intent, with each delegating principal adding a signature layer that names the next party.
Scope Reduction	The principle that each layer of a Delegation Chain may grant a subset, not a superset, of its parent’s scope (Section 3.4).
Authorization Token	A signed authorization credential carrying scope and expiration information about a delegated capability. Examples: OAuth 2.0 access tokens, GNAP grants, ZTNP Permits.
Intent-Scoped Authorization Token	An Authorization Token additionally extended with the four ZTIP fields defined in Section 4: intent_hash, intent_scope, chain_root_iss, chain_root_jti.
Key Source	A mechanism by which a verifier obtains the public key of a signing principal — a JWKS endpoint, an X.509 chain, an Issuer Key Set (when composed with ZTNP), or a DID resolution. ZTIP is agnostic to the specific Key Source mechanism.
Behavioral Claims	A claim shape (Section 5) carrying behavioral safety properties of an agent. May appear in any signed credential about an agent.

Table 1

3. Delegation Chain Attestation

A Delegation Chain is a nested-JWS structure in which each delegating principal adds a signature layer. The root of the chain is a Signed Intent produced by the Originator. Each downstream layer wraps the previous chain and is signed by the principal delegating to the next.

ZTIP does not specify how each principal’s public key is discovered. Implementations resolve keys via a Key Source appropriate to their deployment. Section 6 defines specific composition profiles and their Key Source mechanisms.

3.1. Chain Structure

Three-hop example (user → orchestrator → sub-agent → tool):

```
Outermost (sub-agent A's JWS):
  payload: {
    "del_chain_ver": "0.1",
    "delegator": "agent:A",
    "delegatee": "tool:T",
    "scope_reduction": { "actions": ["read"], "data": ["internal"] },
    "iat": ..., "exp": ...,
    "inner": "<orchestrator O's compact JWS>"
  }

Middle (orchestrator O's JWS):
  payload: {
    "del_chain_ver": "0.1",
    "delegator": "principal:O",
    "delegatee": "agent:A",
    "scope_reduction": { ... },
    "iat": ..., "exp": ...,
    "inner": "<root Signed Intent compact JWS>"
  }

Root (Signed Intent):
  payload: {
    "del_chain_ver": "0.1",
    "intent_root": true,
    "originator": "user:alice",
    "intent_object": { ... structured intent (Section 3.2) ... },
    "intent_hash": "<base64url(SHA-256(JCS(intent_object) UTF-8 bytes))>",
    "authorized_chain": ["principal:O", "agent:A"],
    "scope": { ... },
    "iat": ..., "exp": ...
  }
```

3.2. Intent Object Schema and Canonicalization

The root layer's intent — what the originating principal is authorizing — MUST be expressed as a structured JSON object, not as free-form natural-language text. Three reasons:

1. Free-form text is not reproducibly hashable across implementers (whitespace, line endings, and encoding choices all introduce non-determinism).

2. Free-form text is not machine-comparable to a downstream operation; verifiers need structured fields to evaluate intent_scope against a requested operation (Section 4).

3. A structured intent forces the originator’s user interface to disambiguate intent at the point of signing, rather than deferring interpretation to downstream agents — which is precisely the gap that creates confused-deputy attacks.

3.2.1. Intent Object Schema

The intent_object field on the root Signed Intent MUST be a JSON object containing at least:

Field	Type	Required	Description
action	string	Yes	A short verb describing the user’s desired operation (e.g., "summarize", "search", "compose").
scope	object	Yes	The scope within which the originator authorizes downstream agents to operate. MUST equal the root layer’s scope field.
target	string	No	A human-readable description of what the action operates on.
constraints	object	No	Profile-defined additional constraints (e.g., must_not, time-window limits).

Table 2

Profile documents MAY extend the schema with additional fields. Extension fields MUST NOT alter the meaning of the required fields above.

3.2.2. Canonicalization

The canonical form of an intent_object is its JSON Canonicalization Scheme serialization, per [RFC8785]. JCS provides:

* Deterministic key ordering (lexicographic).

* Deterministic number representation (no trailing zeros, no scientific notation for non-special values).

* UTF-8 encoding of the resulting byte sequence.

* No insignificant whitespace.

Implementers MUST use a JCS implementation conformant with RFC 8785. Variant canonicalization schemes MUST NOT be used.

3.2.3. Intent Hash

intent_hash is computed as:

```
intent_hash = base64url(SHA-256(JCS(intent_object) encoded as UTF-8 bytes))
```

base64url is the URL-safe base64 encoding without padding, per Section 5 of [RFC7515].

Verifiers receiving a chain MUST recompute intent_hash from the root layer's intent_object and verify it matches both:

1. The intent_hash field within the same root layer (consistency check).

2. The intent_hash field on any Intent-Scoped Authorization Token (Section 4) issued under this chain.

Mismatch causes rejection with INTENT_SCOPE_MISMATCH.

3.2.4. Inline Examples

Three intent objects illustrating common patterns. Each is shown as the input JSON, the JCS canonical bytes (where shown), and the resulting intent_hash. The hash values shown are computed against an RFC 8785 reference implementation; conforming implementations MUST produce these values for these inputs.

Example 1: Read-only summarization.

```
{
  "action": "summarize",
  "scope": {
    "actions": ["read"],
    "data": ["internal", "pii"],
    "tools": ["email.list", "email.read"]
  },
  "target": "unread emails from the last 24 hours",
  "constraints": {
    "must_not": ["email.send", "email.delete"]
  }
}
```

JCS canonical bytes (the actual canonical form, with no insignificant whitespace):

```
{"action":"summarize","constraints":{"must_not":["email.send","email.delete"]},"scope":{"
actions":["read"],"data":["internal","pii"],"tools":["email.list","email.read"]},"target"
:"unread emails from the last 24 hours"}
```

JCS sorts object keys lexicographically (note action before constraints before scope before target at the top level, and must_not after actions/data/tools are reordered within the inner objects) but *preserves the insertion order of array elements* — must_not retains email.send before email.delete from the input. RFC 8785 specifies key sorting for objects, not for array elements.

Computed intent_hash: Q9h_MJaQrDtKRb7MKfwg664jUWmVlErfdS8Qm1y6qNc.

Example 2: Knowledge-base search with redaction.

```
{
  "action": "search",
  "scope": {
    "actions": ["read"],
    "data": ["internal"],
    "tools": ["kb.query"]
  },
  "target": "internal product specifications matching 'thermostat'",
  "constraints": {
    "redact": ["customer_pii", "pricing_internal"]
  }
}
```

Computed intent_hash: vMdbs17cp0K0-TJKz8l5iTPMSgXLVN4Epyjq5yz7gYY.

Example 3: Financial transfer with strict allowlist.

```
{
  "action": "transfer_funds",
  "scope": {
    "actions": ["write"],
    "tools": ["bank.transfer"],
    "rate_limit": { "max": 1, "window_seconds": 86400 }
  },
  "target": "vendor invoice payment",
  "constraints": {
    "amount_max_usd": 500,
    "destination_must_be_in": ["preapproved_vendors"]
  }
}
```

Computed intent_hash: OW_76HLPAd8nVL7Z3e_jk1Q_8aQmFzn71hqrTMSfpeQ.

The constraints field is profile-specific. Verifiers MUST ignore unrecognized constraint fields they do not understand, EXCEPT when local policy requires recognition of those fields — in which case the policy MUST reject Posture Assertions or Authorization Tokens whose intent contains uninterpretable constraints.

3.3. Verification Rules

A chain recipient MUST:

1. Unwrap each JWS layer and verify against the stated delegator's public key, obtained from the deployment's Key Source.

2. Verify the chain of delegator/delegatee references is unbroken. Each layer's delegatee MUST equal the next layer's delegator (or, for the layer immediately above the root, MUST appear in the root's authorized_chain).

3. *Enforce scope monotonicity* per Section 3.4. Attempts to expand scope MUST cause rejection with reason code DEL_CHAIN_SCOPE_EXPANDED.

4. Verify the root is signed by an Originator the recipient trusts.

5. Verify no layer is expired (exp in past, accounting for clock skew under 5 minutes).

6. *Verify chain depth* is within the configured maximum per Section 3.5.

7. Recompute intent_hash from the root's intent_object per Section 3.2 and verify the recomputed value matches the root's claimed intent_hash.

Implementations supporting Delegation Chains MUST enforce scope monotonicity (rule 3), chain-depth limit (rule 6), and intent-hash consistency (rule 7). These are the load-bearing security properties of the chain.

3.4. Formal Definition of Scope Monotonicity

Rule 3 in Section 3.3 requires "scope_reduction MUST be a subset of the parent's scope." This subsection formalizes "subset" for each scope field type so that conforming implementations agree on what monotonicity means.

A scope value at child layer C is *monotone* with respect to parent layer P when, for every field present in C, the relation in the table below holds:

Field	Type	"C ⊆ P" relation
actions	array of strings	Every element of C.actions is an element of P.actions. C.actions MAY be empty.
data	array of strings	Every element of C.data is an element of P.data. C.data MAY be empty.
tools	array of strings	Every element of C.tools is an element of P.tools. C.tools MAY be empty.
rate_limit.max	integer	C.max <= P.max.
rate_limit (combined)	object	C.max / C.window_seconds <= P.max / P.window_seconds (effective rate is no greater) AND C.max <= P.max (burst capacity is no greater).
ttl	integer (seconds)	C.ttl <= P.ttl.
iat (per-layer)	integer (Unix seconds)	C.iat >= P.iat.
exp (per-layer)	integer (Unix seconds)	C.exp <= P.exp.

Table 3

Field introduction. A child layer MUST NOT introduce a scope field that is not present in its parent. A field absent in P but present in C is a scope expansion and MUST cause rejection with DEL_CHAIN_SCOPE_EXPANDED.

Profile-defined fields. When a profile defines additional scope fields, the profile document MUST specify the subset relation for those fields. Verifiers encountering profile-defined fields without a known subset relation MUST treat them as opaque: presence in C requires identical value in P (i.e., C.field deeply equals P.field) for monotonicity to hold.

Empty vs. omitted in the child. An empty array ([]) in a child layer means "no operations of this type are authorized." An omitted field in a child layer is interpreted as inheriting the parent's value, NOT as removing the parent's constraint. A child layer that wishes to drop a dimension entirely MUST set it to an empty array.

Diagnostic on rejection. When a chain is rejected for scope violation, the verifier SHOULD include in the DEL_CHAIN_SCOPE_EXPANDED reason the specific field path and the offending values. Example: "field": "tools", "child_value": "email.send", "parent_authorizes": ["email.list", "email.read"].

3.5. Maximum Chain Depth

Implementations MUST enforce a maximum chain depth limit. The depth is the number of nested JWS layers, counting both the root Signed Intent and each delegation wrapper.

* *RECOMMENDED maximum: 8 layers.* This is sufficient for the orchestration patterns observed at the time of this specification's publication; deployments needing more SHOULD document why.

* Deployments operating in environments with bounded delegation patterns MAY configure a lower limit (e.g., 3 for simple two-hop chains).

* Verifiers MUST reject chains exceeding the configured limit with reason code DEL_CHAIN_DEPTH_EXCEEDED.

* The depth check MUST be performed before recursive signature verification, to prevent stack-exhaustion or computational denial-of-service attacks via deeply nested chains.

The depth limit also bounds the legitimate value of the max_tool_call_depth Behavioral Claim (Section 5). An agent claiming max_tool_call_depth: N SHOULD NOT participate in chains where the verifier's configured depth limit is less than N.

Cost-scaling note. Signature verification, JSON parsing, and JCS canonicalization each scale linearly with chain depth: an N-layer chain requires N signature verifications and N JSON-payload parses at the receiver. The depth limit therefore bounds two distinct concerns — stack-exhaustion (addressed by the pre-verification depth check above) and per-request CPU cost (which scales linearly up to the limit). Verifiers operating at high request rates SHOULD configure a depth limit at the lower end of the range their orchestration patterns permit, and SHOULD treat the configured limit as a CPU-

budget parameter as well as a stack-safety parameter. Alternative chain encodings (flat hash-linked structures, batch-verifiable signatures) were considered but rejected for this base specification in favor of the broad implementer familiarity with nested JWS; profiles MAY define alternative encodings where the cost matters.

3.6. Carrying the Chain in Authorization Flows

The Delegation Chain is carried alongside an Authorization Token. The exact carriage mechanism depends on the composition profile (Section 6). When a recipient’s policy requires a chain (e.g., a policy field require_delegation_chain: true), absence MUST cause DEL_CHAIN_MISSING denial.

Carriage MAY be either session-bound (the chain is supplied once at session establishment and reused for subsequent operations within the session) or per-call (each operation carries its own chain). Per-call carriage is appropriate for stateless gateways and edge-deployed verifiers that cannot maintain per-session state. Session-bound carriage is appropriate for long-lived sessions where re-supplying the chain on every request is wasteful. Implementations supporting both modes MUST advertise their preference in the relevant composition profile’s discovery mechanism.

3.7. Chain-Specific Reason Codes

The following reason codes are registered in the IANA ZTIP Denial Reason Code Registry (Section 9):

Code	Meaning
DEL_CHAIN_MISSING	Policy requires a Delegation Chain but none was provided.
DEL_CHAIN_BROKEN	Chain of delegator/delegatee references has a gap.
DEL_CHAIN_SCOPE_EXPANDED	A child layer attempts to grant broader scope than its parent.
DEL_CHAIN_EXPIRED	One or more chain layers has expired.
DEL_CHAIN_UNTRUSTED_ROOT	The root Signed Intent is not signed by a trusted Originator.
DEL_CHAIN_DEPTH_EXCEEDED	The chain exceeds the verifier’s configured maximum depth.

Table 4

4. Intent-Scoped Authorization

4.1. Motivation

An authorization token issued without reference to the original authorized intent can be used for any action within its stated scope, regardless of what the user actually asked. This is the mechanism by which prompt injection escalates to a confused-deputy attack: the injected instruction substitutes a new operational intent, but the token’s scope is broad enough to authorize it.

Intent-Scoped Authorization Tokens bind the token to the root Signed Intent (Section 3). A resource receiving such a token can verify that the action being requested is consistent with the original intent, not just with the token’s scope.

4.2. Field Extensions to Authorization Tokens

ZTIP defines four additional claims for any signed Authorization Token:

Field	Type	Required	Description
intent_hash	string	Yes	base64url(SHA-256(JCS(intent_object)) UTF-8 bytes)) per Section 3.2 — MUST equal the root Signed Intent's intent_hash.
intent_scope	object	Yes	The authorized scope derived from the root Signed Intent.
chain_root_iss	string	Yes	Identifier of the root Signed Intent's Originator.
chain_root_jti	string	Yes	jti of the root Signed Intent (for revocation/audit).

Table 5

These fields are added to the Authorization Token's existing claim set. The exact placement depends on the composition profile (Section 6):

* *OAuth 2.0 / JWT access token*: top-level claims alongside iss, sub, scope, exp.

* *ZTNP Permit*: top-level fields alongside permit_id, constraints.

* *GNAP grant*: as an extension element of the grant response.

The token issuer (whoever issues the underlying authorization token) MUST verify that the action requested by the bearer is within intent_scope before the token is honored. If the bearer attempts to invoke a tool or access data outside intent_scope, the token issuer MUST deny the request even if the token's other scope fields would otherwise permit it.

4.3. Scope Consistency Check

A token verifier enforcing an Intent-Scoped Authorization Token MUST evaluate the following before each gated operation:

1. Determine the operation's (action, data_classes, tool) signature from the request being gated.

2. Verify that the operation falls within intent_scope, using the same subset relations as Section 3.4. Specifically: the operation's tool MUST be in intent_scope.tools (when intent_scope.tools is present); the operation's action MUST be in intent_scope.actions; each data class touched by the operation MUST be in intent_scope.data.

3. If any check fails, return a denial with reason code INTENT_SCOPE_MISMATCH.

The verifier MUST additionally verify that the token's intent_hash matches the recomputed hash from the root Signed Intent's intent_object (per Section 3.2). A mismatch indicates either a substituted token or a tampered chain and MUST cause INTENT_SCOPE_MISMATCH rejection.

4.4. Operation Classification

The Scope Consistency Check above requires the verifier to determine an operation's (action, data_classes, tool) signature. ZTIP does not standardize how operations are classified into actions and data classes: a service mesh fronting a relational database has different action vocabularies (SELECT, INSERT) than an email gateway (email.read, email.send) or a financial system (transfer, query_balance). Operation classification is therefore deployment- and profile-defined, not protocol-defined.

This is a deliberate deferral, but it is also a security-relevant gap. Misclassification — labeling a write-causing tool call as read, mapping a data-exfiltrating tool to a benign-looking action, or having two parties disagree on what email.send means — silently bypasses intent-scope enforcement. The cryptographic guarantees of Sections 3 and 4 are then evaluating an operation signature that does not correspond to the operation actually performed.

Two requirements follow:

1. *Profiles SHOULD specify a strict operation taxonomy* for their domain — the set of valid tool names, the action each tool implements, and the data classes each operation touches — so that classification is not left to ad-hoc deployment choice. The taxonomy SHOULD be authenticated (e.g., served from a tool-capability registry that the verifier trusts) so that a tool cannot self-classify into a more permissive category at request time.

2. *Verifiers MUST treat operation classification as part of their security boundary.* A tool whose runtime behavior does not match its declared classification is a security failure regardless of cryptographic verification. Deployments SHOULD audit tool implementations against their declared classifications, and SHOULD reject operations whose classification cannot be authenticated against a trusted source.

Cryptographic verification of the chain and the intent hash is necessary but not sufficient: the intent-binding guarantee depends on the verifier and the tool agreeing on what operation is being performed. Treat the operation taxonomy with the same rigor as the cryptography.

5. Behavioral Claim Extensions

For deployments where a verifier needs behavioral posture information about an agent, ZTIP defines a claim shape carrying behavioral safety properties. This claim shape can appear in any signed credential about an agent — typically a Posture Assertion when composed with ZTNP, a JWT claim about the bearer when composed with OAuth, or a vendor-defined credential.

Behavioral claims are advisory and non-normative with respect to ZTIP's core security guarantees. The chain-integrity, scope-monotonicity, and intent-binding properties of Sections 3 and 4 do not depend on any behavioral claim. A verifier MAY ignore behavioral claims entirely without weakening any of those properties; a verifier that consults behavioral claims is making a _policy_ decision about acceptable agent behavior, not relying on a security primitive. Behavioral claims are a transport for the sort of attestation that one party makes to another about how an agent is operated; their meaning is established by issuer reputation, the evidence sub-field below, and out-of-band agreement, not by the protocol. Deployments dismissing behavioral claims as "marketing, not protocol" can deploy ZTIP without them and still get the chain-integrity and intent-binding guarantees.

The claims appear under the namespace ai_behavior within the credential's claim payload.

5.1. Standard AI Behavior Claims

The following claim names are defined by this specification:

Claim	Type	Description
prompt_injection_tested	boolean	Tested against prompt-injection scenarios.
tool_call_audit_logged	boolean	All tool invocations are logged to a tamper-evident audit trail.
tool_misuse_score	number	0.0-1.0; lower = lower observed misuse risk. Calibration is profile-specific; ZTIP does not standardize a measurement methodology.
output_validated	boolean	Agent output is validated before being acted upon.
human_in_loop_policy	string	One of: never, on_high_risk, on_data_egress, always. Profiles MAY define additional values.
max_tool_call_depth	integer	Maximum delegation-chain depth the agent will recurse into. SHOULD be the verifier's configured chain-depth limit (Section 3.5).
data_exfil_controls	boolean	Agent has data-exfiltration controls.

Table 6

A verifier's policy MAY reference ai_behavior claims as prerequisites for granting specific actions. For example: a policy may require prompt_injection_tested: true before granting actions: ["email_send"].

5.1.1. Evidence Sub-Field

Each behavioral claim MAY be accompanied by a sibling evidence object that records how the claim was substantiated. Verifiers concerned about the rigor of a behavioral claim SHOULD inspect its evidence object before relying on the claim.

The evidence object SHOULD contain:

Field	Type	Description
source	string	Identifier of the party that generated the evidence (e.g., URI of an evaluator, internal red-team identifier, vendor name).
method	string	Methodology used to substantiate the claim (e.g., red_team_test_suite_v3, automated_prompt_injection_corpus, manual_review_2026Q1).
date	string	ISO 8601 date when the evidence was generated.
validity_days	integer	OPTIONAL: how long the evidence is considered valid; verifiers MAY treat older evidence as expired.
evidence_hash	string	OPTIONAL: SHA-256 hash of an evidence artifact (test report, log file) the verifier can request out-of-band.

Table 7

Example:

```
{
  "ai_behavior": {
    "prompt_injection_tested": true,
    "evidence": {
      "prompt_injection_tested": {
        "source": "https://example-evaluators.org",
        "method": "owasp_llm_top10_2025_corpus_v2",
        "date": "2026-04-15",
        "validity_days": 90,
        "evidence_hash": "Q7nB...base64url..."
      }
    }
  }
}
```

evidence entries are nested by claim name within an evidence object alongside the claims they substantiate (so a verifier can iterate claim → evidence).

The Issuer signing the credential is responsible for the truthfulness of evidence entries. Profiles MAY define additional required evidence fields for specific claims (e.g., requiring evidence.method for tool_misuse_score).

5.1.2. Baseline Corpora for "Tested" Claims

A boolean claim such as prompt_injection_tested: true is meaningful only with respect to a defined test corpus. Without a referenced corpus the claim collapses to vendor self-attestation and offers a verifier no basis for comparing two issuers' assertions. ZTIP does not standardize a single corpus — the field is moving rapidly and no single test set captures the full attack surface — but does specify how a corpus is referenced.

When evidence.method substantiates a _tested claim, the value SHOULD identify a publicly documented test corpus and version. Examples of such corpora include:

* The OWASP Top 10 for Large Language Model Applications (commonly cited as "OWASP LLM Top 10"), with a year and version identifier (e.g., owasp_llm_top10_2025_v1).

* Vendor- or community-maintained prompt-injection test suites with public methodology (e.g., adversarial-prompt corpora published by recognized red teams).

* Internal corpora, where the methodology is documented and the corpus version is identified, accompanied by an evidence_hash over the corpus or its specification.

Verifiers SHOULD treat claims whose evidence.method does not identify a public corpus, an evidence_hash over an internal corpus, or a recognized methodology as no stronger than vendor self-attestation. Verifiers depending on a _tested claim for a policy decision SHOULD additionally require that the corpus referenced in evidence.method is one the verifier accepts; deployments differ in which corpora they consider authoritative, and ZTIP intentionally leaves that choice to the verifier rather than mandating one corpus globally.

This is consistent with the framing in Section 5: behavioral claims are advisory. Pointing at a corpus does not make the claim a security primitive — it makes it a policy input that a verifier can compare across issuers without deferring entirely to issuer reputation.

5.2. Vendor-Defined and Bespoke Claims

Implementations MAY define additional behavioral claims for their domain. ZTIP does not maintain a central registry of behavioral claim names; instead, three naming conventions allow extension without coordination:

* *URI-namespaced names* (RECOMMENDED for cross-organization deployments): the claim name is a URI per [RFC3986] in a domain controlled by the publisher. Example: https://acme.example/ztip/claims/red_team_score_v1. Anyone reading the URI can identify the publisher and look up the claim's definition.

* *Vendor-prefixed names* (RECOMMENDED for in-domain consortia): a registered organizational identifier as a prefix. Example: acme:red_team_score_v1.

* *Privately-agreed names* (acceptable when both parties understand the meaning out-of-band): unprefixed identifiers used only within deployments where issuer and verifier have agreed semantics. Example: internal_score.

This pattern follows JWT private-claim conventions ([RFC7519] Section 4.3) and OAuth scope-string conventions. Verifiers MUST ignore unrecognized claims they do not understand. Verifiers that depend on a behavioral claim for a policy decision MUST fail closed if the claim is absent.

5.3. Interaction with Standard Claims

Vendor-defined claims MUST NOT redefine the meaning of any of the standard claims listed above. A vendor wishing to refine the semantics of, say, tool_misuse_score MUST publish a new vendor-namespaced claim (e.g., acme:tool_misuse_score_v2) rather than redefining the standard one.

6. Composition Profiles

ZTIP is designed to compose with multiple authorization systems. This section specifies normative composition profiles. Deployments SHOULD use a published profile when one exists; vendors MAY define additional profiles for their authorization systems.

6.1. Composition with ZTNP

When composed with the Zero-Trust Negotiation Protocol [I-D.miller-ztnp]:

* The Authorization Token is the ZTNP Permit (Section 8 of ZTNP).

* The Key Source is the Issuer Key Set (IKS, Section 6 of ZTNP) for each delegating principal.

* The Delegation Chain (defined in Section 3 of this document) is carried in the PROOF message of ZTNP’s Negotiation phase (Section 7.3 of ZTNP), under the delegation_chain field.

* ZTIP fields (intent_hash, intent_scope, chain_root_iss, chain_root_jti) are top-level fields on the Permit.

* Behavioral Claims appear under the ai_behavior namespace in any credential about the Prover. Their canonical placement is on the Posture Assertion at claims.posture.ai_behavior (a profile-defined extension under ZTNP’s claims.posture per Section 5.4 of ZTNP). Profiles MAY additionally place per-issuance Behavioral Claims on the Permit at ai_behavior (e.g., the MCP profile’s legacy_stdio_binding flag), where they describe a property of how the Permit was issued rather than the long-lived posture of the Prover.

A worked example using this profile appears in Appendix A.

6.2. Composition with OAuth 2.0 / JWT

When composed with OAuth 2.0 [RFC6749] access tokens in JWT format ([RFC7519]):

* The Authorization Token is the JWT access token.

* The Key Source for chain-layer signature verification is each principal's published JWKS endpoint, located via OAuth Authorization Server Metadata or a similar discovery mechanism.

* ZTIP fields are top-level claims in the access token JWT, alongside iss, sub, aud, exp, scope.

* The Delegation Chain is carried in a separate HTTP header alongside the bearer token:

```
Authorization: Bearer <jwt_access_token>
ZTIP-Chain: <delegation_chain_compact_jws>
```

* Behavioral Claims appear in the access token under a top-level ai_behavior claim (an object), alongside the Intent-Scoped fields.

The JWT access token, with ZTIP fields, looks like:

```
{
  "iss": "https://auth.example",
  "sub": "agent:summarizer-3",
  "aud": "https://api.example",
  "exp": 1745504400,
  "iat": 1745500900,
  "scope": "email.read",
  "intent_hash": "Q9h_MJaQrDtKRb7MKfwg664jUWmVlErfdS8Qm1y6qNc",
  "intent_scope": { "actions": ["read"], "tools": ["email.read"] },
  "chain_root_iss": "user:alice",
  "chain_root_jti": "intent_01HVXYZ_SUMMARIZE_REQUEST",
  "ai_behavior": {
    "prompt_injection_tested": true,
    "human_in_loop_policy": "on_high_risk"
  }
}
```

This composition layers ZTIP cleanly above OAuth without modifying OAuth's core flows. The OAuth Authorization Server is responsible for issuing tokens that include the ZTIP fields; the resource server is responsible for performing the scope-consistency check (Section 4.2) on each request.

6.3. Composition with GNAP

When composed with the Grant Negotiation and Authorization Protocol [RFC9635], ZTIP uses GNAP's extension-parameter registries (Section 10 of [RFC9635]) rather than overloading existing fields.

Delegation Chain in the grant request. The Delegation Chain is carried as a top-level extension parameter ztip_delegation_chain on the grant request body, in compact-JWS form. This parameter is intended for registration in the "GNAP Grant Request Parameters" registry (Section 10.3 of [RFC9635]).

```
POST /grant HTTP/1.1
Content-Type: application/json

{
  "access_token": { "access": [ ... ] },
  "client": { "key": { ... } },
  "ztip_delegation_chain": "<compact-JWS of outermost layer>"
}
```

The AS validates the chain (Section 3.3 of this document) before issuing access tokens. A failed validation MUST cause the grant request to be rejected with the appropriate ZTIP reason code reported in an extension field on the GNAP error response.

ZTIP fields on the issued access token. When the AS issues an access token in response to a chain-bearing grant request, the token MUST carry the ZTIP fields intent_hash, intent_scope, chain_root_iss, and chain_root_jti (Section 4 of this document). Placement depends on the access-token format the AS issues:

* If the AS issues a JWT access token, the ZTIP fields appear as top-level claims (per the OAuth 2.0 / JWT composition in Section 6.2 of this document).

* If the AS issues an opaque token, the ZTIP fields are returned alongside the token in the grant response under an extension parameter ztip (object), intended for registration in the "GNAP Grant Response Parameters" registry (Section 10.12 of [RFC9635]). The opaque token’s introspection response MUST also include these fields when introspection is supported.

Key Source for chain layer signatures. Each delegating principal in the chain has its own signing key. GNAP’s grant-request flow exposes the _immediate_ client’s key via the client.key field; ZTIP requires keys for _every_ principal in the chain. The composition resolves this by requiring that each chain layer’s delegator identifier be a URI dereferenceable to a JWKS, or alternatively that the AS maintain an out-of-band trust store for known delegators. Profiles MAY further constrain this; this base spec leaves the choice to deployment.

Continuation requests. GNAP supports continuation of a grant for token refresh and modification. When a continuation request modifies the authorized scope, the client MAY supply a new ztip_delegation_chain value (e.g., representing a further-narrowed delegation). The AS MUST validate that the new chain's scope is consistent with — and not broader than — the chain that originally established the grant.

What this section does not specify. Wire-level normative details — exact field names accepted into the IANA GNAP registries, exact error-response shape, interaction-mode semantics when a chain is required for an interaction step — are deliberately left to a companion profile document. Implementers wanting an end-to-end conformant GNAP+ZTIP deployment SHOULD await or contribute to that companion document. The sketch above is sufficient for early experimentation and for soliciting feedback from the GNAP working group, but it is not a complete profile.

6.4. Custom Authorization Systems

Implementations MAY compose ZTIP with bespoke authorization systems. The composition document MUST specify:

1. The signed credential serving as the Authorization Token.
2. A Key Source for each principal that may sign a delegation chain layer.
3. A carriage mechanism for the chain alongside the credential.
4. A reason-code mapping if the composition's transport differs from HTTP.
5. The placement of ZTIP fields (intent_hash, etc.) within the Authorization Token.
6. The placement of Behavioral Claims within any associated agent credential.

A composition profile SHOULD be published as a separate document when intended for cross-organization use.

6.5. Deployment Pattern: Transparent Intermediate Signer

This subsection describes a recurring deployment pattern that ZTIP supports without additional protocol machinery. It is informative; the underlying mechanics (Sections 3 and 4) are unchanged.

In some delegation chains, requiring per-action human approval is operationally prohibitive — a chat-driven session where the user types ten messages per minute cannot reasonably re-prompt for explicit approval on each turn. The practical pattern that emerges:

1. *Layer 0 (root):* The user (Originator) signs a session-level intent at session establishment. The intent_object describes the broad scope the user is authorizing for the session ("answer questions and call tools related to my project X using only my internal documents"). This is one human-consented signing event.

2. *Layer 1 (transparent intermediate):* A trusted Planner or session controller — running under the user's identity but operating without per-turn user approval — signs delegation layers programmatically as the session proceeds. Each per-turn delegation reduces scope further (scope_reduction) based on the specific user request that turn. The Planner is _transparent_ because it adds chain layers without re-prompting; it is _trusted_ because it is structurally above the prompt-injection-exposed Worker and constrained by the root's scope.

3. *Layer 2 (worker):* A sub-agent (potentially exposed to prompt injection from tool outputs) operates under the per-turn delegation. Its scope is reduced from the Planner's scope, which is reduced from the root.

The transparent intermediate signer pattern works within ZTIP's existing primitives:

* The chain has arbitrary depth (Section 3.5 caps at 8 by default).

* Every layer enforces scope monotonicity (Section 3.4).

* The root retains the user-consented intent_object; intermediate layers carry only scope_reduction (no new intent).

* Any operation reaching the final tool is constrained by the intersection of all layers' scopes.

Security requirements for the intermediate signer:

* The intermediate's signing key MUST be under operational control of the same trust principal as the user's signing environment (the Planner is part of the user's session infrastructure, not a third-party service).

* The intermediate MUST enforce scope monotonicity at signing time; an intermediate that signs scopes broader than the root authorized is a security failure regardless of downstream verification.

* The intermediate SHOULD log per-turn signing decisions to a tamper-evident audit trail. The log SHOULD include the user's per-turn input that derived the scope_reduction (so a subsequent audit can reconstruct what was authorized for each turn).

* The intermediate SHOULD use a short-lived signing key bound to the session (e.g., generated at session establishment, deleted at session termination), to limit the blast radius if the key is exposed.

When to use this pattern:

* Long-running interactive sessions where user re-approval per action is unworkable.

* Programmatic decomposition of a high-level user intent into low-level tool calls, where the decomposition is performed by trusted infrastructure, not by the prompt-injection-exposed Worker.

* Multi-turn chat UX where each turn produces a new sub-intent within the session-level intent.

When NOT to use this pattern:

* High-stakes single-action authorizations (financial transfers, irreversible operations) where per-action explicit consent is the security boundary.

* Deployments where the intermediate signer is not under the user's trust principal (e.g., a third-party SaaS Planner).

Trust-boundary implications. The transparent intermediate signer pattern shifts the human-consent boundary from per-action to per-session, with a trusted automated signing authority producing the per-turn delegation layers in between. This is a deliberate tradeoff for operational viability, and adopters SHOULD treat it as a meaningful change in their threat model rather than an implementation detail. A compromised intermediate cannot expand scope (Section 3.4 prevents that — every per-turn delegation is still strictly within the root intent's scope), but it can manipulate behavior within the authorized space: which tools to invoke, in what order, with what arguments, drawing on which session inputs. For most session-level scopes this residual surface is acceptable; for scopes broad enough that within-scope behavior can itself cause harm, it is not. Two consequences:

1. The intermediate signer is a high-value target equivalent in sensitivity to the Originator's signing surface for the duration of the session, and SHOULD receive equivalent operational hardening (HSM-backed keys, restricted access, monitored signing surfaces).

2. The session-level intent_object SHOULD be no broader than the user would be willing to authorize unattended for the session's duration. If a session-level intent encompasses high-stakes actions, those actions belong in a separate, narrower intent signed at the moment they are needed (per the "When NOT to use this pattern" guidance above), not within the broad session intent.

Future profile documents MAY formalize specific implementations of this pattern (e.g., by specifying a distinct identifier prefix for transparent intermediates, audit-log requirements, key-lifetime bounds). This base specification deliberately leaves the implementation details to deployments and profile authors.

7. Security Considerations

This section is REQUIRED by [RFC3552]. ZTIP introduces security properties beyond those provided by any single authorization system; this section enumerates the additional adversaries and mitigations specific to ZTIP. Composition-specific security considerations (e.g., the security considerations of OAuth 2.0 or ZTNP) continue to apply to deployments using those compositions.

7.1. Trust Anchor

The security of ZTIP is strictly bounded by the trustworthiness of the Originator and the integrity of the Originator's signing environment. This is the protocol's trust anchor, analogous to the certificate-authority hierarchy in TLS or the issuer in OAuth: every guarantee ZTIP provides downstream is conditional on the Originator having signed something the Originator actually meant to sign.

A compromised or malicious Originator can produce a maximally-broad intent_object and a chain authorizing any downstream principal; verifiers cannot distinguish such an intent from a legitimate one within the chain itself. ZTIP's defenses (scope monotonicity, intent binding, chain auditability) protect against compromise _after_ signing — they do not protect against compromise _at_ signing time.

Mitigations for the trust-anchor surface are deliberately out-of-protocol and SHOULD include:

* HSM-backed or hardware-bound storage of the Originator's signing key.

* Short Signed-Intent lifetimes (exp close to iat) so that a compromised intent has limited reach.

* Multi-factor or additional-channel approval at the signing surface for high-stakes intents (e.g., financial transfers, irreversible operations).

* Human review of intent_object contents at the signing UI before the signature is produced — i.e., the user signs what the user sees.

* Tamper-evident logging of every Originator signing event, so that compromise can be detected after-the-fact even if it cannot be prevented in real time.

Deployments evaluating ZTIP for a particular threat model SHOULD treat the Originator's signing environment with the same operational rigor they would apply to a CA's root key.

7.2. Additional Adversaries

Adversary	Goal
ADV-PROMPT-INJECT	Controls user-supplied content reaching an agent; wants to substitute the operational intent.
ADV-CHAIN-FORGER	Attempts to forge a Delegation Chain claiming authority from a root Originator the chain recipient trusts.
ADV-SCOPE-EXPANDER	Controls a sub-agent in the middle of a chain; attempts to grant downstream agents broader scope than the parent layer authorized.
ADV-DEPTH-FLOODER	Constructs deeply-nested chains to consume verifier resources (CPU, stack, signature-verification time) at the receiver.

Table 8

7.3. Attack Surface and Mitigations

Attack	Mitigation	Reference
Prompt-injected confused deputy	Intent-Scoped Authorization Tokens (Section 4) bind authorization to the original signed intent. The injected instruction substitutes a new intent that does not match intent_hash; gated operations fail with INTENT_SCOPE_MISMATCH.	Section 4
Chain forgery	Each layer's signature is verified against the issuing principal's published key (Section 3.3 rule 1). An adversary cannot forge a layer without the corresponding private key.	Section 3.3

Scope expansion mid-chain	Scope monotonicity enforcement (Section 3.3 rule 3 and Section 3.4) rejects any chain in which a child layer grants broader scope than its parent.	Section 3.3, 3.4
Chain replay	Chain layers carry iat/exp; receivers MUST reject expired layers (Section 3.3 rule 5). The root Signed Intent's jti can be tracked to prevent intent replay across sessions.	Section 3.3
Originator key compromise	Outside ZTIP's defense — same status as Issuer key compromise in any signing-based system. Mitigated by short Signed-Intent lifetimes and Originator-side HSM use.	(Out of scope)
Behavioral-claim misrepresentation	Behavioral claims (Section 5) are signed by whatever credential issuer carries them. Verifiers MAY require specific issuers known to assess behavioral properties rigorously.	Section 5
Chain-depth flood / verifier resource exhaustion	Maximum chain depth enforcement (Section 3.5). Depth MUST be checked before signature verification to prevent stack/ CPU exhaustion.	Section 3.5

Table 9

7.4. Out of Scope

ZTIP does not defend against:

* *Compromise of the Originator before they sign the intent.* The Originator's signing environment is the trust root; if it is compromised, the adversary can sign arbitrary intents that downstream verifiers will treat as legitimate. Mitigations are out-of-protocol (HSM, hardware-bound signing, multi-factor approval at signing time).

* *Side-channel observation of intent contents.* The intent_object is visible to every verifier in the chain (each verifier needs to recompute the hash). Information that should not be visible in transit MUST be redacted from the canonical intent before signing — see Privacy Considerations (Section 8) for guidance.

8. Privacy Considerations

This section is informed by [RFC6973].

Intent string sensitivity. The canonical-intent-string (the source of intent_hash) describes what a user asked the system to do. It MAY be sensitive (medical, legal, financial). Implementations MUST NOT log canonical-intent-strings without retention controls. The intent_hash is non-reversible and MAY be logged.

Intent visibility is a structural tradeoff, not just a privacy concern. Verifiable intent binding requires every verifier in the chain to see the intent contents (or a hash thereof, with downstream lookup). For deployments where intent contents are themselves sensitive — financial transaction details, medical context, legal strategy, regulated business data — this visibility is a deployment-feasibility constraint, not a privacy nicety. Some enterprise deployments may reject intent-binding altogether on this ground; deployments evaluating ZTIP for adoption SHOULD plan their intent schema with this constraint in mind from the outset. The mitigations below shape what can be carried in intent_object without leaking; they do not remove the underlying tradeoff between verifiability and confidentiality.

Intent contents are visible to all verifiers. Every verifier in the chain receives the root layer's intent_object (it must, in order to recompute the hash and evaluate intent_scope). The intent contents are therefore visible to every layer between Originator and final verifier. Originators MUST NOT include in intent_object any information that must remain confidential to a subset of verifiers. Three patterns help:

* *Field redaction at signing time.* The Originator's signing application removes sensitive details from intent_object before canonicalization, signing only what verifiers need to evaluate scope. The user-readable intent stays in the user's local logs, not in the signed chain.

* *Reference-by-hash.* Sensitive payloads are hashed; the chain carries only the hash. Downstream agents look up the actual content via a separate authenticated channel.

* *Profile-specific encryption.* Some profiles may layer envelope encryption over intent_object for verifiers who hold the appropriate key. This is out of scope for ZTIP base; profile documents MAY specify it.

Delegation chain disclosure. A Delegation Chain reveals the orchestration topology (which orchestrator, which sub-agents, which tools). Where this is sensitive, deployments MAY truncate chains exposed to external receivers, retaining only the layers the receiver needs to verify.

Behavioral-claim disclosure. Some behavioral claims (e.g., human_in_loop_policy: never) reveal architectural choices that adversaries could exploit. Issuers and Provers SHOULD coordinate on which behavioral claims are appropriate to expose at which trust boundaries.

Originator linkability. A persistent Originator identifier (user:alice) creates linkability across all chains the Originator participates in. Deployments MAY use ephemeral Originator identifiers per-session where the deployment context permits.

9. IANA Considerations

This document requests the following IANA actions.

The ZTIP IANA footprint is intentionally minimal: a single new registry (Denial Reason Codes), a single HTTP field registration (ZTIP-Chain), and zero new registries for behavioral claim names. The single registry uses Specification Required allocation per [RFC8126] — anyone able to write a stable specification document can register a new reason code, with no IETF consensus action required per registration. Composition profile documents are expected to add reason codes for their domain; the Specification Required policy keeps that path open for vendor and consortium specifications without burdening either IANA or the IETF process.

9.1. ZTIP Denial Reason Code Registry

This document requests creation of a registry titled "ZTIP Denial Reason Codes". It administers reason codes returned by ZTIP-aware verifiers when rejecting a request.

Allocation Policy: Specification Required (per [RFC8126]).

Initial Registrations:

Code	Meaning	Reference
DEL_CHAIN_MISSING	Policy requires a Delegation Chain but none was provided	This document, Section 3
DEL_CHAIN_BROKEN	Chain of delegator/ delegatee references has a gap	This document, Section 3
DEL_CHAIN_SCOPE_EXPANDED	A child layer attempts to grant broader scope than its parent	This document, Sections 3, 3.4
DEL_CHAIN_EXPIRED	One or more chain layers has expired	This document, Section 3
DEL_CHAIN_UNTRUSTED_ROOT	The root Signed Intent is not signed by a trusted Originator	This document, Section 3
DEL_CHAIN_DEPTH_EXCEEDED	The chain exceeds the verifier’s configured maximum depth	This document, Section 3.5
INTENT_SCOPE_MISMATCH	The requested operation is outside the scope authorized by the root Signed Intent	This document, Section 4

Table 10

Composition profiles MAY add codes via Specification Required.

9.2. HTTP Field Name Registration

This document requests registration of the following HTTP field name in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" (per [RFC9110] Section 16.3.1):

Field Name	Status	Reference
ZTIP-Chain	permanent	This document, Section 6.2

Table 11

Specification: The ZTIP-Chain HTTP header field carries a Delegation Chain (Section 3) as a single compact-serialized JSON Web Signature [RFC7515] value. It is sent by clients alongside an Authorization: Bearer header in the OAuth 2.0 composition profile (Section 6.2). The field value is the outermost JWS of a Delegation Chain in compact serialization. The field MUST NOT appear more than once in a single request; a request bearing multiple ZTIP-Chain fields MUST be rejected with DEL_CHAIN_BROKEN.

Comments: The author searched the IANA HTTP Field Name Registry as of the date of publication and found no existing field named ZTIP-Chain, no existing field beginning with ZTIP-, and no closely-related field name. The closest prefix-matching field is Client-Cert-Chain, which addresses TLS certificate-chain transport — a different domain from delegation-chain transport.

9.3. Note on Behavioral Claim Names

ZTIP does not request creation of an IANA registry for behavioral claim names. Section 5 specifies a standard set of claims (prompt_injection_tested, tool_call_audit_logged, etc.) and an extension mechanism using URI-namespaced names, vendor-prefixed names, and privately-agreed names. This intentional non-allocation mirrors the framework_id approach in ZTNP: the diversity of bespoke and enterprise behavioral claims makes a central registry impractical, and the namespacing conventions in Section 5 provide collision-free extension without IANA gatekeeping.

10. Implementation Status

This section records known implementations per RFC 7942 and is to be removed before publication as an RFC.

A reference TypeScript implementation of Delegation Chain verification, Intent-Scoped Authorization enforcement, and Behavioral Claim parsing is in progress at:

https://github.com/agent-trust-protocols/agent-trust-protocols/tree/main/reference/typescript/ztip

Test vectors covering valid chains, broken chains, scope-expanded chains, depth-exceeded chains, expired chains, and intent-scope mismatches are at:

https://github.com/agent-trust-protocols/agent-trust-protocols/tree/main/test-vectors/ztip

A Python reference implementation is planned. Composition-profile reference implementations (OAuth, ZTNP) are planned as the protocols stabilize.

11. References

11.1. Normative References

[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>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/rfc/rfc8785>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

11.2. Informative References

[I-D.miller-ztnp]
Miller, J., "Zero-Trust Negotiation Protocol (ZTNP) Core", Work in Progress, Internet-Draft, draft-miller-ztnp-00, n.d., <https://datatracker.ietf.org/doc/html/draft-miller-ztnp-00>.

[Macaroons2014]
Birgisson, A., Politz, J. G., Erlingsson, ., Taly, A., Vrable, M., and M. Lentczner, "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud", NDSS Symposium 2014, 2014, <https://research.google/pubs/pub41892/>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/rfc/rfc3552>.

[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>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[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>.

[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>.

[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>.

[RFC9635] Richer, J., Ed. and F. Imbault, "Grant Negotiation and Authorization Protocol (GNAP)", RFC 9635, DOI 10.17487/RFC9635, October 2024, <https://www.rfc-editor.org/rfc/rfc9635>.

Appendix A. Worked Example: Three-Hop Delegation Chain (ZTNP Composition)

This appendix presents a complete example of a three-party delegation chain using the *ZTNP composition profile* (Section 6.1). It is informative; it does not introduce new normative requirements. Implementers SHOULD verify their implementation against this example.

A.1. Scenario

User Alice asks her orchestrator to "summarize my unread emails from the last 24 hours and produce a markdown digest." The orchestrator delegates the work to a specialized summarizer agent, which in turn calls the email-read tool. Three signing principals participate:

* user:alice — the Originator (root)

* principal:orchestrator-1 — the orchestrator

* agent:summarizer-3 — the summarizer agent

The downstream tool tool:email.read receives the call and verifies the entire chain before serving the request.

A.2. A.1 Layer 0: Signed Intent (Root)

Alice’s signing application produces the following intent_object:

```
{
  "action": "summarize",
  "scope": {
    "actions": ["read"],
    "data": ["internal", "pii"],
    "tools": ["email.list", "email.read"]
  },
  "target": "unread emails from the last 24 hours",
  "constraints": {
    "must_not": ["email.send", "email.delete"]
  }
}
```

After JCS canonicalization per Section 3.2, the canonical bytes are:

```
{"action":"summarize","constraints":{"must_not":["email.send","email.delete"]},"scope":{"
actions":["read"],"data":["internal","pii"],"tools":["email.list","email.read"]},"target"
:"unread emails from the last 24 hours"}
```

The base64url-encoded SHA-256 of these UTF-8 bytes is the intent_hash. For this intent_object the value is:

intent_hash = Q9h_MJaQrDtKRb7MKfwg664jUWmVlErfdS8Qm1y6qNc

This is the same intent_object (and therefore the same hash) as Example 1 in Section 3.2.

The root layer’s full payload (before JWS signing):

```
{
  "del_chain_ver": "0.1",
  "intent_root": true,
  "originator": "user:alice",
  "intent_object": { /* as shown above */ },
  "intent_hash": "Q9h_MJaQrDtKRb7MKfwg664jUWmVlErfdS8Qm1y6qNc",
  "authorized_chain": ["principal:orchestrator-1", "agent:summarizer-3"],
  "scope": { /* same value as intent_object.scope */ },
  "iat": 1745500800,
  "exp": 1745504400,
  "jti": "intent_01HVXYZ_SUMMARIZE_REQUEST"
}
```

This payload is signed with Alice’s private key. The result is a compact JWS denoted JWS_root.

A.3. A.2 Layer 1: Orchestrator’s Delegation

The orchestrator receives JWS_root and delegates to the summarizer agent:

```
{
  "del_chain_ver": "0.1",
  "delegator": "principal:orchestrator-1",
  "delegatee": "agent:summarizer-3",
  "scope_reduction": {
    "actions": ["read"],
    "data": ["internal", "pii"],
    "tools": ["email.list", "email.read"]
  },
  "iat": 1745500850,
  "exp": 1745504400,
  "inner": "<JWS_root as compact JWS>"
}
```

The orchestrator does not narrow scope here. It signs with its own key, producing JWS_orch.

A.4. A.3 Layer 2: Summarizer's Delegation

The summarizer narrows scope (no PII, only email.read):

```
{
  "del_chain_ver": "0.1",
  "delegator": "agent:summarizer-3",
  "delegatee": "tool:email.read",
  "scope_reduction": {
    "actions": ["read"],
    "data": ["internal"],
    "tools": ["email.read"]
  },
  "iat": 1745500900,
  "exp": 1745504400,
  "inner": "<JWS_orch as compact JWS>"
}
```

Both reductions (data: dropping pii; tools: dropping email.list) are valid per the Section 3.4 monotonicity table. The summarizer signs with its own key, producing JWS_sum. This is the outermost layer and is what travels in the PROOF message's delegation_chain field (per the ZTNP composition profile, Section 6.1).

A.5. A.4 Verification Transcript at the Tool

The email.read tool verifies the chain:

Step 1: Unwrap JWS_sum. Verify against agent:summarizer-3's public key (from the IKS, since this is a ZTNP composition).

Step 2: Check expiration: current time = 1745501000, JWS_sum.exp = 1745504400. Not expired.

Step 3: Recurse into JWS_sum.inner. Unwrap JWS_orch. Verify against principal:orchestrator-1's public key.

Step 4: Recurse into JWS_orch.inner. Unwrap JWS_root. Verify against user:alice's public key. user:alice is in the tool's trusted-originators set.

Step 5: Verify root structure: intent_root: true; originator matches trusted set.

Step 6: Verify chain integrity. JWS_orch.delegator is in JWS_root.authorized_chain; JWS_orch.delegatee = JWS_sum.delegator.

Step 7: Verify scope monotonicity (Section 3.4) — all subset checks pass.

Step 8: Verify chain depth: 3 layers, well within the default maximum of 8.

Step 9: Recompute intent_hash from JWS_root.intent_object; compare to JWS_root.intent_hash. Match.

All checks pass.

A.6. A.5 Intent-Scoped Permit Issued Alongside the Chain

Under the ZTNP composition (Section 6.1), an Intent-Scoped Permit covering this scenario:

```
{
  "iss": "https://gateway.example",
  "sub": "agent:summarizer-3",
  "iat": 1745500900,
  "exp": 1745504400,
  "permit_id": "permit_01HVXYZ_INTENT_TEST",
  "constraints": {
    "actions": ["read"],
    "data": ["internal"],
    "tools": ["email.read"]
  },
  "ch_binding": {
    "method": "tls-exporter",
    "label": "EXPORTER-ZTNP-permit-binding",
    "context_hash": "..."
  },
  "intent_hash": "Q9h_MJaQrDtKRb7MKfwg664jUWmVlErfdS8Qm1y6qNc",
  "intent_scope": {
    "actions": ["read"],
    "data": ["internal"],
    "tools": ["email.read"]
  },
  "chain_root_iss": "user:alice",
  "chain_root_jti": "intent_01HVXYZ_SUMMARIZE_REQUEST"
}
```

For a prompt-injected email.send({to: "attacker@example", ...}) call, the intent_scope.tools does not include email.send. The verifier returns INTENT_SCOPE_MISMATCH. This is the prompt-injection-resistance guarantee.

A.7. A.6 Failure Modes

Three modifications that would cause rejection:

1. JWS_sum.scope_reduction.tools includes email.send → DEL_CHAIN_SCOPE_EXPANDED.

2. JWS_orch's signer is unknown to the tool → signature verification fails at Step 3.

3. JWS_root.intent_object differs from what the orchestrator believed → recomputed intent_hash differs → INTENT_SCOPE_MISMATCH.

4. The chain has more than 8 layers → DEL_CHAIN_DEPTH_EXCEEDED.

Machine-readable test vectors corresponding to this example are at test-vectors/ztip/.

Appendix B. Comparison with Macaroons and Related Capability Tokens

This appendix is informative. It compares ZTIP to macaroons [Macaroons2014] and the related capability-with-caveats tokens (biscuits, wafers) that have appeared since. ZTIP and macaroons share an append-only scope-reduction property and are sometimes considered for the same problems; the comparison clarifies where they overlap and where they diverge.

B.1. At a Glance

Property	Macaroons / Biscuits / Wafers	ZTIP Delegation Chain
Authentication primitive	HMAC keyed by issuer (third-party caveats use additional discharge tokens)	Asymmetric digital signature per layer (any JWS signature alg)
Identity of each delegator	Not bound — any token holder can append a caveat	Each layer's delegator and delegatee are signed identifiers
Scope-reduction model	Caveats: predicates ("user = alice", "expires < T") evaluated at	Structured scope fields with the formal subset relation in

	verification	Section 3.4
Originator intent binding	None — the token represents capability, not intent	intent_hash binds the chain to a structured originator intent
Verification depends on shared secret?	Yes (or third-party discharge for cross-domain)	No — verifier needs each delegator's public key, obtained via the deployment's Key Source
Token format	Self-contained capability token	Layered JWS attached alongside an Authorization Token
Composes with existing token formats?	Generally replaces them	Extends OAuth 2.0, GNAP, ZTNP, or vendor tokens

Table 12

B.2. When Macaroons Fit Better Than ZTIP

* Closed deployments where issuer and verifier share an HMAC key and no third party participates in delegation.

* Single-trust-domain capability stores where "anyone holding the token may further restrict it" is the desired semantics.

* Resource-constrained verifiers that cannot perform asymmetric signature verification.

* Use cases where the _who_ of each delegation step is irrelevant — only the _what_ (the resulting set of permitted operations) matters.

B.3. When ZTIP Fits Better Than Macaroons

* Multi-principal delegation chains where each delegating principal (user, orchestrator, sub-agent) has its own identity that the verifier needs to verify and audit.

* Deployments where the originator’s _intent_ is a first-class security property — that is, where the verifier needs to reject operations consistent with the token’s stated scope but inconsistent with what the user actually asked for. This is the prompt-injection / confused-deputy mitigation in Section 4.

* Deployments composing with existing PKI, JWT-based authorization, OAuth 2.0, or GNAP, where introducing a parallel HMAC trust system would be operationally undesirable.

* Audit and forensic environments where reconstructing "which entity authorized which step" must be possible from the chain itself.

B.4. Conceptual Lineage

ZTIP’s append-only scope reduction (Section 3.4) shares its philosophical lineage with macaroons. Both encode the principle that delegation may only narrow, never broaden, what was authorized. ZTIP’s contribution is to (a) bind the chain’s root to a _structured intent_ the originator signed, (b) require each layer to be signed by an identified principal rather than anyone holding the token, and (c) compose with existing authorization-token formats rather than replacing them. The macaroons paper’s caveat predicates also influenced ZTIP’s design of the constraints field on intent_object (Section 3.2), which is profile-extensible to support arbitrary additional predicates beyond the structured scope dimensions.

Appendix C. Conformance Profile

ZTIP’s three primitives (Delegation Chain Attestation, Intent-Scoped Authorization, Behavioral Claim Extensions) are independently useful, and deployments MAY adopt one without the others (Section 1). The conformance profile defines two conformance levels accordingly.

Full conformance. An implementation is fully conformant with ZTIP if it implements all three primitives:

* Produces and validates Delegation Chain JWS structures per Section 3.

* Enforces scope monotonicity across delegation layers per Section 3.4.

* Enforces a configured maximum chain depth per Section 3.5.

* Issues and enforces Intent-Scoped Authorization Tokens per Section 4.

* Carries ai_behavior Behavioral Claims when applicable per Section 5.

* Implements at least one composition profile from Section 6.

Partial conformance. An implementation supporting only a subset of the three primitives MAY claim partial conformance by listing the primitives it implements (e.g., "ZTIP partial: Delegation Chain Attestation only" or "ZTIP partial: Delegation Chain Attestation + Intent-Scoped Authorization, no Behavioral Claims"). Partial-conformant implementations of any one primitive MUST satisfy that primitive's normative requirements in full — partial conformance refers to which primitives are implemented, not to weakening the requirements within a chosen primitive.

All conforming implementations (full or partial) MUST declare which composition profiles they support.

Acknowledgments

The author thanks the early reviewers for feedback on multi-principal delegation. ZTIP's design draws on the confused-deputy literature and on emerging research into prompt-injection mitigations. The composition-profile pattern follows the precedent set by DPoP [RFC9449].

Author's Address

Jake Miller
Email: jake@zivis.ai