Network Working Group
Internet-Draft
Intended status: Informational
Expires: 14 November 2026

R. Nelson
Authproof

13 May 2026

Delegation Receipt Protocol for AI Agent Authorization
draft-nelson-agent-delegation-receipts-07

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 removes the operator as a trusted third party 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 14 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:

delegationId:
: The SHA-256 hash of the canonical serialization of the Authorization Object body (all fields except delegationId and signature). Encoded as the string "sha256:" followed by the lowercase hex digest.

version:
: Protocol version string. This document defines version "1".

scope:
: An object with four keys: "reads", "writes", "deletes", and "executes", each containing an array of permitted resource:operation strings. The "executes" array *MUST* contain SHA-256 hashes of the static capability DAG of each authorized program; program names or URIs *MUST NOT* be used in the "executes" array. 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.

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.

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

operatorInstructions:
: The plaintext instruction string whose SHA-256 hash is recorded in instructionHash.

signerPublicKey:
The User's public key as a JSON Web Key [RFC7517]. Implementations *SHOULD* use ECDSA P-256 (crv: "P-256") [RFC7518] keys.

signature:
The ECDSA P-256 signature over the canonical serialization of the Authorization Object body, encoded as base64url.

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.

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

The complete structure is illustrated below:

```
{
  "delegationId":  "<sha256-of-canonical-body>",
  "version":       "1",
  "scope": {
    "reads":    ["<resource>:<operation>"],
    "writes":   ["<resource>:<operation>"],
    "deletes":  ["<resource>:<operation>"],
    "executes": ["sha256:<capability-dag-hash>"]
  },
  "boundaries": ["<prohibition-string>"],
  "timeWindow": {
    "notBefore": "<ISO-8601-timestamp>",
    "notAfter":  "<ISO-8601-timestamp>"
  },
  "instructionHash":      "sha256:<hex-digest>",
  "operatorInstructions": "<operator-instruction-text>",
  "signerPublicKey":      { "<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.

The delegationId is computed as:

```
delegationId = "sha256:" ||
               lowercase_hex(SHA-256(canonical_body))
```

Implementations *MUST* compute the delegationId 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 via the WebAuthn/FIDO2 API [W3C-WebAuthn] [FIDO2]. Hardware key custody (Trusted Platform Module or device secure enclave) is *RECOMMENDED*.

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 (delegationId).

* 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.

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.

The complete verification procedure is:

```
Verify(receipt, action):
  (1) if Revoked(receipt.delegationId)
                                      -> fail
  (2) if not VerifySig(receipt.signature,
                       canonical(receipt.body),
                       receipt.signerPublicKey)
                                      -> 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 action.type == EXECUTE and
         Hash(SafescriptDAG(program))
         != receipt.scope.executes[n]
                                      -> fail
  (7) if Hash(currentOperatorInstructions)
         != receipt.instructionHash
                                      -> 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):
Computes the static capability signature of the actual program the agent has been given and compares it to the hash committed in the "executes" scope field. Substituting a different program after the receipt is signed is detectable without runtime introspection.

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.

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.

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 seven-check verification algorithm as a formal function:

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

  CHECK 2: Revocation Status
    IF revocationRegistry.isRevoked(receipt.receiptId) THEN
      RETURN DENY, "RECEIPT_REVOKED"

  CHECK 3: Time Window
    IF receipt.expiresAt < NOW() THEN
      RETURN DENY, "RECEIPT_EXPIRED"
    IF receipt.createdAt > 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"
    IF action.operation IN receipt.scope.deniedActions THEN
```

```
      RETURN DENY, "ACTION_EXPLICITLY_DENIED"

  CHECK 5: Operator Instruction Hash
    currentHash = SHA-256(canonicalize(operatorInstructions))
    IF currentHash != receipt.operatorInstructionsHash THEN
      RETURN DENY, "OPERATOR_INSTRUCTIONS_MISMATCH"

  CHECK 6: 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 7: 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 8: 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"

  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.
TAU_SESSION_EXHAUSTED	Session anomaly capacity exhausted: tauSession has fallen to or below tauMin. Execution blocked regardless of trustScore. Not resettable by reauthorization.

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
)
```

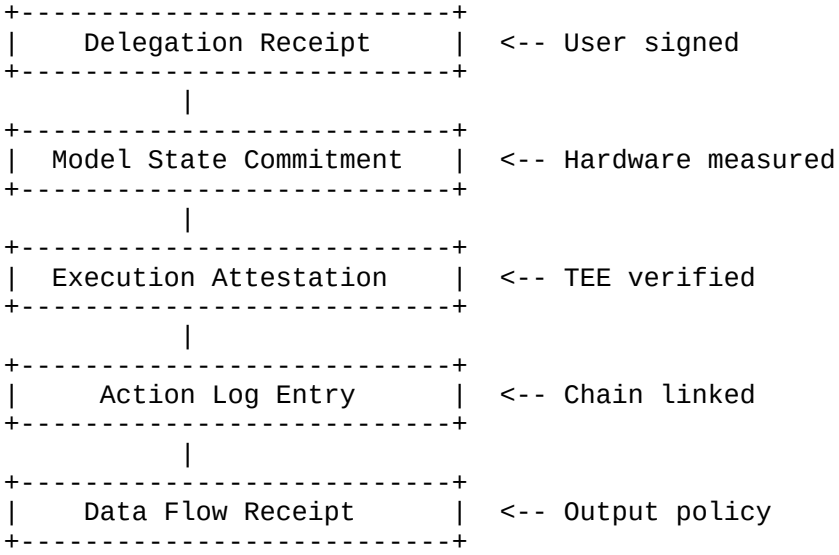
Including receiptHash as the fifth component binds the model measurement to the specific delegation. The same model with the same system prompt but a different receipt produces a different measurement. A commitment *MUST NOT* be reused across delegations.

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.

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 ModelDriftDetected error 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:

"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.

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. Once a session's anomaly capacity is exhausted, the session is permanently closed to further execution.

Trust decays when anomalies are detected:

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

Anomaly severity levels are defined as:

```
Prompt injection detected        : severity 5
Sensitive data in external dest. : severity 4
Frequency spike                  : severity 3
Scope edge usage                 : severity 2
First-time action                : severity 1
```

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, making previously marginal decisions tip into REQUIRE_APPROVAL or BLOCK territory.

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 needs to hand off a subtask to another Agent, the chain of authority *MUST* remain auditable and bounded. DRP enforces three invariants at every delegation hop.

10.1. Scope Attenuation

Each delegation step *MUST* produce a strict proper subset of the parent's authorized scope. Specifically:

1. Every action the child Agent is permitted *MUST* already appear in the parent's allowedActions list. An Agent *MUST NOT* grant permissions it was not itself given.

2. The child *MUST* have strictly fewer permitted actions than the parent. Equal scope *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.

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 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.

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.

10.2. 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).

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.

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.

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.

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(SafescriptDAG(program)) equals the hash committed in C (execution hash check); (vii) the SHA-256 hash of the Operator's current instructions equals R.instructionHash (instruction hash check); and (viii) if R.toolSchemaHash is present, the SHA-256 hash of the current tool schema equals R.toolSchemaHash (tool schema hash check).

For any action a, Verify(R, a) = true if and only if all of (i)-(viii) 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. The signing layer of the protocol is therefore not post-quantum secure under current implementations. The migration path 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.

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.

11.4. Degraded Operation

When the verifier cannot construct the required authorization state due to log unavailability or unverifiable revocation status, 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.

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.

12. IANA Considerations

This document has no IANA actions.

13. References

13.1. 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>.

13.2. 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>.

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",
    "createdAt",
    "expiresAt",
    "publicKey",
    "scope",
    "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"]
    },
    "createdAt": {
      "type": "string",
      "format": "date-time",
      "description": "ISO 8601 datetime at which this receipt
                      was created and signed."
    },
    "expiresAt": {
      "type": "string",
      "format": "date-time",
      "description": "ISO 8601 datetime at which this receipt
                      expires.  Verification MUST fail after
                      this time."
    },
    "publicKey": {
      "type": "object",
      "description": "The authorizing user's public key in
```

```
                    JSON Web Key format (RFC 7517).",
      "required": ["kty", "crv", "x", "y"],
      "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."
        }
      }
    },
    "scope": {
      "$ref": "#/definitions/ScopeSchema"
    },
    "operatorInstructionsHash": {
      "type": "string",
      "description": "SHA-256 hash of the canonical operator
                      instructions string, formatted as
                      sha256:<hex>.",
      "pattern": "^sha256:[0-9a-f]{64}$"
    },
    "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 JSON
                        serialization of all fields except
                        signature, over which the signature
                        is computed."
      },
      "signature": {
        "type": "string",
        "description": "Base64url-encoded ECDSA P-256 signature
                        over canonicalPayload."
      },
      "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}$"
      }
    },
    "definitions": {
      "ScopeSchema": {
        "type": "object",
        "required": ["version", "allowedActions"],
        "properties": {
          "version": {
            "type": "string",
            "description": "Scope schema version.",
            "default": "1.0"
          },
          "allowedActions": {
            "type": "array",
            "description": "Explicit list of permitted actions.",
            "items": {
              "$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"]
      }
    }
  }
```

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