Network Working Group
Internet-Draft
Intended status: Informational
Expires: 22 November 2026

R. Nelson
Authproof

21 May 2026

Delegation Receipt Protocol for AI Agent Authorization
draft-nelson-agent-delegation-receipts-09

Abstract

This document defines the Delegation Receipt Protocol (DRP), a cryptographic authorization primitive for AI agent deployments. Before any agent action executes, the authorizing user signs an Authorization Object containing scope boundaries, time window, operator instruction hash, and model state commitment. This signed receipt is published to an append-only log before the agent runtime receives control. The protocol reduces reliance on the operator as a trusted intermediary by making the user's private key the sole signing authority over the delegation record.

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 22 November 2026.

Copyright Notice

Table of Contents

1. Introduction

Agentic AI systems execute actions on behalf of human principals using natural language instructions as their primary authorization artifact. This creates a structural gap between the authorization a user believes they granted and the instructions an operator delivers to the agent at runtime. No existing cryptographic mechanism makes that gap detectable.

This document specifies the Delegation Receipt Protocol (DRP), a cryptographic authorization primitive that addresses this gap. DRP requires every agent action to be preceded by a user-signed Authorization Object -- the Delegation Receipt -- anchored to a tamper-evident append-only log. The receipt commits the user's authorized scope, operational boundaries, validity window, and a cryptographic hash of the operator's stated instructions. Any deviation by the operator from those instructions is provable from the public log without additional trust assumptions.

DRP is not a replacement for existing IETF agent authorization work. WIMSE, AIP, and OAuth 2.0 Token Exchange [RFC8693] address service-to-agent trust. DRP addresses the upstream layer: user-to-operator trust. In a complete agentic trust stack, these layers are complementary.

A reference implementation of this protocol is available as an open-source SDK at https://github.com/Commonguy25/authproof-sdk under the MIT License. A hosted service implementing the protocol is available at https://cloud.authproof.dev with a free tier requiring no credit card.

1.1. Novel Contributions

This document introduces three cryptographic primitives that do not appear in existing agent authorization frameworks or IETF drafts:

Model State Attestation (Section 7):

The delegation receipt is bound to a cryptographic measurement of the model state at authorization time. If the operator substitutes a different model after the user signs the receipt, the measurement changes and execution is blocked. This closes the operator model substitution attack vector that existing frameworks do not address. The protocol distinguishes between malicious substitution (always blocked) and provider updates (requires reauthorization) using the ProviderUpdate vs MaliciousSubstitution classification defined in Section 7.3.

Scope Discovery Protocol (Section 8):

Before authorization, the agent runs in a sandboxed observation mode with no real resource access. It simulates the intended task and records every resource it attempts to access. This produces a draft ScopeSchema grounded in actual agent behavior rather than operator-specified assumptions. The user reviews a plain-language summary and signs only what they explicitly approve. This closes the upstream design-time gap where users cannot accurately specify scope before understanding agent behavior.

Session State and Adaptive Authorization (Section 9):

A continuously updated trust score tracks behavioral anomalies across the session lifetime. Trust decays on anomaly detection and recovers slowly on clean behavior. Decision thresholds tighten automatically as trust degrades. Sessions suspend when trust falls below a configurable floor, requiring explicit user reauthorization. This extends the static pre-execution authorization model to cover dynamic session-level risk that cannot be captured at delegation time.

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.

The following terms are used throughout this document:

Delegation Receipt:
: A signed Authorization Object produced by the User prior to any agent action. Contains scope, boundaries, time window, and operator instruction hash. *MUST* be anchored to an append-only log before the agent runtime receives control.

Authorization Object:
: The canonical JSON body that is signed to produce a Delegation Receipt. The receipt ID is the SHA-256 hash of this body in its canonical serialization.

User:
: The human principal whose resources and authority are being delegated. The User's private key is the sole signing authority for Delegation Receipts.

Operator:
: The developer or organization that builds and deploys the agent. The Operator provides instructions to the agent and is bound by the instruction hash committed in the receipt.

Agent:
: The AI system taking actions on behalf of the User. The Agent *MUST* verify a valid Delegation Receipt before executing any action.

Append-Only Log:
: A tamper-evident ledger to which Delegation Receipts are anchored prior to execution. Implementations *SHOULD* use a decentralized transparency log following the Certificate Transparency model.

Log Anchor:
: An inclusion proof returned by the append-only log after a receipt is submitted. The log anchor establishes the authoritative issuance timestamp.

Scope:
: An explicit allowlist of permitted operations embedded in a Delegation Receipt. Operations are classified as reads, writes, deletes, or executes. All operations not listed are denied by default.

Boundaries:
: Explicit prohibitions embedded in a Delegation Receipt that survive any subsequent Operator instruction. Boundaries *MUST NOT* be waived or overridden by the Operator.

Instruction Hash:
: The SHA-256 hash of the Operator's stated instructions at delegation time. Any change to the Operator's instructions after receipt issuance is detectable by recomputing this hash.

Micro-Receipt:
A minimal Delegation Receipt covering a single action not included in the parent receipt's scope. *MUST* reference the parent receipt hash.

Model State Commitment:
A cryptographic measurement binding a specific model identity, version, system prompt hash, and runtime configuration hash to a Delegation Receipt.

Scope Discovery:
A protocol that derives authorized scope from sandboxed observation of agent behavior rather than upfront user specification.

Session State:
A live, stateful risk evaluation layer that tracks trust decay, sensitivity classification, and behavioral anomalies across the lifetime of an agent session.

3. Problem Statement

3.1. The Agentic Delegation Chain

Agentic AI systems involve at minimum three principals:

* User -- the human whose resources and authority are delegated.

* Operator -- the developer or company that builds and deploys the agent.

* Agent -- the AI system taking actions on the User's behalf.

The delegation chain is:

User --> Operator --> Agent --> Services

The User grants authority to the Operator. The Operator translates that authority into instructions for the Agent. The Agent acts on downstream services. At each step, fidelity to the User's original intent depends entirely on the honesty and competence of the intermediate party.

3.2. The Missing Cryptographic Anchor

In current agentic deployments, the User's authorization is captured in natural language -- a chat message, a consent checkbox, a terms-of-service agreement. None of these produce a cryptographically verifiable record of what the User actually authorized at the moment of delegation.

This creates three compounding problems:

The repudiation problem:
: If an agent takes an action the User did not authorize, there is no cryptographic evidence of what the User did authorize. The Operator's account of the authorization is the only record, and it is unverifiable.

The drift problem:
: Operators may update system prompts, change agent behavior, or respond to external pressure in ways that diverge from the User's original authorization. Nothing in the current architecture makes this divergence detectable.

The audit problem:
: Regulators auditing agentic behavior have no evidence chain connecting agent actions to original user consent. The Operator's logs are the only source of truth, controlled by the party whose conduct is under scrutiny.

3.3. IETF Framework Analysis

Several IETF working groups have produced or are producing specifications for agent identity and authorization. Each addresses a different trust boundary; none addresses user-to-operator trust.

WIMSE (Workload Identity in Multi-System Environments) addresses service-to-service authentication: can service B verify that a request came from legitimate workload A? It does not address whether the workload was authorized by the User to make that request in the first place.

AIP (Agent Identity Protocol) defines credential structures for agent principals and addresses how agents present identity to services they call. Like WIMSE, its trust model is downstream of the Operator -- it assumes the Operator has correctly represented the User's authorization.

draft-klrc-aiagent-auth addresses OAuth-style authorization flows for AI agents, allowing agents to obtain access tokens for downstream APIs. It solves the service authorization problem -- whether the agent can call an API -- but not the delegation integrity problem -- whether the Operator's instructions faithfully represent the User's authorization.

OAuth 2.0 Token Exchange [RFC8693] and Rich Authorization Requests [RFC9396] provide mechanisms for scoped token issuance and delegation chains between services but operate at the service layer. The User's intent is represented by the OAuth grant, which is under Operator control.

The gap is consistent across all existing frameworks: user-to-operator trust is taken as a precondition. DRP addresses that precondition directly.

4. The Delegation Receipt

4.1. Receipt Structure

A Delegation Receipt is a JSON object with the following *REQUIRED* fields:

receiptId:
: The SHA-256 hash of the canonical serialization of the Authorization Object body (all fields except receiptId and signature). Encoded as the string "rec_" followed by the lowercase hex digest.

schemaVersion:
: Protocol schema version string. This document defines version "1.0".

scope:
: An object with two keys: "allowedActions" and "deniedActions", each containing an array of action descriptors. Each action descriptor is an object with "operation" and "resource" string fields. Actions in "deniedActions" take precedence over "allowedActions". Natural language *MUST NOT* appear in any scope field.

boundaries:
: An array of prohibition strings that *MUST* be enforced regardless of Operator instruction. The array *MUST NOT* be empty; implementations with no explicit prohibitions *SHOULD* populate a conservative default. The *RECOMMENDED* conservative default *MUST* deny all write, delete, and execute operations on resources

not explicitly listed in the receipt's allowedActions: ["deny:write:*", "deny:delete:*", "deny:execute:*"]. Implementations *MUST* document the default boundaries value in use.

timeWindow:
An object with "notBefore" and "notAfter" fields, each an ISO 8601 timestamp. The authoritative time reference is the log timestamp (see Section 5.3), not the client clock.

operatorInstructionsHash:
The SHA-256 hash of the Operator's stated instructions at delegation time, encoded as "sha256:" followed by the lowercase hex digest.

operatorInstructions:
The operatorInstructions field is OPTIONAL in the receipt JSON object, but implementations *MUST* ensure that either (a) the plaintext operator instructions are present in this field, or (b) the instructions are stored via a verifiable off-log commitment mechanism, with the operatorInstructionsHash field providing the cryptographic binding. Verifiers that require plaintext access for audit purposes *SHOULD* reject receipts where this field is absent unless they have access to the off-log store.

canonicalPayload:
The base64url-encoded canonical JSON serialization of all Authorization Object fields except signature. This is the exact byte string over which the ECDSA P-256 signature is computed and verified. Carrying the canonical payload in the receipt allows verifiers to re-verify the signature without reconstructing the canonical form from scratch. The canonicalPayload field *MUST* be computed over all fields of the Authorization Object except canonicalPayload itself and signature. Verifiers *MAY* use this field directly to skip reconstruction, provided they first verify the signature over the encoded value.

publicKey:
The User's public key as a JSON Web Key [RFC7517]. Implementations *SHOULD* use Ed25519 ([RFC8032], crv: "Ed25519"), which is *RECOMMENDED* for all new deployments. ECDSA P-256 (crv: "P-256", [RFC7518]) is also supported for interoperability with deployments that do not support Ed25519.

signature:
The digital signature over the canonical serialization of the Authorization Object body, encoded as base64url. The signature algorithm *MUST* match the algorithm of the publicKey field: ECDSA

P-256 (per [RFC7518]) when publicKey.kty is "EC", or Ed25519 (per [RFC8032]) when publicKey.kty is "OKP". See Section 4.3 for the complete signing procedure.

The following *OPTIONAL* fields are defined by this document:

teeMeasurement:
Model state commitment object binding the receipt to a specific TEE enclave measurement. See Section 7.

modelCommitment:
SHA-256 measurement of the model state at authorization time, formatted as "sha256:" followed by the lowercase hex digest. When present, verification *MUST* fail if the current model measurement does not match. See Section 7.

scopeSchema:
Machine-readable structured allowlist and denylist derived from the Scope Discovery Protocol. See Section 8.

toolSchemaHash:
The SHA-256 hash of the canonical serialization of the complete set of tool schemas available to the agent at delegation time, formatted as "sha256:" followed by the lowercase hex digest. When present, Check 11 of the verification algorithm (Section 6.4) *MUST* recompute this hash at execution time and fail if it does not match. Detects tool schema changes made after authorization was granted.

discoveryMetadata:
Metadata recorded during a Scope Discovery Protocol observation session (Section 8), including observationCount, abortedByTimeout, and riskFlags. Provides an auditable trail from observation to receipt issuance. *MUST* be present when the receipt was produced by the Scope Discovery Protocol.

trustedSources:
An array of strings identifying instruction sources that are permitted to drive agent behavior. Typical values include "user", "system_prompt", and "verified_tool". When present, Check 13 of the verification algorithm (Section 6.4) *MUST* reject any action whose instructionSource is not in this list. See Section 6.1 for the prompt injection threat.

parentReceiptId:
The receiptId of the Orchestrator’s Delegation Receipt that authorized creation of this sub-receipt. When present, the receipt is treated as a sub-receipt and subjected to the parent

scope containment check (Check 14). When absent, the receipt is treated as a root receipt and requires a User signature. See Section 10.

orchestratorSignature:
An ECDSA P-256 signature by the Orchestrator's key over the binding string "orchestrator-delegation:" || parentReceiptId || ":" || receiptId. Present only in sub-receipts. Not included in the signed body. See Section 10.1.

metadata:
An *OPTIONAL* object containing implementation-defined key-value pairs. The metadata object is covered by the receipt signature (i.e., included in the canonicalPayload). Implementations *MAY* include arbitrary metadata; verifiers that do not recognize metadata keys *MUST* ignore them. Metadata keys beginning with "x-" are reserved for private use.

providerUpdatePolicyId:
OPTIONAL. A string identifier referencing the providerUpdatePolicy configuration entry that governed this delegation at issuance time. Including this field allows verifiers and auditors to determine, from the receipt alone, which update policy was in effect without accessing operator-side configuration. The value is an opaque string scoped to the Operator's trust anchor; its format is implementation-defined. This field is covered by the receipt signature.

All fields defined in this section, including all *OPTIONAL* fields, are included in the canonical serialization of the Authorization Object body and are therefore covered by the receipt signature. Once signed, no field may be added, removed, or altered without invalidating the signature. The sole exception is orchestratorSignature, which is explicitly excluded from the signed body as noted in its field definition above.

The complete structure is illustrated below:

```
{
  "receiptId":               "rec_<lowercase-hex-of-canonical-body>",
  "schemaVersion":           "1.0",
  "scope": {
    "allowedActions": [
      { "operation": "<op>", "resource": "<resource>" }
    ],
    "deniedActions": [
      { "operation": "<op>", "resource": "<resource>" }
    ]
  },
  "boundaries": ["<prohibition-string>"],
  "timeWindow": {
    "notBefore": "<ISO-8601-timestamp>",
    "notAfter":  "<ISO-8601-timestamp>"
  },
  "operatorInstructionsHash": "sha256:<hex-digest>",
  "operatorInstructions":     "<operator-instruction-text>",
  "publicKey":                { "<JWK per RFC 7517>" },
  "signature":                "<base64url-ecdsa-p256-signature>"
}
```

4.2. Canonical Serialization

The canonical serialization of a receipt body is defined as follows:

1. Serialize the Authorization Object as JSON with keys sorted in lexicographic ascending order at every nesting level.

2. Remove all insignificant whitespace (no spaces, no newlines outside of string values).

3. Encode the result as UTF-8.

4. Apply Unicode NFC normalization to all string values before serialization. All string values in the Authorization Object *MUST* be normalized to Unicode Normalization Form C (NFC) prior to JSON encoding. Implementations *MUST NOT* produce or accept canonical payloads containing strings in non-NFC normal form.

5. Arrays *MUST* be serialized in the order they appear in the schema definition. Array elements *MUST NOT* be reordered during canonicalization. The allowedActions and deniedActions arrays *MUST* preserve insertion order.

6. Floating-point values *MUST* be serialized using the shortest decimal representation that round-trips per IEEE 754 double-precision semantics. Implementations *MUST NOT* produce trailing zeros, unnecessary exponent notation, or representations that do not round-trip to the same IEEE 754 value.

The canonical computation excludes the canonicalPayload field itself and the signature field. All other fields present in the Authorization Object at signing time *MUST* be included.

Implementations *SHOULD* follow the JSON Canonicalization Scheme defined in [RFC8785] as a compatible reference implementation of the canonicalization requirements in this section.

The receiptId is computed as:

```
receiptId = "rec_" ||
               lowercase_hex(SHA-256(canonical_body))
```

Implementations *MUST* compute the receiptId over the body before the signature field is added. The signature field *MUST NOT* be included in the data that is signed.

4.3. Signing Procedure

Receipt issuance *MUST* follow this sequence:

1. The Operator presents their intended instructions to the User, along with the proposed scope, boundaries, and time window.

2. The User reviews the scope, boundaries, time window, and Operator instructions.

3. The User signs the canonical Authorization Object body using their private key. Two signing algorithms are defined by this document:

   * Ed25519 (per [RFC8032]): *RECOMMENDED* for all new deployments due to smaller key and signature sizes, simpler implementation, and resistance to certain ECDSA implementation pitfalls. Ed25519 public keys are carried as JWK with kty: "OKP" and crv: "Ed25519".

   * ECDSA P-256 (crv: "P-256", per [RFC7518]): the baseline algorithm supported by all conforming implementations. *REQUIRED* for implementations that must interoperate with deployments that do not support Ed25519.

Signing via the WebAuthn/FIDO2 API [W3C-WebAuthn] [FIDO2] is one valid mechanism; hardware key custody (Trusted Platform Module or device secure enclave) is *RECOMMENDED* for production deployments requiring non-repudiation guarantees, but is not required by this specification.

4. The signed receipt is submitted to a decentralized append-only log.

5. The log assigns a timestamp and returns a log anchor (inclusion proof).

6. No agent action *MAY* begin until the log anchor is confirmed.

The log timestamp established in step 5 is the authoritative issuance time. Client clocks *MUST NOT* be used as the time reference for receipt validation.

5. The Append-Only Log

5.1. Log Entry Structure

Each entry in the append-only log *MUST* contain:

* The Delegation Receipt hash (receiptId).

* The SHA-256 hash of the preceding log entry (for chain linking; see Section 5.2).

* A trusted timestamp conforming to [RFC3161].

* The submitter’s public key hash.

* A monotonically increasing entry sequence number.

Implementations *SHOULD* use a log format compatible with Certificate Transparency [RFC6962] to enable standard log consistency verification.

Action log entries produced during agent execution follow the same structure. Each action log entry *MUST* include:

* The receipt hash authorizing the action.

* The action type, payload hash, and destination.

* The SHA-256 hash of the preceding action log entry.

* An RFC 3161 timestamp and the agent's ECDSA P-256 signature over the entry body.

5.2. Chain Linking

Each log entry *MUST* include the SHA-256 hash of the immediately preceding entry. This chain structure guarantees:

1. Log entries cannot be inserted retroactively without producing a detectable chain break.

2. Log entries cannot be deleted without invalidating the chain hash of the subsequent entry.

3. Any two parties holding the same entry hash can verify independently that they share the same log view up to that entry.

The chain structure makes it impossible to insert or delete individual action records without producing a detectable inconsistency that any log monitor can identify.

5.3. Timestamp Authority

Implementations *MUST* use an RFC 3161 [RFC3161] Time-Stamp Authority (TSA) to produce the authoritative timestamp for each Delegation Receipt anchored to the log. The TSA timestamp:

1. Establishes the authoritative notBefore time for the associated Delegation Receipt.

2. Is used in lieu of the client clock for all time window validation (see Section 6.1).

3. Is included in the log anchor returned to the submitter.

If the TSA is unreachable, implementations *MAY* record a local-clock timestamp marked "UNVERIFIED_TIMESTAMP". An agent verifier *MUST* treat UNVERIFIED_TIMESTAMP as insufficient evidence of authorization time in production deployments.

Implementations performing time-window validation (Section 6.4 Check 3) MUST account for clock skew between the verifier's local clock and the TSA's clock. A tolerance window of up to 300 seconds (5 minutes) is *RECOMMENDED*. Implementations *MAY* configure a tighter tolerance; the configured skew tolerance *MUST* be documented and communicated to all parties in the delegation trust chain. Verifiers *MUST NOT* accept a receipt whose timeWindow.notAfter value, even accounting for the full configured skew tolerance, has elapsed.

5.4. Denied Call Logging

Implementations *MUST* log all verification decisions including those that result in a DENY outcome. Logging only PERMIT decisions creates a blind spot for detecting prompt injection and model drift.

The distribution of denied calls over time is a leading indicator of adversarial activity. A model being prompt- injected will generate denied calls with novel action classes and scope edge probing that differs detectably from normal operation. Post-hoc analysis of the deny path is the primary forensic signal for incident reconstruction.

Denied call log entries *MUST* include the full call context, the specific denial reason code, and the session risk score at the time of denial. Implementations *SHOULD* expose denied call distribution analytics to enable real-time anomaly detection.

6. Pre-Execution Verification

6.1. Verification Checks

Before executing any action, the Agent *MUST* perform all of the following checks in order. All checks *MUST* pass; any single failure *MUST* abort the action without partial execution.

Check numbers match Section 6.4 (Section 6.4); optional checks appear only when the relevant receipt fields are present.

```
Verify(receipt, action):
  (1) if Revoked(receipt.receiptId)
                                      -> fail
  (2) if not VerifySig(receipt.signature,
                       canonical(receipt.body),
                       receipt.publicKey)
                                      -> fail
  (3) if not InTimeWindow(receipt.timeWindow,
                          logTimestamp)
                                      -> fail
  (4) if not InScope(action, receipt.scope)
                                      -> fail
  (5) if ViolatesBoundary(action, receipt.boundaries)
                                      -> fail
  (6) if Hash(ExecutionGraph(action.program))
         != receipt.scope.executes[action.programIndex]
                                      -> fail
  (7) if Hash(currentOperatorInstructions)
         != receipt.operatorInstructionsHash
                                      -> fail
  (8) [model state attestation -- if receipt.modelCommitment present,
       see Section 6.4 CHECK 8]
  (9) [session risk evaluation -- conditional on sessionState,
       see Section 6.4 CHECK 9]
  (10) [nonce replay detection -- conditional on receipt.nonce,
        see Section 6.4 CHECK 10]
  (11) if receipt.toolSchemaHash PRESENT and
          SHA-256(canonical(currentToolSchema))
          != receipt.toolSchemaHash
                                      -> fail
  (12) if receipt.toolOutputHash PRESENT and
          action.toolOutput PRESENT and
          SHA-256(action.toolOutput)
          != receipt.toolOutputHash
                                      -> fail
  (13) if receipt.trustedSources PRESENT and
           action.instructionSource PRESENT and
           action.instructionSource NOT IN receipt.trustedSources
                                      -> fail
  return true
```

The revocation pre-check (step 1) *MUST* be performed before any other check. A revoked receipt *MUST* fail immediately regardless of whether other checks would pass.

6.2. Check Ordering

The check ordering reflects distinct security properties; each step eliminates a distinct attack surface.

Revocation check (1):
: Ensures a receipt explicitly invalidated by the User cannot authorize further actions, regardless of its cryptographic validity.

Signature check (2):
: Confirms the receipt was signed by the holder of the User's private key and has not been altered since signing. Any tampering with the receipt body invalidates the signature under ECDSA P-256.

Time window check (3):
: Validates the action against the log-assigned TSA timestamp, not the client clock. Prevents time manipulation attacks in which an agent extends its own authorization window by adjusting local time.

Scope check (4):
: Enforces the deny-by-default allowlist. If the action is not explicitly listed, it *MUST* fail regardless of Operator instruction.

Boundary check (5):
: Enforces the User's hard limits, which survive any subsequent Operator instruction or override.

Execution hash check (6):
: Verifies that Hash(ExecutionGraph(action.program)) matches the committed hash in receipt.scope.executes for the requested program. Applies only when action.type is EXECUTE.

Instruction hash check (7):
: Compares the SHA-256 hash of the Operator's current instructions against the hash committed at delegation time. If the Operator has changed its instructions since the receipt was issued, the mismatch is immediately detectable from the log entry, with no reliance on the Operator's own account.

Model state attestation check (8):
When the receipt contains a modelCommitment field, recomputes the model state measurement at execution time and compares it to the committed value. A mismatch indicates that the model was substituted or updated after authorization was granted. Returns MALICIOUS_MODEL_SUBSTITUTION or PROVIDER_UPDATE_REQUIRES_REAUTH on failure.

Session risk evaluation check (9):
When session state is present, evaluates the cumulative session risk score against the configured thresholds. If the session trust score has fallen below the block threshold or tauSession has been exhausted, the action is denied. Returns SESSION_RISK_THRESHOLD_EXCEEDED or TAU_SESSION_EXHAUSTED on failure.

Replay detection check (10):
Records each presented receiptId in a per-session presentation log. If the same receiptId is presented more than once within the same session, the action is denied. Prevents a captured receipt from being re-presented in a concurrent or replayed context. Returns REPLAY_DETECTED on failure.

Tool schema integrity check (11):
When the receipt contains a toolSchemaHash field, recomputes the SHA-256 hash of the canonical serialization of all tool schemas currently available to the agent. A mismatch indicates that the tool set changed after authorization was granted. Returns TOOL_SCHEMA_DRIFT on failure.

Tool output hash check (12):
When the receipt contains a toolOutputHash field, the verifier computes the SHA-256 hash of the tool output that triggered the action and compares it to the committed value. A mismatch indicates that the tool output was altered between delegation time and execution time. Returns TOOL_OUTPUT_TAMPERED on failure.

Instruction provenance check (13):
When the receipt contains a trustedSources field, the verifier inspects the instructionSource field of the action. If the source is not in the trustedSources list, the action is denied. This check is the primary defense against prompt injection attacks in which a poisoned document or untrusted tool output manipulates the agent into requesting an in-scope action for malicious purposes. Returns UNTRUSTED_INSTRUCTION_SOURCE on failure.

Parent scope containment check (14):
Applied only to sub-receipts carrying a parentReceiptId field. Confirms that the sub-receipt's scope is a strict subset of the parent's scope, that the time window is contained within the parent's, that the chain depth does not exceed maxChainDepth, and that the orchestrator binding signature is valid. Returns PARENT_SCOPE_VIOLATION on failure.

6.3. Failure Handling

When any verification check fails, the Agent *MUST*:

1. Abort the action immediately. No partial execution is permitted.

2. Record the failure in the action log, including the specific check that failed and the reason string.

3. Not fall back to Operator instruction. A failed verification check *MUST NOT* be overridden by any runtime parameter, environment variable, or Operator-supplied flag.

4. Surface the failure to the User when the failing check is one of: revocation (1), instruction hash mismatch (7), or execution hash mismatch (6). These failures indicate possible Operator deviation from the committed authorization and *SHOULD* be escalated.

The pause-and-request behavior described in this section is an implementation-layer optimization that occurs after the verifier returns DENY with reason SCOPE_VIOLATION. It does not modify the verification algorithm defined in Section 6.4; the action remains denied until the agent presents a new receipt that passes all checks.

When a scope check (4) fails because an action is outside the current receipt's scope, the Agent *MAY* pause execution and request a Micro-Receipt from the User covering the specific action. The Micro-Receipt *MUST* reference the parent receipt hash and *MUST* be anchored to the append-only log before the action is attempted.

Implementations *MUST* always make a safe fallback action available when execution is blocked. The designated safe fallback action is NO_OP_WITH_LOG: it performs no operation and writes a full audit log entry containing the denial reason, a snapshot of the session state at the time of denial, and a timestamp. NO_OP_WITH_LOG is unconditionally available regardless of verification state or session trust level. Every DENY decision returned by the gate *MUST* include a safeAlternative field set to NO_OP_WITH_LOG, providing callers with a guaranteed safe path that preserves the audit record without executing any agent action.

6.4. Verification Algorithm

The following pseudocode specifies the complete verification algorithm as a formal function. The checks are numbered 1-14; revocation (Check 1) MUST be performed before any other check.

```
FUNCTION VerifyReceipt(receipt, action, operatorInstructions,
                       sessionState):

  INPUT:
    receipt             : signed delegation receipt object
    action              : the agent action being requested
    operatorInstructions: current operator instruction string
    sessionState        : current session state object (optional)

  OUTPUT:
    PERMIT or DENY with reason code

  CHECK 1: Revocation Status
    IF revocationRegistry.isRevoked(receipt.receiptId) THEN
      RETURN DENY, "RECEIPT_REVOKED"

  CHECK 2: Signature Verification
    IF NOT VerifySignature(receipt.signature,
                           receipt.canonicalPayload,
                           receipt.publicKey) THEN
      RETURN DENY, "INVALID_SIGNATURE"

  CHECK 3: Time Window
    IF receipt.timeWindow.notAfter < NOW() THEN
      RETURN DENY, "RECEIPT_EXPIRED"
    IF receipt.timeWindow.notBefore > NOW() THEN
      RETURN DENY, "RECEIPT_NOT_YET_VALID"

  CHECK 4: Scope Validation
    IF action.operation NOT IN receipt.scope.allowedActions THEN
      RETURN DENY, "ACTION_NOT_IN_SCOPE"
```

```
      // NOTE: implementations MAY pause execution and request a
      // Micro-Receipt from the User rather than returning a hard DENY.
      // This pause-and-request behavior is a runtime-layer optimization
      // described in Section 6.3 and does not alter the verifier's
      // decision; the action remains denied until a valid receipt is
      // presented.
    IF action.operation IN receipt.scope.deniedActions THEN
      RETURN DENY, "ACTION_EXPLICITLY_DENIED"

  CHECK 5: Boundary Check
    FOR EACH prohibition IN receipt.boundaries DO
      IF action MATCHES prohibition THEN
        RETURN DENY, "ACTION_EXPLICITLY_DENIED"

  CHECK 6: Execution Hash Check (if EXECUTE action)
    IF action.type == EXECUTE THEN
      IF Hash(ExecutionGraph(action.program)) !=
         receipt.scope.executes[action.programIndex] THEN
        RETURN DENY, "EXECUTION_HASH_MISMATCH"

  CHECK 7: Operator Instruction Hash
    currentHash = SHA-256(canonicalize(operatorInstructions))
    IF currentHash != receipt.operatorInstructionsHash THEN
      RETURN DENY, "OPERATOR_INSTRUCTIONS_MISMATCH"

  CHECK 8: Model State Attestation
    IF receipt.modelCommitment IS PRESENT THEN
      currentMeasurement = measureModelState()
      IF currentMeasurement != receipt.modelCommitment THEN
        IF modelSubstitutionDetected(receipt,
                                     currentMeasurement) THEN
          RETURN DENY, "MALICIOUS_MODEL_SUBSTITUTION"
        ELSE
          RETURN DENY, "PROVIDER_UPDATE_REQUIRES_REAUTH"

  CHECK 9: Session Risk Evaluation (if sessionState present)
    IF sessionState IS PRESENT THEN
      riskResult = evaluateSessionRisk(action, sessionState)
      IF riskResult.decision == "BLOCK" THEN
        RETURN DENY, "SESSION_RISK_THRESHOLD_EXCEEDED"
      IF riskResult.decision == "REQUIRE_APPROVAL" THEN
        RETURN REQUIRE_APPROVAL, riskResult.reasons

  CHECK 10: Replay Detection
    IF presentationLog.contains(receipt.receiptId, sessionId) THEN
      RETURN DENY, "REPLAY_DETECTED"
    presentationLog.record(receipt.receiptId, sessionId)
```

```
  CHECK 11: Tool Schema Integrity (if toolSchemaHash present)
    IF receipt.toolSchemaHash IS PRESENT THEN
      currentHash = SHA-256(canonicalize(currentToolSchema))
      IF currentHash != receipt.toolSchemaHash THEN
        RETURN DENY, "TOOL_SCHEMA_DRIFT"

  CHECK 12: Tool Output Hash Binding
    IF receipt.toolOutputHash IS PRESENT AND
       action.toolOutput IS PRESENT THEN
      currentHash = "sha256:" || SHA-256(action.toolOutput)
      IF currentHash != receipt.toolOutputHash THEN
        RETURN DENY, "TOOL_OUTPUT_TAMPERED"

  CHECK 13: Instruction Provenance
    IF receipt.trustedSources IS PRESENT AND
       action.instructionSource IS PRESENT THEN
      IF action.instructionSource NOT IN receipt.trustedSources THEN
        RETURN DENY, "UNTRUSTED_INSTRUCTION_SOURCE"

  CHECK 14: Parent Scope Containment (sub-receipts only)
    IF receipt.parentReceiptId IS PRESENT THEN
      parentReceipt = delegationLog.get(receipt.parentReceiptId)
      IF parentReceipt IS NULL THEN
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

      // Enforce chain depth limit
      // Depth-counting traverses parentReceiptId links iteratively
      // (not recursively). Each hop reads only the receiptId and
      // parentReceiptId fields; it does NOT re-run the full
      // verification algorithm on ancestor receipts.
      depth = countChainDepth(receipt, delegationLog)
      IF depth > MAX_CHAIN_DEPTH THEN
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

      // Parent receipt must itself be valid
      IF NOT VerifySignature(parentReceipt.signature,
                             parentReceipt.canonicalPayload,
                             parentReceipt.publicKey) THEN
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

      // Verify orchestrator binding signature when present
      IF receipt.orchestratorSignature IS PRESENT THEN
        bindingStr = "orchestrator-delegation:" ||
                     receipt.parentReceiptId    || ":" ||
                     receipt.receiptId
        IF NOT VerifySignature(receipt.orchestratorSignature,
                               bindingStr,
                               parentReceipt.publicKey) THEN
```

```
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

    // Sub-receipt time window must be within parent time window
    IF receipt.timeWindow.start < parentReceipt.timeWindow.start OR
       receipt.timeWindow.end   > parentReceipt.timeWindow.end THEN
      RETURN DENY, "PARENT_SCOPE_VIOLATION"

    // Every child allowed action must be permitted by the parent scope
    FOR EACH childAction IN receipt.scope.allowedActions DO
      IF NOT parentReceipt.scope.permits(childAction) THEN
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

    // Every parent denied action must be preserved in the child scope
    FOR EACH parentDenied IN parentReceipt.scope.deniedActions DO
      IF NOT receipt.scope.deniedActions.covers(parentDenied) THEN
        RETURN DENY, "PARENT_SCOPE_VIOLATION"

    // Strict proper subset check: compare over EXPANDED concrete sets.
    // Wildcard patterns must be expanded before comparison so that
    // "read:*" and {"read:email", "read:calendar"} are compared
    // correctly under the scope's wildcard semantics (see Section 10.2).
    childActions  = EXPAND(SET(receipt.scope.allowedActions))
    parentActions = EXPAND(SET(parentReceipt.scope.allowedActions))
    IF childActions == parentActions THEN
      RETURN DENY, "SCOPE_NOT_STRICT_SUBSET"

  RETURN PERMIT

END FUNCTION
```

6.5. Denial Reason Codes

When VerifyReceipt returns DENY, implementations *MUST* include one of the following reason codes in the structured failure response:

Reason Code	Description
INVALID_SIGNATURE	Receipt signature verification failed
RECEIPT_REVOKED	Receipt has been explicitly revoked
RECEIPT_EXPIRED	Receipt time window has elapsed
RECEIPT_NOT_YET_VALID	Receipt creation time is in

	the future
ACTION_NOT_IN_SCOPE	Requested action not in allow list
ACTION_EXPLICITLY_DENIED	Requested action in deny list
OPERATOR_INSTRUCTIONS_MISMATCH	Operator instructions hash does not match receipt commitment
MALICIOUS_MODEL_SUBSTITUTION	Model identity changed after receipt was signed
PROVIDER_UPDATE_REQUIRES_REAUTH	Model version updated by provider; reauthorization required
SESSION_RISK_THRESHOLD_EXCEEDED	Session trust score below block threshold
REPLAY_DETECTED	Receipt presented more than once concurrently
TOOL_SCHEMA_DRIFT	Tool schema hash at execution time does not match hash committed at receipt issuance time. Tool specification has changed since authorization was granted.
TOOL_OUTPUT_TAMPERED	The hash of the tool output that triggered the action does not match the hash committed in the receipt at delegation time. The tool output may have been modified between delegation and execution.
UNTRUSTED_INSTRUCTION_SOURCE	The instruction that triggered the action originated from a source not listed in the receipt's trustedSources field. The instruction source may have been manipulated by a prompt injection attack delivered through an untrusted input channel such as a

	retrieved document or external tool output.
TAU_SESSION_EXHAUSTED	Session anomaly capacity exhausted: tauSession has fallen to or below tauMin. Execution blocked regardless of trustScore. Not resettable by reauthorization. A new session established under a new Delegation Receipt begins with tauSession initialized to sessionCapacity (default: 100). Implementations *MUST NOT* carry tauSession state across session boundaries; each new receipt-bounded session receives a fresh tauSession value.
SESSION_LIFETIME_EXCEEDED	Session wall-clock lifetime has exceeded maxLifetimeSeconds. The session MUST be terminated and reauthorization required before further actions may proceed.
SCOPE_NOT_STRICT_SUBSET	Sub-receipt allowedActions is not a strict proper subset of the parent receipt’s allowedActions
PARENT_SCOPE_VIOLATION	Sub-receipt failed parent scope containment check (receipt not found, time window overflow, chain depth exceeded, or orchestrator binding failure)

Table 1

7. Model State Attestation

7.1. Commitment Binding

A valid Delegation Receipt proves that a User authorized an Agent to act within defined scope. It does not prove that the model executing the receipt is the model the User authorized. An Operator could silently substitute a fine-tuned model variant after the receipt is signed; all verification checks would pass because the receipt itself is genuine.

Model State Attestation closes this gap with a two-phase cryptographic protocol that binds the receipt to a measurement of model state at both delegation time and execution time.

Phase 1 -- Commitment (at delegation time):

The Operator commits to the exact model state that will execute. The commitment is a SHA-256 measurement of five components concatenated in canonical order:

```
modelMeasurement = SHA-256(
  normalize(modelId)       ||
  normalize(modelVersion)  ||
  systemPromptHash         ||
  runtimeConfigHash        ||
  receiptHash
)
```

Including receiptHash as the fifth component binds the model measurement to the specific delegation. The same model with the same system prompt but a different receipt produces a different measurement. A commitment *MUST NOT* be reused across delegations.

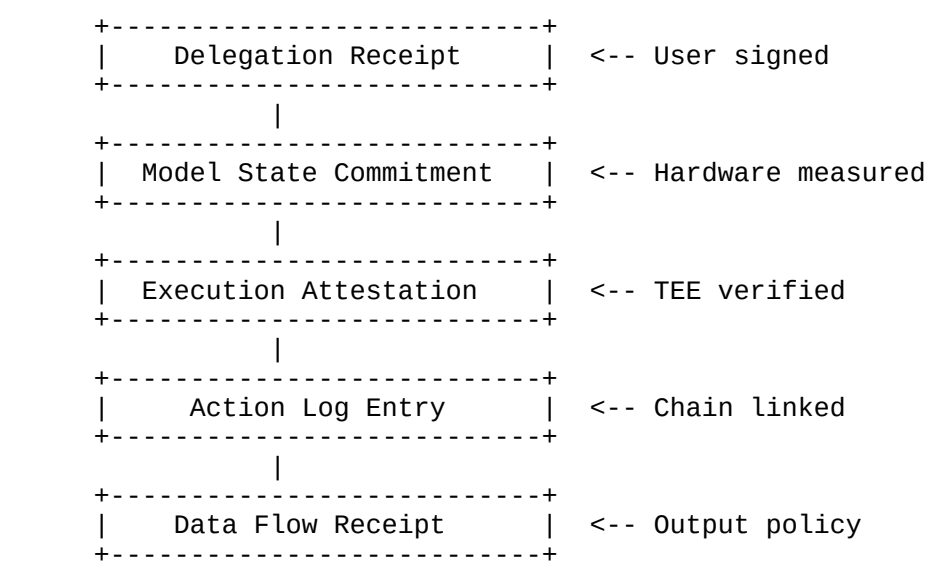
A verifier implementing commitment-reuse detection *MUST* maintain a persistent log of (modelCommitment, receiptId) pairs seen across all validated receipts. When validating a receipt that contains a modelCommitment field, the verifier *MUST* check whether that commitment value already appears in the log under a different receiptId. If a duplicate is found, the verifier *MUST* return DENY with reason "COMMITMENT_REUSE_VIOLATION". Implementors *SHOULD* use a cryptographic set structure (e.g., a Bloom filter backed by a persistent store) to support efficient membership queries at scale.

The commitment is signed by the Operator’s ECDSA P-256 key and attested by the TEE runtime, producing a sealed artifact that includes modelId, modelVersion, systemPromptHash, runtimeConfigHash, committedAt, the Operator’s signature, and a TEE attestation quote. See Section 11.3 for the compliance profiles that govern when TEE attestation is required versus optional.

Phase 2 -- Verification (at execution time):

Immediately before the agent function executes, the current model state is measured using the same five-component computation. The resulting measurement *MUST* equal the committed measurement. If the two measurements differ for any reason, execution *MUST* be blocked with a MALICIOUS_MODEL_SUBSTITUTION denial identifying exactly which components changed.

With Model State Attestation in place, the complete verifiable chain of accountability is:



An auditor presented with a chain proof can verify each layer independently and confirm that a logged action was taken by the model the User authorized, acting within the defined scope, under conditions unaltered since authorization was granted.

7.2. Provider Update Handling

Hosted model providers may silently update the underlying model behind a versioned alias without Operator action. Treating this identically to a deliberate substitution attack is too blunt: it would block legitimate executions whenever a provider retires a model version, forcing operators to recommit on every provider maintenance cycle.

Model State Attestation distinguishes two categories of measurement mismatch:

MaliciousSubstitution:
The Operator explicitly changed the model identifier, system prompt, or runtime configuration after the commitment was signed. This *MUST* always be a hard block. Indicators are any of:

* currentModelId != committedModelId, OR

* currentSystemPromptHash != committedSystemPromptHash, OR

* currentRuntimeConfigHash != committedRuntimeConfigHash

ProviderUpdate:
The model version changed, but the Operator’s configured modelId is unchanged. The provider updated the model behind a stable alias. Indicators are all of:

* currentModelId == committedModelId, AND

* currentModelVersion != committedModelVersion

Operators declare how provider updates are handled at construction time via the providerUpdatePolicy field:

The providerUpdatePolicy is an operator-side configuration parameter and does not appear as a field in the Delegation Receipt JSON object. It is conveyed through the operator’s trust anchor configuration and is not included in the canonical receipt payload or the JSON Schema defined in Appendix A.

"block":
Treat provider updates identically to MaliciousSubstitution. Any version change *MUST* block execution immediately. *RECOMMENDED* when strict model pinning is required.

"reauthorize" (default):
When a provider update is detected, return { allowed: false, reason: "PROVIDER_UPDATE_DETECTED", requiresReauthorization: true } and block all subsequent executions under this attestation instance until the User explicitly acknowledges the change.

The "reauthorize" policy preserves a recovery path. The provider update is flagged and the system halts, but the cause is identified as a non-malicious provider action. A human *MUST* explicitly invoke reauthorize() with userApproval: true before execution resumes. This is an explicit human-in-the-loop checkpoint; the system *MUST NOT* silently resume execution after a provider update.

Operators *SHOULD* populate the providerUpdatePolicyId field of issued receipts with the identifier of the applicable policy entry so that auditors can determine the policy in effect from the receipt record alone.

7.3. Malicious Substitution Detection

The per-component comparison in the Phase 2 verification identifies exactly which aspects of model state changed: model identity, version, system prompt, or runtime configuration. This enables forensic analysis of how an unauthorized execution occurred.

The full SHA-256 measurement comparison is performed in addition to per-component comparison. This is redundant given the component checks but provides a cryptographic guarantee: even if the component comparison logic contains a bug, the measurement comparison will detect any state change.

Model State Attestation proves:

1. Model identity at commitment time. The Operator committed to a specific (modelId, modelVersion, systemPromptHash, runtimeConfigHash) tuple before execution began. The ECDSA signature and TEE attestation prove this commitment was made inside a trusted environment and has not been altered.

2. Model state at execution time. The TEE verification attestation proves the measurement was recomputed inside the enclave immediately before execution, and that the recomputed measurement matched the committed measurement.

3. Delegation binding. The receiptHash component ensures the commitment is irrevocably bound to the specific delegation. A commitment made under receipt A *MUST NOT* be presented as valid under receipt B.

Model State Attestation does not prove that the committed model is safe, aligned, or correctly configured. It does not inspect the content of the system prompt. In simulation mode (the default for testing), attestations are signed with a software ECDSA key rather than produced by real TEE hardware. Production deployments *SHOULD* use Intel SGX, Intel TDX, or ARM TrustZone attestation.

8. Scope Discovery Protocol

The Scope Discovery Protocol addresses the upstream authorization gap: a User cannot correctly define agent scope before observing agent behavior. Asking users to define scope upfront produces one of two failure modes:

Over-authorization:
: The User grants broad permissions to avoid blocking the agent. The agent is then authorized to perform operations the User never intended to permit.

Under-authorization:
: The User grants narrow permissions and the agent fails mid-task, requiring repeated round-trips to expand scope. Users respond by granting progressively wider permissions under frustration.

Neither outcome produces a receipt that reflects the User's actual intent. The scope field becomes a legal fiction rather than a genuine authorization boundary.

The Scope Discovery Protocol inverts the authorization sequence. Instead of asking Users to define scope before running the agent, it runs the agent first in a sandboxed simulation and uses the observed behavior to derive the scope definition.

The protocol proceeds in four stages:

Stage 1 -- Sandboxed observation:
: The agent function is wrapped with transparent proxies for every supported resource type (email, calendar, payment, files, database, network). Every operation call is intercepted, timestamped, and appended to an observation log. Mock data matching the expected structure is returned so the agent proceeds normally. No real I/O occurs; no side effects are produced.

Stage 2 -- Scope generation:
: The observation log is analyzed to produce:

  a. A draftScope with an allowedActions list (de-duplicated observed operations) and conservative deniedActions defaults for delete, execute, and payment operations.

  b. A plainSummary in non-technical language suitable for end-user review.

c. riskFlags for: delete operations, execute operations, payment operations, external send and write operations, and any operation called more than 50 times in the observation session.

d. suggestedDenials for dangerous operations the agent did not use, with per-entry explanations.

Stage 3 -- Plain language review:
The Operator or User reviews the plain summary, risk flags, and suggested denials before approving. An approve() call accepts "remove" and "add" arrays for surgical modification of the draft scope. This is the moment of genuine human authorization, grounded in observed behavior rather than speculation.

Stage 4 -- Cryptographic commitment:
The approved scope is embedded into a Delegation Receipt using the standard signing procedure (Section 4.3). The receipt includes a scopeSchema field with the structured allowedActions and deniedActions lists, and a discoveryMetadata field recording observation count, any timeout abort, and the risk flags at generation time.

The receipt produced by Scope Discovery is structurally identical to one produced by direct issuance. It carries all standard fields and *MUST* be verifiable by the standard Verify procedure (Section 6.1) without modification.

The critical property of observation-based scope generation is grounding: every entry in allowedActions corresponds to an operation the agent actually performed during a representative run. This is a structural record of what the agent did, not a user estimate of what it might need.

Grounding has three practical consequences:

Precision:
The allowedActions list contains exactly the resource/ operation pairs observed. An agent that reads email but never writes it receives a receipt authorizing read on email, not write.

Defensibility:
The discoveryMetadata.observationCount and riskFlags fields provide evidence that scope was derived from observation. The audit trail runs from observation to draft to approval to receipt.

Ratcheting:
: Each time the agent’s behavior changes, a new observation session produces a new draft. If the agent begins calling a new operation class in a new version, that operation surfaces in the risk flags before any receipt is issued for the updated agent. Drift in agent behavior is detectable before it is authorized.

For operators who trust their agent’s observed behavior and do not require manual review, a guided mode provides a single-call end-to-end flow that runs observe, generate, approve, and finalize automatically. The returned riskFlags allow operators to inspect what was flagged even when they choose not to gate on it.

9. Session State and Adaptive Authorization

A Delegation Receipt is a static artifact. It answers one question at one moment in time: did this User authorize this Agent to perform this class of actions? It cannot answer whether a specific action is safe to take right now, given everything that has happened in this session.

Static receipts have three blind spots for long-running sessions:

The drift problem:
: A receipt issued for "manage my calendar" remains technically valid after the agent has sent 400 emails and received a prompt injection payload. The receipt scope string has not changed and cannot reflect runtime events.

The escalation problem:
: A receipt with a generous scope becomes progressively more dangerous as the agent accumulates sensitive data and accesses external services. Risk is not static -- it depends on what came before.

The injection problem:
: A receipt cannot detect that the agent’s input pipeline has been compromised mid-session by a prompt injection attack embedded in retrieved content. The receipt was signed before the session began; it has no knowledge of runtime inputs.

SessionState closes these gaps by maintaining a live, stateful view of each session that evolves with every action. It tracks a trustScore for each session, initialized at 100 and bounded between 0 and 100.

Three formally distinct quantities govern session risk evaluation. Implementations *MUST* maintain all three:

trustScore:
A Lyapunov-style bounded recoverable budget. Initialized at 100, bounded in [0, 100]. Decremented by anomaly.severity * trustDecayRate on each anomaly event; incremented by trustRecoveryRate on each clean action. The recovery property is definitional: trustScore is not monotone and is not a load functional. It models a resilience budget that is restored by sustained clean behavior.

cumulativeAnomalyMass:
A monotone, non-decreasing quantity tracking the total structural burden accumulated over the session lifetime. cumulativeAnomalyMass has two components:

Active (anomaly-driven):
Incremented by anomaly.severity on each detected anomaly event. Records the discrete burden contributed by individual anomaly detections.

Passive (time-driven):
Incremented by passivePressureRate * elapsedSeconds on each call to the session risk evaluator, where elapsedSeconds is the time elapsed since the previous evaluation. The default passivePressureRate is 0.001 per second, yielding 3.6 units of passive burden per session- hour and 36 units per ten session-hours, even with zero detected anomalies.

cumulativeAnomalyMass is never decremented. It provides a permanent session-level record of total anomaly exposure that is not erased by subsequent clean behavior and is available for post-session forensic analysis independently of the final trustScore value.

tauSession:
A strictly decreasing capacity gate derived from cumulativeAnomalyMass:

```
tauSession = sessionCapacity - cumulativeAnomalyMass
```

Initialized to sessionCapacity (default: 100). Never recovered. When tauSession <= tauMin (default: 10), the gate condition fails and execution *MUST NOT* proceed regardless of trustScore. The gate condition is checked before all trustScore-derived checks in dynamic_admissible:

```
if tauSession <= tauMin: DENY TAU_SESSION_EXHAUSTED
```

tauSession provides a hard lifetime cap on cumulative anomaly exposure that is not resettable by reauthorization. A new session established under a new Delegation Receipt begins with tauSession initialized to sessionCapacity (default: 100). Implementations *MUST NOT* carry tauSession state across session boundaries; each new receipt-bounded session receives a fresh tauSession value. Once a session's anomaly capacity is exhausted, the session is permanently closed to further execution.

A new session established under a new Delegation Receipt begins with tauSession initialized to sessionCapacity (default: 100). Implementations *MUST NOT* carry tauSession state across session boundaries; each new receipt-bounded session receives a fresh tauSession value.

In addition to the anomaly-capacity gate, sessions *MUST* enforce an absolute wall-clock lifetime cap. The maximum session lifetime is 25 hours from session start. When elapsed >= 25h, the session *MUST* be terminated and reauthorization *MUST* be required regardless of trustScore or tauSession. This bound prevents indefinitely-running sessions that accumulate unbounded passive anomaly pressure from gradually transitioning to SUSPENDED while remaining nominally valid.

Implementations *MUST* enforce this limit via the maxLifetimeSeconds configuration parameter. The default value is 90000 seconds (25 hours). Implementations *MUST* document the configured value and *MUST NOT* set maxLifetimeSeconds to zero or a negative value. When elapsed >= maxLifetimeSeconds, the verifier *MUST* return SESSION_LIFETIME_EXCEEDED and *MUST NOT* permit further execution under the current session.

The three configurable risk quantities have the following defaults. Implementations *MUST* document the values in use and *SHOULD* expose them as configuration parameters:

```
blockThreshold     : 70   (implementation-defined; MUST be > approvalThreshold)
approvalThreshold  : 40   (implementation-defined; MUST be > 0)
trustDecayRate     : 1.0  (implementation-defined; MUST be > 0)
trustRecoveryRate  : 0.01 (implementation-defined; MUST be >= 0)
```

The following table provides the single authoritative reference for all session state threshold values. All threshold comparisons in the pseudocode and prose of this section use these values:

Parameter	Default Value	Condition / Meaning
blockThreshold	70	Risk score at or above this value blocks the action
approvalThreshold	40	Risk score at or above this value requires explicit approval
ACTIVE status lower bound	trustScore >= 30	Session is in normal operation
DEGRADED status range	10 <= trustScore < 30	Risk multiplier applied; heightened scrutiny
SUSPENDED status threshold	trustScore < 10	All actions blocked regardless of risk score
tauMin	10	tauSession at or below this value blocks all actions
sessionCapacity	100	Initial value of tauSession
trustDecayRate	1.0	Multiplier applied to anomaly severity when decrementing trustScore
trustRecoveryRate	0.01	Amount added to trustScore per clean action

Table 2: Session State Threshold Values

Trust decays when anomalies are detected:

```
trustScore -= anomaly.severity * trustDecayRate
```

Anomaly severity weights are defined on a [0, 1] float scale (see Table below). Representative default values are:

```
Prompt injection detected        : severity weight 0.8
Sensitive data in external dest. : severity weight 0.6
Scope boundary probe             : severity weight 0.4
Timing anomaly                   : severity weight 0.2
```

The following table provides example anomaly severity weights for common event types. These values are illustrative defaults; implementations *MAY* tune them to operational risk tolerance:

Anomaly Type	Severity Weight
Prompt injection detected	0.8
Repeated denial pattern	0.6
Scope boundary probe	0.4
Timing anomaly	0.2

Table 3

Severity weights use a [0.0, 1.0] scale. Implementations *MAY* tune these weights to operational risk tolerance. All weights *MUST* be in the range [0.0, 1.0]; a weight of 1.0 represents the maximum single-event anomaly severity.

These weights are illustrative defaults on a [0, 1] scale. Implementations MAY tune severity weights to operational risk tolerance. All severity weights MUST be in the range [0, 1]. The pseudocode in this section uses these weights directly; a weight of 1.0 represents the maximum possible anomaly severity for a single event.

Trust recovers slightly on each clean action:

```
trustScore += trustRecoveryRate  (default: 0.01)
```

Session status is driven by trust score thresholds:

```
trustScore >= 30 : ACTIVE    -- normal operation
trustScore <  30 : DEGRADED  -- risk scores amplified
trustScore <  10 : SUSPENDED -- all actions blocked
```

The DEGRADED state does not block operations directly. Instead, it causes the risk scorer to apply a multiplier to every score via Check 5 (finalScore = rawScore * (1 + (100 - trust) / 100)), making previously marginal decisions tip into REQUIRE_APPROVAL or BLOCK territory. This multiplier is applied independently of sensitivity-level threshold adjustments: sensitivity adjustments modify the _thresholds_ against which finalScore is compared, while the DEGRADED multiplier modifies finalScore itself. Both transformations are applied and the result compared against the adjusted thresholds.

The following pseudocode defines the evaluateSessionRisk function referenced in Check 9 of the verification algorithm (Section 6.4):

```
   FUNCTION evaluateSessionRisk(session, event):

     // Gate 1: tauSession hard cap
     IF session.tauSession <= session.tauMin THEN
       RETURN { decision: "BLOCK", reason: "TAU_SESSION_EXHAUSTED" }

     // Update cumulative anomaly mass (always include passive pressure)
     elapsed = now() - session.lastEvaluatedAt
     session.cumulativeAnomalyMass +=
         session.passivePressureRate * elapsed
     session.lastEvaluatedAt = now()

     // Update trust score and discrete anomaly mass
     IF event.type == "ANOMALY" THEN
       session.trustScore -= event.severity * trustDecayRate
       session.cumulativeAnomalyMass += event.severity
     ELSE  // clean action
       session.trustScore = MIN(100,
         session.trustScore + trustRecoveryRate)

     // Recompute tauSession from total cumulative mass
     session.tauSession = session.sessionCapacity
                          - session.cumulativeAnomalyMass

     // Apply DEGRADED multiplier if trust is low
     rawScore = computeRiskScore(event, session)
     IF session.trustScore < 30 THEN
       finalScore = rawScore * (1 + (100 - session.trustScore) / 100)
     ELSE
       finalScore = rawScore

     // Gate 2: trust thresholds
     IF session.trustScore < 10 OR session.status == "SUSPENDED" THEN
       RETURN { decision: "BLOCK",
                reason: "SESSION_RISK_THRESHOLD_EXCEEDED" }
     IF finalScore >= blockThreshold THEN
       RETURN { decision: "BLOCK",
                reason: "SESSION_RISK_THRESHOLD_EXCEEDED" }
     IF finalScore >= approvalThreshold THEN
       RETURN { decision: "REQUIRE_APPROVAL",
                reasons: getRiskReasons(finalScore) }

     RETURN { decision: "ALLOW" }

   END FUNCTION
```

Before each action, every payload is classified into one of four sensitivity levels:

```
RESTRICTED   : SSN, credit card, medical identifiers, API keys
CONFIDENTIAL : Internal email addresses, system prompts,
               database schemas, config files
INTERNAL     : Company domain references, internal project
               names, user IDs
PUBLIC       : Everything else
```

Each level modifies the block and approval thresholds:

```
RESTRICTED   : Block threshold drops to at most 60
CONFIDENTIAL : Approval threshold drops to at most 40
INTERNAL     : No change
PUBLIC       : All thresholds relax by +10
```

The complete decision engine evaluates five risk checks and maps the final score to one of three outcomes:

```
Check 1 -- Sensitive data scan:
  SSN pattern              +35
  Credit card pattern      +35
  API key pattern          +30
  High-entropy string      +20
  Prompt injection pattern +40
  Password keyword         +25

Check 2 -- External exfiltration:
  External domain + sensitive data  +30
  First-time external domain        +15

Check 3 -- Frequency anomaly:
  Same action type >10x in 60s  +25
  >50 total actions in session  +15

Check 4 -- Scope edge usage:
  New permission class   +10
  At scope boundary      +10

Check 5 -- Trust multiplier:
  finalScore = rawScore * (1 + (100 - trust) / 100)

if session.status == SUSPENDED       --> BLOCK
if finalScore >= blockThreshold      --> BLOCK
if finalScore >= approvalThreshold   --> REQUIRE_APPROVAL
else                                 --> ALLOW
```

The checks are deterministic and ordered. The same action, payload, and session state always produce the same score. Every BLOCK or REQUIRE_APPROVAL decision *SHOULD* be accompanied by a structured reason object identifying which specific checks contributed to the score.

SessionState *MUST* be integrated with the PreExecutionVerifier as a final check, running after all static receipt checks pass (see Section 6.1). An action that passes all cryptographic checks but produces a BLOCK outcome from session risk evaluation *MUST NOT* execute.

The architectural insight is that authorization is not binary. A valid receipt is a necessary condition for execution, not a sufficient one. Real-world safety requires a live, stateful layer that observes behavior and adapts its decisions based on the full session context.

The complete executability predicate is formally:

```
executable(a, R, session, t) =
  Verify(R, a) AND dynamic_admissible(session, a, t)
```

where Verify(R, a) establishes static receipt admissibility (the complete set of pre(R) and admissible(a, R) checks defined in Section 11) and dynamic_admissible(session, a, t) establishes runtime session admissibility. dynamic_admissible evaluates checks in the following order: (1) tauSession gate -- if session.tauSession <= tauMin, DENY TAU_SESSION_EXHAUSTED immediately; (2) trust score threshold check; (3) sensitivity classification; (4) risk score evaluation at time t. Both the tauSession gate and the trustScore threshold must pass independently. Execution requires both predicates to hold simultaneously. A receipt that passes Verify is a necessary but not sufficient condition for execution.

10. Multi-Agent Delegation Chains

When a delegated Agent (the Orchestrator) needs to hand off a subtask to another Agent (the Sub-Agent), the chain of authority *MUST* remain auditable and bounded. DRP enforces the narrowing-only invariant: a Sub-Agent receipt *MUST* have scope that is a strict subset of the Orchestrator’s receipt. The Orchestrator *MUST NOT* grant a Sub-Agent any action that was not in its own scope. Scope can only narrow, never widen.

10.1. Parent-Child Receipt Relationship

A Sub-Agent receipt is distinguished from a root receipt by two additional fields in its payload:

parentReceiptId:
: The SHA-256 hash of the Orchestrator's Delegation Receipt that authorized the Orchestrator to create this sub-delegation. When this field is present the receipt is treated as a sub-receipt and subjected to the parent scope containment check (Check 14). When absent the receipt is treated as a root receipt and requires a User signature.

orchestratorSignature:
: An ECDSA P-256 signature by the Orchestrator's key over the binding string "orchestrator-delegation:" || parentReceiptId || ":" || receiptId, where receiptId is the SHA-256 hash of the sub-receipt body plus its main signature. This field is separate from the main signature field and is not included in the signed body. It cryptographically links the Orchestrator's key to the specific parent-child pair. The User's signature is not required for sub-receipts because the User already authorized the Orchestrator to act and the Orchestrator is creating a narrower delegation.

10.2. Scope Attenuation (Narrowing-Only Rule)

Each delegation step *MUST* produce a strict proper subset of the parent's authorized scope. The subset relation is defined under wildcard semantics: an action pattern P1 covers action pattern P2 if every concrete action matched by P2 is also matched by P1. Specifically:

1. Every action the child Agent is permitted *MUST* be covered by the parent's allowedActions list under wildcard semantics. An Agent *MUST NOT* grant permissions it was not itself given. Formally: for every child allowed action C, there *MUST* exist a parent allowed action P such that every concrete action matched by C is also matched by P.

2. The child's allowedActions list *MUST* be a strict proper subset of the parent's under wildcard semantics: the set of concrete actions covered by the child *MUST* be a proper subset of the set covered by the parent. A child that covers exactly the same set of concrete actions as the parent *MUST* be rejected, even if the two lists are expressed differently (e.g., via differing wildcard patterns that expand to the same set). Implementations *MUST* expand wildcard patterns to their concrete action sets before performing the strict proper subset comparison; a child scope

expressed as "read:*" and a parent scope expressed as an explicit enumeration that resolves to the same concrete actions are equal under this rule and *MUST* be rejected.

3. Every action explicitly denied by the parent in deniedActions *MUST* be carried forward to the child. A child *MAY* add new denied actions but *MUST NOT* remove any denial that the parent established. A child denial pattern covers a parent denial when the child's operation pattern matches the parent's operation and the child's resource pattern matches the parent's resource under wildcard semantics.

The receipt chain *MUST* track delegation depth. The root receipt, signed directly by the User, is at depth 0. Each delegation increments depth by 1. When a delegation would produce a receipt at depth >= maxDepth, the implementation *MUST* raise a MaxDepthExceeded error before creating the receipt. The *RECOMMENDED* default maxDepth is 3, meaning at most three levels of agent-to-agent hand-off before the chain *MUST* be re-anchored at the User level. This bound prevents unbounded delegation trees where authority leaks through an unconstrained number of intermediaries.

Implementations that require delegation chains deeper than 3 *SHOULD* do so only in deployment architectures where each orchestrator layer is independently audited. Chains deeper than 3 increase verification cost linearly: each additional hop requires one additional parent receipt retrieval, signature verification, and scope containment check. They also expand the attack surface for PARENT_SCOPE_VIOLATION exploits, since a compromised intermediary at any depth can attempt to widen scope or drop a denial that a shallower intermediary established. Before increasing maxDepth, implementors *SHOULD* evaluate whether the additional orchestrator layers provide genuine isolation or merely add verification overhead without a corresponding security benefit.

The root receipt *MUST* carry a valid ECDSA P-256 signature from the User's key. If the root could be generated by an Agent or Operator without User involvement, the entire chain could be bootstrapped unilaterally, defeating the protocol. Any downstream Agent that wants to prove its authority can walk the chain to the root and demonstrate a continuous path of scope-narrowing receipts.

10.3. Verification Chain for Sub-Receipts

When a verifier encounters a receipt whose parentReceiptId field is present, it *MUST* perform Check 14 in addition to Checks 1-13. Check 14 is defined as follows:

1. Retrieve the parent receipt from the delegation log using parentReceiptId. If not found, return PARENT_SCOPE_VIOLATION.

2. Count the chain depth by following parentReceiptId links from the current receipt to the root. If the depth exceeds maxChainDepth, return PARENT_SCOPE_VIOLATION.

3. Verify the parent receipt's main ECDSA signature. If invalid, return PARENT_SCOPE_VIOLATION.

4. If orchestratorSignature is present, verify it against the binding string "orchestrator-delegation:" || parentReceiptId || ":" || receiptId using the *parent receipt's* publicKey. If invalid, return PARENT_SCOPE_VIOLATION.

5. Confirm that the sub-receipt's time window is contained within the parent receipt's time window (i.e., child.start >= parent.start and child.end <= parent.end). If not, return PARENT_SCOPE_VIOLATION.

6. Confirm that every action in the sub-receipt's allowedActions is also permitted by the parent receipt's scope. If any child allowed action is not covered by the parent, return PARENT_SCOPE_VIOLATION.

7. Confirm that every action in the parent receipt's deniedActions is also present in the sub-receipt's deniedActions. A child deny rule covers a parent deny rule when the child's operation pattern matches the parent's operation and the child's resource pattern matches the parent's resource. If any parent denied action is missing from the child, return PARENT_SCOPE_VIOLATION.

The verifier MUST NOT re-run the full verification algorithm on ancestor receipts during Check 14 (i.e., it does not recurse into parentReceiptId chains beyond the immediate parent). Depth-counting, by contrast, MUST traverse the chain iteratively to enforce maxChainDepth, reading only the receiptId and parentReceiptId fields of each ancestor. Each receipt in the chain is fully verified only when it is itself presented for execution.

10.4. Cascade Revocation

Revocation of a receipt in a multi-agent chain *MAY* or may not cascade to child receipts, depending on the revocation call.

When cascadeToChildren is true:
A breadth-first traversal of all descendants *MUST* be performed and each descendant marked revoked. The cascade *SHOULD* be

anchored to the append-only log before agents are notified, to prevent a race condition where a child Agent completes an action between the parent revocation and cascade propagation.

When cascadeToChildren is false:
Only the named receipt is invalidated. Its children remain valid until explicitly revoked. This allows surgical removal of one Agent from a chain without disrupting sibling branches.

Cascade revocation entries *MUST* be signed by the User's private key and anchored to the append-only log with the same requirements as the original revocation procedure (see Section 11.1).

10.5. Revocation Log

Revocation entries *MUST* be published to the same append-only log as Delegation Receipts. A revocation record is a first-class log entry with the same chain-linking and TSA timestamping requirements as a Delegation Receipt. Publishing revocation to the same log ensures that any party with read access to the delegation log can independently verify revocation status without relying on a separate revocation infrastructure.

Implementations *MAY* additionally operate an operator-managed Certificate Revocation List (CRL) for low-latency revocation propagation in environments where log polling latency is unacceptable. When both mechanisms are in use, the append-only log entry is authoritative: the CRL *MUST* be a subset of the log's revocation state and *MUST NOT* revoke receipts not present in the log. Log-based revocation is *RECOMMENDED* for deployments requiring cryptographic non-repudiation of the revocation act itself. CRL-based revocation is appropriate as a supplementary fast-path when polling latency exceeds operational requirements.

Implementations that poll the revocation log *SHOULD* use the following polling intervals:

* *RECOMMENDED* polling interval for real-time deployments: 60 seconds.

* *RECOMMENDED* polling interval for batch deployments: 300 seconds.

Implementations *MUST* document the configured polling interval and *MUST NOT* set it to zero or a negative value.

10.6. Revocation Authority and Propagation

The following principals are authorized to revoke a Delegation Receipt:

* The User who signed the receipt (the key holder).
* The Operator, for receipts issued under that Operator's instruction set.
* A platform-level revocation authority designated in the Operator's trust anchor configuration, if present.

Revocation is propagated by publishing a signed revocation record to the append-only log. A revocation record *MUST* contain: the receiptId being revoked, the revocation timestamp, the revoking principal's public key, and a signature over those fields. Verifiers *MUST* treat any receipt whose receiptId appears in the revocation log as revoked regardless of its time window.

Implementations *SHOULD* propagate revocation records to all active verifiers within 60 seconds of publication to the log. This is the *RECOMMENDED* propagation timeout. Verifiers that have not received a revocation record within 300 seconds of the log publication timestamp *SHOULD* treat the absence as a potential network partition and enter a degraded verification posture (i.e., requiring re-validation of any receipt before accepting the next action).

When a revocation record arrives while a session is mid-execution (i.e., an action authorized by the revoked receipt is currently in progress), the verifier *MUST* complete the in-flight action if and only if it is idempotent and non-destructive (e.g., READ operations), then *MUST* block all subsequent actions and terminate the session with status REVOKED. Non-idempotent or destructive in-flight actions *MUST* be aborted immediately upon receipt of the revocation record.

11. Security Considerations

11.1. Threat Model

DRP considers the following adversaries and mitigations:

Compromised Operator:
An attacker who gains control of the Operator's systems can alter the instructions delivered to the Agent. Under DRP, any instruction diverging from the hash committed in the Delegation Receipt is immediately detectable at step (7) of Verify. The attacker cannot issue instructions that pass hash verification without the User's private key.

Malicious Operator:
An Operator who intentionally instructs the Agent to exceed the User's authorization -- under commercial pressure, legal compulsion, or bad faith -- produces a detectable instruction hash mismatch. The discrepancy between committed and actual instructions is an auditable fact in the append-only log. The Operator cannot alter the log entry and cannot alter the signed receipt.

Log Integrity:
The security of the protocol depends on the tamper-evidence of the append-only log. Implementations *SHOULD* use decentralized log implementations following the Certificate Transparency model that do not depend on a single operator for integrity. Log fork detection follows established approaches from CT ecosystems [RFC6962].

Key Compromise:
If the User's signing key is compromised, an attacker can issue Delegation Receipts in the User's name. Hardware key custody using a FIDO2 authenticator [FIDO2] significantly reduces this risk by making key extraction technically infeasible on modern devices with secure enclaves.

Revocation:
When a User wishes to revoke a Delegation Receipt, they *MUST*:

1. Construct a revocation record containing: the SHA-256 hash of the original receipt, a reason string, and a revocation timestamp.

2. Sign the revocation record with the same private key used to sign the original receipt.

3. Publish the signed revocation record to the append-only log, producing an immutable log anchor.

The log anchor establishes the authoritative revocation time. Actions taken before this timestamp under the original receipt remain valid. Actions attempted after this timestamp *MUST* fail.

Verification *MUST* check revocation before any other check (Section 6.1, step 1). Because the revocation record is itself signed by the User and anchored to the log, it carries the same evidentiary weight as the original receipt. Revocation is auditable, tamper-evident, and does not depend on the Operator to propagate or acknowledge it.

Replay Attack:
A Delegation Receipt is a static signed artifact. An adversary who intercepts or stores a receipt could re-present it in a later session or in a concurrent context to authorize actions the User did not intend for that context. DRP mitigates replay through Check 10 of the verification algorithm: the verifier records each presented receiptId in a per-session presentation log and rejects any receipt that appears more than once within the same session, returning REPLAY_DETECTED. Implementations *SHOULD* include a short-lived nonce in every action submission and maintain a per-session replay cache with entries expiring at timeWindow.notAfter. The append-only log provides a secondary audit trail: a re-presented receipt ID appearing in the log for two distinct sessions is detectable post-hoc without reliance on the per-session cache.

Model Substitution:
An Operator could silently substitute a fine-tuned model variant after receipt issuance. Model State Attestation (Section 7) closes this gap by binding a cryptographic measurement of the model state to the receipt at delegation time and re-verifying inside a TEE at execution time.

Prompt Injection:
A signed Delegation Receipt enforces what actions are permitted but does not by itself verify that the instruction to take a permitted action originated from a trusted source. A prompt injection attack may manipulate an agent into requesting an in-scope action for malicious purposes -- for example, a receipt permits send_email but a poisoned retrieved document instructs the agent to send to an attacker-controlled address. DRP addresses this vector through instruction provenance checking (step 13 of Verify): the receipt *MAY* include a trustedSources field listing the instruction sources -- such as user, system_prompt, or verified_tool -- that are permitted to influence agent behavior. When present, the verifier *MUST* reject any action whose instructionSource is not in the list, returning UNTRUSTED_INSTRUCTION_SOURCE. This gives the User cryptographic control over which input channels can drive agent behavior, independent of the content of those inputs.

Operator Instruction Confidentiality:
The operatorInstructions field carries the Operator's plaintext system prompt in the receipt, which is published to the append-only log. Operators with sensitive system prompts -- such as proprietary workflow logic or confidential configuration -- should consider the confidentiality implications of log publication. The hash binding in operatorInstructionsHash provides full integrity assurance without requiring plaintext disclosure: an Operator *MAY* omit the operatorInstructions field from the log entry and store the plaintext off-log, provided the SHA-256 hash committed in operatorInstructionsHash can be independently verified by the User on demand. The integrity guarantee is preserved by the hash alone; the plaintext is needed only for human-readable auditing. The operatorInstructionsHash field thus serves as the cryptographic binding for the off-log commitment option described in Section 4.1.

Micro-Receipt Fatigue:
A malicious Operator could structure a workflow to generate many micro-receipt requests in rapid succession, inducing the User to approve actions they do not meaningfully review. This is analogous to notification fatigue attacks against MFA prompts. The protocol makes every approval a signed, auditable artifact. Rate-limiting and UI affordances are the primary mitigation; protocol implementations *SHOULD* enforce a minimum inter-request interval for micro-receipt prompts.

Chip Away Attack:
An adversary in control of the agent alternates anomalous actions with clean actions in an attempt to reset the session trust score between each anomalous step. Because trustScore is a recoverable Lyapunov budget, a sequence of the form clean-anomalous-clean-anomalous could in principle maintain trustScore above the block threshold indefinitely while avoiding a single high-severity spike. tauSession closes this attack surface: it is derived from the monotone cumulativeAnomalyMass and is never decremented by clean actions. Each anomalous action permanently reduces tauSession regardless of subsequent clean behavior. When tauSession falls to or below tauMin, execution is permanently blocked for that session and reauthorization cannot restore it. An adversary executing a chip-away pattern merely exhausts the session anomaly capacity faster. A new session established under a new Delegation Receipt begins with tauSession initialized to sessionCapacity (default: 100). Implementations *MUST NOT* carry tauSession state across session boundaries; each new receipt-bounded session receives a fresh tauSession value.

Authority Continuity:
An identity authority upstream of the receipt signer may degrade, rotate, or expire while a long-lived Delegation Receipt remains cryptographically valid. For example, an enterprise SSO provider may rotate signing keys or suspend an account after the receipt was issued; the receipt itself remains valid under the User's local key, which may no longer accurately reflect the principal's authorization status. Implementations *SHOULD* use short-lived receipts for delegations tied to upstream identity infrastructure and *SHOULD* integrate SCIM provisioning events or short-lived OAuth 2.0 access tokens to detect identity state changes that would invalidate an otherwise cryptographically sound receipt.

Non-Repudiation:
Let R be a Delegation Receipt with content C, user signature sigma, and log anchor L. Under the ECDSA P-256 EUF-CMA unforgeability assumption, no party without the User's private key can produce a valid sigma for any C. Therefore, the existence of a valid receipt on the log is non-repudiable evidence that the holder of the private key authorized the content of C at time L.

Authorization Persistence:
Authorization at time t requires both (a) a valid signed receipt anchored at time L and (b) the absence of any valid revocation record for that receipt anchored at time L' where L' < t. The validity of sigma (established under Non-Repudiation) is a necessary but not sufficient condition for continued authorization: it proves the receipt was genuinely issued but does not establish that it remained unrevoked through time t. Non-repudiation and authorization persistence are formally distinct results. The protocol proves both: Non-Repudiation via the EUF-CMA unforgeability property of ECDSA P-256, and Authorization Persistence via the tamper-evidence property of the append-only log applied to both receipt anchors and revocation record anchors.

Soundness:
Verification decomposes into two independent predicates:

```
Verify(R, a) = pre(R) AND admissible(a, R)
```

pre(R) holds if and only if: (i) R has not been revoked (revocation check); (ii) the signature sigma over the canonical body of R is valid under the User's public key (signature verification); and (iii) the log timestamp falls within R.timeWindow (time window validity).

admissible(a, R) holds if and only if: (iv) a appears in the scope allowlist in C (scope check); (v) a does not violate any prohibition in C (boundary check); (vi) if a is an execution action, Hash(ExecutionGraph(program)) equals the hash committed in C (execution hash check); (vii) the SHA-256 hash of the Operator's current instructions equals R.operatorInstructionsHash (instruction hash check); (viii) if R.modelCommitment is present, the current model state measurement equals R.modelCommitment (model state attestation check); (ix) if sessionState is present, the session risk evaluation passes (session risk evaluation); (x) the receipt has not been presented previously in this session (replay detection); (xi) if R.toolSchemaHash is present, the SHA-256 hash of the current tool schema equals R.toolSchemaHash (tool schema hash check); (xii) if R.toolOutputHash is present and action.toolOutput is present, the SHA-256 hash of action.toolOutput equals R.toolOutputHash (tool output hash check); (xiii) if R.trustedSources is present and action.instructionSource is present, action.instructionSource appears in R.trustedSources (instruction provenance check); and (xiv) if R.parentReceiptId is present, the sub-receipt passes the parent scope containment verification (parent scope containment).

For any action a, Verify(R, a) = true if and only if all of (i)-(xiv) hold simultaneously. Any deviation in any component causes Verify to return false. The Operator cannot alter C without invalidating sigma; the Operator cannot alter L by the tamper-evidence property of the append-only log. The executable() predicate in Section 9 further establishes that Verify(R, a) = true is a necessary but not sufficient condition for execution.

dynamic_admissible now requires both (a) trustScore above the block threshold and (b) tauSession above tauMin. Either condition failing independently is sufficient to produce a DENY outcome. The tauSession gate is evaluated first and is not resettable by reauthorization: once a session's anomaly capacity is exhausted, no subsequent clean actions or reauthorization calls can restore admissibility for that session.

11.2. Semantic Gap

The protocol does not eliminate the semantic gap between authorized scope and authorized intent. A User who authorizes "write to calendar" may not intend to authorize deletion of all existing events. The Scope Discovery Protocol (Section 8) narrows this gap by grounding scope definitions in observed agent behavior rather than user speculation.

The "executes" scope class narrows the gap further for code execution by requiring the SHA-256 hash of the program's static capability DAG rather than a program name or URI. A program identified by name or URI can be silently replaced; a program identified by its capability signature *MUST* have the same capability set as the authorized version. Any capability addition or removal changes the signature.

Natural language *MUST NOT* appear in any scope field. Scope entries *MUST* be structured resource:operation pairs. This restriction exists because natural language scope definitions are ambiguous, subject to interpretation, and cannot be used for deterministic validation. A scope entry of "manage email" does not deterministically resolve to a set of permitted or denied operations.

Cryptographic primitive upgrade path: DRP uses SHA-256 throughout for receipt ID computation, instruction hash commitment, manifest body hashing, action log chain linking, and revocation record linking. The version field in the receipt structure provides a migration path to SHA-3-256 (FIPS 202) or BLAKE3 in a future protocol version. Both are drop-in replacements for the SHA-256 role in this protocol. No structural redesign is required for a hash function migration.

Quantum resistance: ECDSA P-256 is vulnerable to Shor's algorithm on a sufficiently capable quantum computer. Ed25519 ([RFC8032]) offers an alternative classical signing algorithm with smaller key and signature sizes and is *RECOMMENDED* for new deployments as a drop-in replacement for ECDSA P-256; it shares the same quantum vulnerability under Shor's algorithm and is therefore not post-quantum secure either. The long-term migration path for both algorithms is through the FIDO2/WebAuthn credential layer: because all DRP signing is abstracted behind the WebAuthn API, a platform-level upgrade to post-quantum FIDO2 authenticators (e.g., CRYSTALS-Dilithium, FALCON) upgrades the protocol's quantum resistance without protocol-layer changes. The append-only log and hash commitment structures are unaffected -- SHA-256 preimage resistance is not threatened by known quantum algorithms.

For broader AI-specific risk management considerations beyond the cryptographic scope of this protocol, see the NIST AI Risk Management Framework [NIST-AI-100-1].

11.3. TEE Enforcement

Cryptographic receipt verification alone cannot prevent a compromised agent runtime from calling the verifier with a spoofed receipt, receiving an "allowed" result, and then ignoring the scope. A three-layer enforcement model addresses this:

Layer 1 -- Receipt:
: Cryptographic proof of User authorization (what). The User signs a Delegation Receipt specifying scope, boundaries, and Operator instructions. Any tampering is immediately detectable.

Layer 2 -- TEE:
: Hardware-measured execution environment (where and how). The Agent runs inside an attested enclave whose measurement is bound to the receipt via the teeMeasurement.expectedMrenclave field. Any substitution of model weights, verifier code, or platform produces a different measurement and is detectable before execution.

Layer 3 -- eBPF:
: Kernel-level enforcement. An eBPF LSM hook validates a signed capability token on every relevant syscall (security_file_open, security_socket_connect, security_task_execve). Scope violations *MUST* be denied at the kernel level before they reach userspace.

This document defines two compliance profiles for TEE enforcement:

Standard Profile:
: The modelCommitment field is OPTIONAL. Implementations that omit it skip Check 8 (Model State Attestation) entirely. All other checks remain normative. This profile is suitable for deployments where TEE hardware is unavailable or where the operator accepts the associated risk.

Full-Compliance Profile:
: The modelCommitment field is REQUIRED. Implementations MUST populate it and verifiers MUST perform Check 8. Hardware attestation via a Trusted Execution Environment (TEE) is REQUIRED under this profile. Operators deploying under the Full-Compliance Profile MUST document their TEE attestation mechanism in their trust anchor configuration.

The compliance profile in use MUST be declared in the Operator's trust anchor configuration. Verifiers MUST reject receipts that omit modelCommitment when the governing trust anchor specifies the Full-Compliance Profile.

Without Layer 3, a compromised agent runtime could bypass the verifier. The eBPF LSM runs in kernel space and cannot be disabled by userspace code, including a compromised agent runtime. All three layers *MUST* be present for the enforcement model to be complete and non-bypassable.

Intel TDX and AMD SEV-SNP provide encrypted memory pages inaccessible to the host OS and hypervisor, hardware-rooted attestation quotes signed by the CPU vendor's key, and measured boot that hashes every component loaded into the enclave. DRP binds delegation receipts to enclave measurements via:

```
mrenclave = SHA-256(
  platform || verifierHash || modelHash
)
```

This value is committed into the receipt's teeMeasurement.expectedMrenclave field at delegation time. At execution time, the runtime recomputes mrenclave from its runtime parameters and *MUST* reject execution if there is any mismatch.

The token injection sequence is:

1. ConfidentialRuntime.launch() computes mrenclave and verifies the receipt measurement.

2. PreExecutionVerifier.check() gates execution -- no valid receipt means no execution.

3. TokenPreparer.prepare() builds a signed capability token binding receipt hash, scope hash, and TEE quote hash.

4. The token is injected into the agent process context.

5. The eBPF LSM validates the token on every relevant syscall.

6. Any operation not covered by the token's scope *MUST* be denied at the kernel level.

Note on optional field vs. required enforcement: The teeMeasurement field in the Delegation Receipt structure is *OPTIONAL* -- implementations are not required to include it in every receipt. However, TEE-based enforcement is *REQUIRED* for any deployment claiming full DRP compliance. An implementation that issues receipts without teeMeasurement while not operating within a TEE-enforced runtime does not satisfy the full DRP compliance profile. Deployments *MUST* document whether they operate in TEE-enforced mode and *MUST NOT* claim full DRP compliance without active TEE enforcement.

11.4. Degraded Operation

When the verifier cannot construct the required authorization state due to log unavailability, unverifiable revocation status, or an UNVERIFIED_TIMESTAMP condition (see Section 5.3), execution *MUST NOT* proceed regardless of operator acknowledgment. Operator acknowledgment does not reconstruct a structurally missing verification input and therefore cannot substitute for a complete, verifiable authorization state. Implementations *MUST* fail closed under these conditions in all deployment contexts. An UNVERIFIED_TIMESTAMP is treated equivalently to an unverifiable revocation status: both represent missing verification inputs that cannot be reconstructed by operator assertion.

When operating against a locally cached revocation registry, implementations *MUST* note the cache timestamp and *MUST* reject any receipt where the revocation status cannot be verified against data anchored within the configurable maximum cache age. An unverifiable revocation status is treated equivalently to a verified revocation: execution *MUST NOT* proceed.

Implementations *MUST* provide configuration to specify the maximum acceptable cache age for revocation data. The default maximum cache age is one hour.

When the Time-Stamp Authority (TSA) is unreachable, TEE attestation requirements are not relaxed. Implementations operating under the TEE enforcement profile (Section 11.3) *MUST* continue to require valid TEE attestation for all actions even in degraded mode; the absence of a TSA timestamp does not constitute grounds for bypassing hardware attestation. If both the TSA and the TEE attestation service are unreachable, the implementation *MUST* block all EXECUTE-class actions and *MAY* permit READ-only actions subject to standard scope verification.

11.5. Key Management

Signing keys used to issue Delegation Receipts carry significant authority and *MUST* be managed with care:

Key Rotation:
Implementations *SHOULD* rotate signing keys at least every 90 days or upon any personnel change that affects access to the signing credential. All Delegation Receipts issued under the rotated key remain valid for their stated time window. New receipts issued after rotation *MUST* use the new key. Implementations *SHOULD* publish key rotation events to the append-only log to provide an auditable history of which key was active at any point in time.

Lost or Stolen Key Procedures:
Upon discovery of a lost or stolen signing key, the key holder *MUST* immediately revoke all active receipts issued under the compromised key and re-issue new receipts under a freshly generated key. Revocation records *MUST* be anchored to the append-only log before the new key is placed into service. Key compromise events *SHOULD* be communicated to all Operators holding receipts signed under the compromised key.

12. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs.

authproof-cloud
Organization: Authproof

Maturity level: prototype (alpha)

Coverage: 1,251 tests across 20 test files. The following features are implemented and tested:

* Receipt issuance and Ed25519/ECDSA P-256 signing

* Append-only log anchoring with TSA timestamping

* Pre-execution verification: all 14 checks (Section 6.4), including sub-receipt chain validation (Check 14)

* Session state lifecycle management and evaluateSessionRisk evaluation

* Scope Discovery Protocol (observation mode and sandboxed execution)

* Revocation log publication and polling

The following features are partially implemented or planned:

* Full TEE attestation integration (Check 8) -- in progress; currently validated via software attestation mock. The current implementation satisfies the Standard Profile defined in Section 11.3; it does *NOT* yet satisfy the Full-Compliance Profile, which requires hardware TEE attestation.

* Multi-operator trust anchor federation -- planned

Licensing: MIT License

Contact and repository: https://github.com/Commonguy25/authproof-sdk

13. IANA Considerations

This document requests IANA to create the following registries under a new "Delegation Receipt Protocol (DRP)" registry group.

13.1. DRP Denial Reason Codes Registry

IANA is requested to create a registry titled "DRP Denial Reason Codes". The policy for new registrations is Specification Required. Initial values are:

Reason Code	Description
REVOKED	Receipt has been explicitly revoked
SIGNATURE_INVALID	Receipt signature verification failed
TIME_WINDOW_EXPIRED	Receipt time window has elapsed
SCOPE_VIOLATION	Requested action not in allow list
ACTION_EXPLICITLY_DENIED	Requested action in deny list or boundary
EXECUTION_HASH_MISMATCH	Program execution graph hash does not match receipt commitment

INSTRUCTION_HASH_MISMATCH	Operator instructions hash does not match receipt commitment
REPLAY_DETECTED	Receipt presented more than once in the same session
COMMITMENT_REUSE_VIOLATION	Model commitment value reused across distinct receipt IDs
SCOPE_NOT_STRICT_SUBSET	Sub-receipt allowedActions is not a strict proper subset of the parent receipt’s allowedActions
PARENT_SCOPE_VIOLATION	Sub-receipt failed parent scope containment check (receipt not found, time window overflow, chain depth exceeded, or orchestrator binding failure)
SESSION_LIFETIME_EXCEEDED	Session wall-clock lifetime exceeded maxLifetimeSeconds
TAU_SESSION_EXHAUSTED	Session anomaly capacity (tauSession) exhausted

Table 4

13.2. DRP Operation Types Registry

IANA is requested to create a registry titled "DRP Operation Types". The policy for new registrations is Specification Required. Initial values are:

Operation Type	Description
READ	Read access to a resource
WRITE	Write or modify a resource
EXECUTE	Execute a program or callable resource
DELETE	Delete a resource
DELEGATE	Issue a sub-receipt delegating a subset of scope

Table 5

13.3. DRP Boundary String Format Registry

IANA is requested to create a registry titled "DRP Boundary String Formats" describing the prohibition string format used in the boundaries array of a Delegation Receipt. The policy for new registrations is Specification Required.

The baseline format defined by this document is:

```
prohibition-string = "deny" ":" operation ":" resource
operation          = operation-type / "*"
resource           = resource-identifier / "*"
operation-type     = "read" / "write" / "delete" / "execute"
                   / "delegate"
resource-identifier= 1*( ALPHA / DIGIT / "-" / "_" / "/" )
```

The wildcard character "*" matches any value in its position. A prohibition string of "deny:write:*" prohibits all write operations on all resources. A prohibition string of "deny:delete:email" prohibits delete operations on the email resource specifically.

14. References

14.1. 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/info/rfc2119>.

[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/info/rfc8174>.

[RFC3161] Adams, C., Cain, P., Pinkas, D., and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161, DOI 10.17487/RFC3161, August 2001, <https://www.rfc-editor.org/info/rfc3161>.

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

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

14.2. Informative References

[NIST-AI-100-1]
National Institute of Standards and Technology, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)", NIST AI 100-1, January 2023, <https://doi.org/10.6028/NIST.AI.100-1>.

[W3C-WebAuthn]
Balfanz, D., "Web Authentication: An API for Accessing Public Key Credentials Level 2", W3C Recommendation webauthn-2, April 2021, <https://www.w3.org/TR/webauthn-2/>.

[FIDO2] FIDO Alliance, "Client to Authenticator Protocol (CTAP)", FIDO Alliance Proposed Standard CTAP-v2.1, June 2021, <https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

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

[NC2.5] Barziankou, M., "Navigational Cybernetics 2.5", DOI 10.17605/OSF.IO/NHTC5, 2026, <https://doi.org/10.17605/OSF.IO/NHTC5>.

[RFC9396] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", RFC 9396, DOI 10.17487/RFC9396, May 2023, <https://www.rfc-editor.org/info/rfc9396>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, August 2021, <https://www.rfc-editor.org/info/rfc8785>.

Appendix A. JSON Schema Definitions

A.1. Delegation Receipt Schema

The following JSON Schema (draft-07) defines the structure of a Delegation Receipt as specified in Section 4.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://authproof.dev/schemas/delegation-receipt-1.0.json",
  "title": "DelegationReceipt",
  "type": "object",
  "required": [
    "receiptId",
    "schemaVersion",
    "timeWindow",
    "publicKey",
    "scope",
    "boundaries",
    "operatorInstructionsHash",
    "canonicalPayload",
    "signature"
  ],
  "properties": {
```

```
    "receiptId": {
      "type": "string",
      "description": "Unique receipt identifier.",
      "pattern": "^rec_[0-9a-f]{16,}$"
    },
    "schemaVersion": {
      "type": "string",
      "description": "Schema version. MUST be 1.0.",
      "enum": ["1.0"]
    },
    "timeWindow": {
      "type": "object",
      "description": "The validity window for this receipt.
                      Verification MUST use the log-assigned TSA
                      timestamp, not the client clock.",
      "required": ["notBefore", "notAfter"],
      "properties": {
        "notBefore": {
          "type": "string",
          "format": "date-time",
          "description": "ISO 8601 datetime before which this
                          receipt is not valid."
        },
        "notAfter": {
          "type": "string",
          "format": "date-time",
          "description": "ISO 8601 datetime at which this receipt
                          expires.  Verification MUST fail after
                          this time."
        }
      }
    },
    "publicKey": {
      "description": "The user’s public key as a JSON Web Key (JWK). Ed25519 (kty: OKP, c
rv: Ed25519) is RECOMMENDED. ECDSA P-256 (kty: EC, crv: P-256) is also supported for comp
atibility.",
      "oneOf": [
        {
          "title": "Ed25519 (OKP)",
          "type": "object",
          "properties": {
            "kty": { "type": "string", "enum": ["OKP"] },
            "crv": { "type": "string", "enum": ["Ed25519"] },
            "x":   { "type": "string", "description": "Base64url-encoded public key bytes
 (32 bytes)" }
          },
          "required": ["kty", "crv", "x"],
          "additionalProperties": false
        },
        {
          "title": "ECDSA P-256 (EC)",
```

```
          "type": "object",
          "properties": {
            "kty": { "type": "string", "enum": ["EC"] },
            "crv": { "type": "string", "enum": ["P-256"] },
            "x":   { "type": "string", "description": "Base64url-encoded x coordinate" },
            "y":   { "type": "string", "description": "Base64url-encoded y coordinate" }
          },
          "required": ["kty", "crv", "x", "y"],
          "additionalProperties": false
        }
      ]
    },
    "scope": {
      "$ref": "#/definitions/ScopeSchema"
    },
    "boundaries": {
      "type": "array",
      "description": "Array of prohibition strings that MUST be
                      enforced regardless of scope or Operator
                      instruction.  These represent the User's
                      hard limits.",
      "items": { "type": "string" },
      "minItems": 1
    },
    "operatorInstructionsHash": {
      "type": "string",
      "description": "SHA-256 hash of the canonical operator
                      instructions string, formatted as
                      sha256:<hex>.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "operatorInstructions": {
      "type": "string",
      "description": "The plaintext Operator instruction string
                      whose SHA-256 hash is committed in
                      operatorInstructionsHash."
    },
    "modelCommitment": {
      "type": "string",
      "description": "OPTIONAL. Cryptographic measurement of
                      the model state at authorization time,
                      formatted as sha256:<hex>.  When present,
                      verification MUST fail if the current
                      model measurement does not match.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "metadata": {
      "type": "object",
```

```
      "description": "OPTIONAL. Arbitrary key-value pairs
                      included in the canonical payload and
                      covered by the signature.  MAY include
                      external delegation identifiers.",
      "additionalProperties": {
        "type": "string"
      }
    },
    "canonicalPayload": {
      "type": "string",
      "description": "Base64url-encoded canonical serialization of all receipt fields exc
ept canonicalPayload and signature. Covered by the receipt signature."
    },
    "signature": {
      "type": "string",
      "description": "Base64url-encoded ECDSA P-256 signature
                      over canonicalPayload."
    },
    "toolSchemaHash": {
      "type": "string",
      "description": "OPTIONAL. SHA-256 hash of the canonical
                      serialization of all tool schemas available
                      at delegation time, formatted as
                      sha256:<hex>.  When present, verification
                      MUST fail if the current tool schema hash
                      does not match.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "discoveryMetadata": {
      "type": "object",
      "description": "OPTIONAL. Scope Discovery Protocol metadata.
                      MUST be present when the receipt was produced
                      by the Scope Discovery Protocol.",
      "properties": {
        "observationCount": {
          "type": "integer",
          "description": "Number of operations intercepted during
                          the sandboxed observation session.",
          "minimum": 0
        },
        "abortedByTimeout": {
          "type": "boolean",
          "description": "True if the observation session was
                          terminated by timeout rather than
                          completion."
        },
        "riskFlags": {
          "type": "array",
          "description": "Risk flags raised during scope generation.",
```

```
          "items": { "type": "string" }
        }
      }
    },
    "logEntryHash": {
      "type": "string",
      "description": "OPTIONAL. SHA-256 hash of the append-only
                      log entry produced when this receipt was
                      published, formatted as sha256:<hex>.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "trustedSources": {
      "type": "array",
      "description": "OPTIONAL. Array of trusted instruction source
                      identifiers.  When present, Check 13 MUST
                      reject any action whose instructionSource is
                      not in this list.",
      "items": { "type": "string" }
    },
    "parentReceiptId": {
      "type": "string",
      "description": "OPTIONAL. The receiptId of the parent
                      Delegation Receipt in a multi-agent delegation
                      chain.  When present the receipt is a
                      sub-receipt subject to Check 14.",
      "pattern": "^rec_[0-9a-f]{16,}$"
    },
    "orchestratorSignature": {
      "type": "string",
      "description": "OPTIONAL. Base64url-encoded ECDSA P-256
                      binding signature by the Orchestrator's key
                      over 'orchestrator-delegation:' ||
                      parentReceiptId || ':' || receiptId.
                      Present only in sub-receipts.  Not included
                      in the signed body."
    },
    "providerUpdatePolicyId": {
      "type": "string",
      "description": "OPTIONAL. Opaque string identifying the providerUpdatePolicy config
uration entry in effect at issuance. Scoped to the operator's trust anchor. Covered by th
e receipt signature."
    }
  },
  "definitions": {
    "ScopeSchema": {
      "type": "object",
      "required": ["version", "allowedActions"],
      "properties": {
        "version": {
          "type": "string",
```

```
          "description": "Scope schema version.",
          "default": "1.0"
        },
        "allowedActions": {
          "type": "array",
          "description": "Explicit list of permitted actions.",
          "items": {
            "$ref": "#/definitions/ActionConstraint"
          }
        },
        "deniedActions": {
          "type": "array",
          "description": "Explicit list of prohibited actions.
                          Deny rules take precedence over allow
                          rules.",
          "items": {
            "$ref": "#/definitions/ActionConstraint"
          }
        }
      }
    },
    "ActionConstraint": {
      "type": "object",
      "required": ["operation", "resource"],
      "properties": {
        "operation": {
          "type": "string",
          "description": "The operation type. Wildcards (*) are
                          supported.",
          "examples": ["read", "write", "delete", "send", "*"]
        },
        "resource": {
          "type": "string",
          "description": "The resource identifier. Wildcards (*)
                          are supported.",
          "examples": ["email", "calendar", "database/*", "*"]
        },
        "constraints": {
          "type": "object",
          "description": "OPTIONAL. Argument-level constraints
                          on the action.",
          "additionalProperties": true
        }
      }
    }
  }
}
```

A.2. Action Log Entry Schema

The following JSON Schema defines the structure of an Action Log Entry as specified in Section 5.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://authproof.dev/schemas/action-log-entry-1.0.json",
  "title": "ActionLogEntry",
  "type": "object",
  "required": [
    "entryId",
    "receiptHash",
    "operation",
    "resource",
    "timestamp",
    "previousEntryHash",
    "entryHash"
  ],
  "properties": {
    "entryId": {
      "type": "string",
      "description": "Unique identifier for this log entry."
    },
    "receiptHash": {
      "type": "string",
      "description": "SHA-256 hash of the delegation receipt
                      that authorized this action, formatted
                      as sha256:<hex>.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "operation": {
      "type": "string",
      "description": "The operation that was executed."
    },
    "resource": {
      "type": "string",
      "description": "The resource that was accessed."
    },
    "timestamp": {
      "type": "string",
      "format": "date-time",
      "description": "RFC 3161 trusted timestamp of this
                      log entry."
    },
    "previousEntryHash": {
      "type": "string",
      "description": "SHA-256 hash of the previous log entry.
```

```
                          The first entry in a log uses a
                          well-known genesis hash.",
        "pattern": "^sha256:[0-9a-f]{64}$"
      },
      "entryHash": {
        "type": "string",
        "description": "SHA-256 hash of this entry's canonical
                        serialization excluding entryHash.",
        "pattern": "^sha256:[0-9a-f]{64}$"
      },
      "decision": {
        "type": "string",
        "description": "The verification decision for this
                        action.",
        "enum": ["ALLOW", "REQUIRE_APPROVAL", "BLOCK"]
      },
      "riskScore": {
        "type": "number",
        "description": "OPTIONAL. Session risk score at the
                        time of this action.",
        "minimum": 0,
        "maximum": 100
      }
    }
  }
```

A.3. Session State Schema

The following JSON Schema defines the structure of a Session State object as specified in Section 9.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://authproof.dev/schemas/session-state-1.0.json",
  "title": "SessionState",
  "type": "object",
  "required": [
    "sessionId",
    "receiptHash",
    "trustScore",
    "status",
    "startedAt",
    "actionCount"
  ],
  "properties": {
    "sessionId": {
      "type": "string",
      "description": "Unique session identifier."
```

```
    },
    "receiptHash": {
      "type": "string",
      "description": "SHA-256 hash of the delegation receipt
                      that initiated this session.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "trustScore": {
      "type": "number",
      "description": "Current session trust score. Starts at
                      100 and decays on anomaly detection.",
      "minimum": 0,
      "maximum": 100
    },
    "status": {
      "type": "string",
      "description": "Current session status.",
      "enum": ["ACTIVE", "DEGRADED", "SUSPENDED"]
    },
    "startedAt": {
      "type": "string",
      "format": "date-time",
      "description": "ISO 8601 datetime at which this session
                      was initiated."
    },
    "lastActionAt": {
      "type": "string",
      "format": "date-time",
      "description": "ISO 8601 datetime of the most recent
                      action in this session."
    },
    "actionCount": {
      "type": "integer",
      "description": "Total number of actions evaluated in
                      this session.",
      "minimum": 0
    },
    "anomalyCount": {
      "type": "integer",
      "description": "Total number of anomalies detected in
                      this session.",
      "minimum": 0
    },
    "sensitivityLevel": {
      "type": "string",
      "description": "Highest sensitivity level detected in
                      this session.",
      "enum": ["PUBLIC", "INTERNAL", "CONFIDENTIAL",
```

```
              "RESTRICTED"]
    },
    "maxLifetimeSeconds": {
      "type": "integer",
      "description": "Maximum session lifetime in seconds.
                      When elapsed time since startedAt exceeds
                      this value, the session MUST be terminated
                      and reauthorization required.  Default is
                      90000 (25 hours).",
      "minimum": 1,
      "default": 90000
    },
    "tauSession": {
      "type": "number",
      "description": "Session anomaly capacity. Initialized to sessionCapacity (default 1
00) and decremented by anomaly weight on each detection event. A verifier MUST deny actio
ns when tauSession falls at or below tauMin (default 10).",
      "default": 100
    },
    "cumulativeAnomalyMass": {
      "type": "number",
      "description": "Running sum of anomaly weights observed
                      during the session. MUST be initialized to
                      0.0 and incremented by the anomaly weight
                      on each detection event.",
      "default": 0.0
    },
    "passivePressureRate": {
      "type": "number",
      "description": "Rate at which ambient environmental signals
                      contribute to session risk without discrete
                      anomaly events. MUST be initialized to 0.0.",
      "default": 0.0
    }
  }
}
```

Appendix B. Example JSON Objects

B.1. Example DelegationReceipt

The following is a complete example of a Delegation Receipt JSON object with realistic but fictional values:

```
{
  "receiptId": "rec_a3f8c2d1e9b047560f1234567890abcd",
  "schemaVersion": "1.0",
  "timeWindow": {
    "notBefore": "2026-05-21T00:00:00Z",
    "notAfter":  "2026-05-22T00:00:00Z"
  },
  "publicKey": {
    "kty": "OKP",
    "crv": "Ed25519",
    "x": "PLACEHOLDER_BASE64URL_ED25519_PUBLIC_KEY"
  },
  "scope": {
    "allowedActions": [
      { "operation": "read",  "resource": "email" },
      { "operation": "write", "resource": "calendar" }
    ],
    "deniedActions": [
      { "operation": "delete", "resource": "*" },
      { "operation": "execute", "resource": "*" }
    ]
  },
  "boundaries": [
    "deny:write:*",
    "deny:delete:*",
    "deny:execute:*"
  ],
  "operatorInstructionsHash":
    "sha256:e10dd1f5de5b07fa9f9d32fa13371fefa84c5dc31ae8382cfc7dbaeea0dcd2f9",
  "operatorInstructions": "Summarize unread emails and add meeting summaries to calendar.
",
  "canonicalPayload": "PLACEHOLDER_BASE64URL_CANONICAL_PAYLOAD",
  "signature":        "PLACEHOLDER_BASE64URL_ECDSA_SIGNATURE"
}
```


B.2. Example Scope Discovery Output

The following is an example JSON response from the Scope Discovery Protocol after a sandboxed observation session:

```
{
  "agentId": "agent_email-summarizer-v2",
  "supportedActions": [
    { "operation": "read",  "resource": "email" },
    { "operation": "write", "resource": "calendar" }
  ],
  "boundaries": [
    "deny:delete:*",
    "deny:execute:*",
    "deny:write:email"
  ],
  "riskFlags": [],
  "discoveryTimestamp": "2026-05-21T10:30:00Z",
  "observationCount": 47,
  "abortedByTimeout": false
}
```

Acknowledgements

The authors thank the IETF WIMSE, OAuth, and SCITT working groups for their work on workload identity, token exchange, and supply chain integrity, which informed the design of this protocol.

The formal analysis of session state properties in this document benefited from review and guidance by Maksim Barziankou. The treatment of structural burden, viability budgets, and admissibility predicates in Section 9 and Section 11 draws on primitives formalized in Navigational Cybernetics 2.5 [NC2.5].

Author’s Address

Ryan Nelson
Authproof
Clinton, Oklahoma
United States of America
Email: ryan@authproof.dev
URI: https://authproof.dev