Mandate: A Delegation-Centric Authorization Profile for Autonomous Agent Commerce

Rishabh Sai

May 2026

Abstract

Autonomous agents increasingly interact with external services, commerce systems, payment adapters, and other agents on behalf of users and organizations. Existing authentication systems can identify software clients, but they do not by themselves answer whether an agent is acting under a current, constrained, user-approved mandate, whether the presented token is bound to the agent's key, whether the counterparty is legitimate, and whether the outcome can be audited without leaking private intent. This paper proposes Mandate, a delegation-centric authorization profile that composes existing identity and authorization rails: OAuth-style scoped tokens, proof-of-possession binding, verifiable credentials, signed service metadata, and optional public hash anchoring. Mandate defines five protocol objects: agent credential, user mandate, bound token, service metadata, and signed receipt. It also defines a bilateral verification procedure in which the service verifies the calling agent and mandate, while the calling agent verifies the service before transmitting sensitive instructions. Public chains are treated as optional audit and registry infrastructure, not as the runtime authorization layer. The design keeps private mandates and receipts off-chain while permitting hashes, revocation status, registry entries, and reputation records to be externally verifiable.

1 Introduction

Agents are moving from interface automation to direct service interaction. A travel agent can search fares, create a booking hold, and hand off final payment. A procurement agent can request quotes from supplier agents. A coding agent can invoke remote tools through Model Context Protocol (MCP) servers. In each case, the receiving service must answer more than "is this client authenticated?" It must also answer: who authorized the agent, what action was authorized, what constraints apply, whether the authorization is still current, whether the token presenter holds the expected private key, and whether the service itself is the intended counterparty.

Mandate starts from the claim that agent commerce needs verifiable delegation, not merely agent identity. The central object is a signed user mandate: a constrained statement of delegated authority addressed to a specific audience. The mandate is used with an agent credential, a sender-constrained token, signed counterparty metadata, and a signed receipt. This structure lets agents and services operate through existing rails such as OAuth, DPoP, mutual TLS certificate-bound tokens, W3C Verifiable Credentials, DIDs, A2A, MCP authorization, AP2-style payment mandates, x402, ACP delegated payment credentials, and optional registry systems rather than requiring a monolithic new protocol [15, 8, 4, 12, 20, 19, 14, 1, 2, 3, 21, 7, 5].

2 Contributions

This paper makes four contributions:

1. **A delegation-centric protocol profile.** Mandate separates agent identity, user authorization, runtime access, counterparty authenticity, and outcome audit into independently verifiable objects.

2. **A bilateral verification procedure.** The service verifies the user agent's credential, mandate, token binding, nonce, and policy constraints; the user agent verifies the service's signed metadata before releasing sensitive instructions.

3. **A public-chain privacy boundary.** The profile treats blockchains as optional commitment, revocation, registry, or reputation rails. They do not carry private mandates, travel details, payment details, or full receipts.

4. **A reproducible reference path.** The accompanying implementation defines Ed25519 envelopes, canonical JSON hashing, shared JavaScript/Python test vectors, a Solana devnet memo adapter for receipt-hash anchoring, and a deployed Agent Passport sandbox.

The intended contribution is a profile and reference implementation, not a new consensus protocol, payment network, identity namespace, or model-safety system.

3 Design Goals

1. **Constrained delegation.** Agents should present a narrow mandate with action, audience, expiry, nonce, scope, and user constraints.

2. **Bilateral verification.** Services verify caller authority. Callers verify service metadata before transmitting mandates or payment instructions.

3. **Sender-constrained runtime access.** Tokens should be bound to a key through DPoP or mutual TLS-style confirmation, reducing bearer token replay risk.

4. **Privacy-preserving auditability.** Private intent, travel details, payment details, and full receipts remain off-chain. Public anchors contain only hashes or status.

5. **Compatibility.** The profile should compose existing standards instead of replacing OAuth, verifiable credentials, A2A, MCP, or payment protocols.

4 System Model and Trust Assumptions

The system contains five roles. A principal P is a user or organization delegating authority. A user agent C acts on behalf of P. An issuer I signs credentials for agents, runtimes, or providers. A service agent or protected resource S receives requests and produces receipts. Optional public infrastructure L stores commitments, registry state, revocation pointers, or reputation records.

Mandate assumes standard public-key cryptography, collision-resistant hashing over canonicalized JSON, authenticated HTTPS transport, and an issuer trust configuration chosen by each verifier. It does not assume that the language model inside C is honest. It assumes instead that the runtime can be forced to present explicit signed objects before privileged actions execute.

The profile makes three trust boundaries explicit:

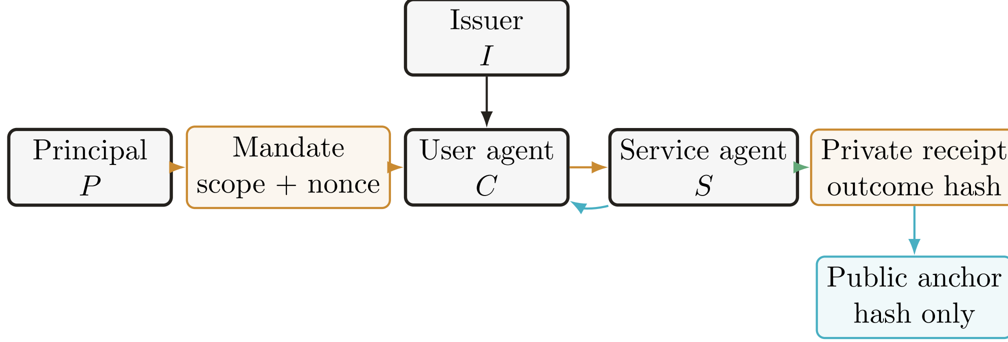

Figure 1: Mandate separates principal delegation, agent identity, runtime access, service authenticity, private receipts, and optional public commitments.

- **Issuer trust.** A verifier decides which issuers can make claims about agents or services.
- **Runtime key custody.** Sender-constrained tokens reduce stolen-token replay, but a compromised runtime that controls the private key can still act until expiry or revocation.
- **Policy enforcement.** Mandate provides machine-verifiable inputs to policy. It does not decide whether a user's high-level intent was wise, manipulated, or economically optimal.

5 Formal Model

Mandate models delegated execution as a decision over signed evidence rather than as an inference over conversational state. Let $\mathsf{canon}(x)$ be the deterministic JSON serialization of object x, $\mathsf{H}(x)$ be a collision-resistant hash over $\mathsf{canon}(x)$, $\mathsf{Sig}_k(x)$ be a signature under key k, and $\mathsf{jkt}(K)$ be a public-key thumbprint. A protocol exchange is a tuple

$$E = (G, M, T, R_S, Q, N)$$

where G is an agent credential, M is a mandate, T is a bound token, R_S is signed service metadata, Q is the concrete request, and N is verifier nonce state. A verifier evaluates E against a local policy Π and accepts only if the following predicates all hold:

$$\begin{aligned}
\mathsf{Accept}(E, \Pi) \iff\ & \mathsf{TrustedIssuer}(G.issuer, \Pi) \wedge \mathsf{Active}(G) \wedge \mathsf{Verify}(M, G.K) \wedge \\
& M.agent = G.subject \wedge M.audience = T.aud = R_S.audience \wedge T.cnf.jkt = \mathsf{jkt}(G.K) \wedge \\
& M.action \in T.scope \wedge M.action \in R_S.accepts \wedge \mathsf{Fresh}(M, T, R_S) \wedge \\
& \neg \mathsf{Seen}(N, M.nonce, M.audience, M.action) \wedge \\
& \mathsf{PolicyAccepts}(\Pi, M, Q).
\end{aligned}$$

This formulation deliberately leaves issuer governance, policy language, revocation transport, and payment adapter semantics to deployments. The profile specifies the evidence that must be bound before those local decisions are made.

6 Threat Model

Mandate focuses on common failures in delegated agent commerce:

- **Agent impersonation.** A malicious client claims to be a user's agent.

- **Stolen bearer token replay.** A valid token is copied and replayed from a different runtime.
- **Overbroad delegation.** A user grants a general permission when the service only needs a bounded mandate.
- **Counterparty spoofing.** An agent sends private intent or payment instructions to a fake service endpoint.
- **Replay.** A mandate or receipt is reused outside its nonce, audience, or validity window.
- **Payment escalation.** A service converts a search or hold permission into a purchase without final approval.
- **Audit leakage.** Sensitive mandates or receipts are published to a public ledger.

Recent agentic-commerce security work argues that payment and commerce agents introduce attack classes across transaction authorization, inter-agent trust, market manipulation, prompt injection, replay, and context binding [13, 11, 6]. Mandate addresses only the authorization evidence layer. Host compromise, malicious model behavior, prompt-injection resilience, policy errors, issuer governance, market manipulation, and regulatory compliance require additional controls. The profile provides a cryptographic and protocol layer for delegated authorization decisions.

7 Protocol Objects

7.1 Agent Credential

An agent credential is an issuer-signed claim about an agent or runtime. It contains the agent identifier, provider identifier, public key thumbprint, supported protocols, assurance level, issuer, issuance time, and credential status. It can be represented as a W3C Verifiable Credential or an issuer-signed JWT depending on the deployment.

7.2 User Mandate

A user mandate is a signed delegation object:

```
{
  "type": "UserMandate",
  "version": "0.1",
  "principal": "did:example:user",
  "agent": "did:web:agent.builder.example",
  "audience": "https://airline.example/a2a",
  "action": "flight.hold.create",
  "constraints": {
    "maxSpendUsd": 500,
    "requiresFinalApproval": true
  },
  "expiresAt": "2026-05-08T15:00:00Z",
  "nonce": "nonce_..."
}
```

The mandate is not a wallet asset or a transferable token. It is a scoped instruction that should be audience-bound, short-lived, and revocable.

7.3 Bound Token

The bound token is a short-lived access token whose confirmation claim identifies the agent's public key. OAuth DPoP and OAuth mutual TLS are examples of sender-constrained token mechanisms [8, 12]. The token references the mandate hash and includes the scopes required by the service.

7.4 Service Metadata

The service publishes signed metadata describing its endpoint, accepted scopes, verification requirements, payment adapter, receipt signing key, and supported protocols. This can be exposed through existing resource metadata, A2A agent cards, MCP authorization metadata, or a service-specific well-known endpoint [10, 1, 14].

7.5 Receipt

The receipt is a service-signed record of the outcome. It binds the service identity, mandate hash, token id, request hash, outcome, and timestamp. Sensitive receipt data remains private. A hash of the receipt can be anchored publicly.

8 Message Grammar and Canonicalization

The reference implementation uses canonical JSON and compact signed envelopes so JavaScript and Python implementations compute identical hashes and signatures. Table 1 summarizes the minimum interoperable message grammar. Deployments MAY encode the same fields as JWTs or W3C Verifiable Credentials if they preserve the binding semantics.

Table 1: Minimum message grammar for the Mandate profile.

Object	**Signed by**	**Required fields**
Agent credential G	Issuer	subject, issuer, public-key thumbprint, status, validity window.
Mandate M	Agent or principal-authorized key	principal, agent, audience, action, constraints, issuedAt, expiresAt, nonce.
Bound token T	Authorization server or service	aud, scope, cnf.jkt, expiresAt, mandate hash or equivalent reference.
Service metadata R_S	Service or trusted issuer	audience, endpoint, accepted actions, receipt key, payment adapter, validity window.
Request Q	Agent key or transport proof	method, endpoint, request body hash, DPoP or mTLS proof reference.
Receipt R	Service receipt key	service id, mandate hash, token id, request hash, outcome, issuedAt.
Anchor A	Public ledger transaction signer	anchor hash, receipt hash, mandate hash, service key thumbprint, chain, issuedAt.

All hashes in the reference vectors are computed as $\mathsf{H}(\mathsf{canon}(x))$. The canonicalizer sorts object keys, omits undefined fields, preserves arrays in order, and serializes without insignificant

whitespace. This is intentionally narrower than general JSON canonicalization standards; production deployments SHOULD profile an existing canonicalization standard and keep the same cross-language test-vector discipline.

9 Binding Invariants

Mandate is useful only if each proof object is bound to the next object in the chain. The reference profile therefore requires the following invariants:

Invariant	**Required binding**
Agent binding	The mandate names the agent identifier, and the mandate signature verifies under the agent key referenced by the active credential.
Audience binding	The mandate audience, token audience, service metadata audience, and HTTPS endpoint identify the same verifier.
Key binding	The token confirmation claim identifies the same public-key thumbprint as the credential or mandate.
Action binding	The mandate action is included in the token scope and accepted by service metadata.
Time binding	Credential, mandate, token, and metadata validity windows are current at execution time.
Replay binding	The tuple $(M.nonce, M.audience, M.action)$ is consume-once under the verifier's replay cache.
Receipt binding	The receipt signs the mandate hash, token id, service id, request hash, outcome, and timestamp.
Anchor binding	The public anchor contains only hashes or registry state that can be recomputed from private records.

These invariants turn a collection of familiar standards into a single authorization decision. A verifier MUST reject an exchange if any binding is absent, ambiguous, stale, or inconsistent.

10 Bilateral Verification

Let C be the calling agent, S the service, M the mandate, T the bound token, G the agent credential, and R_S the signed service metadata. The service accepts a request only if:

1. G is issuer-trusted and active.
2. M is signed by an authorized key and names C.
3. $M.audience = S$ and $T.aud = S$.
4. T is sender-constrained to the public key in G.
5. $T.scope$ permits $M.action$.
6. M and T are unexpired and nonce replay checks pass.
7. Local policy accepts all constraints in M.

The calling agent also verifies S before sending sensitive instructions:

1. the endpoint is domain and TLS consistent with the intended audience;
2. R_S is signed by an expected key or trusted issuer;
3. R_S accepts the requested action and payment adapter;
4. the receipt signing key is declared and current.

11 Verification Algorithm

The reference verifier is intentionally deterministic. It does not infer authority from natural-language intent, model output, or a human-like session. It accepts only a chain of signed and audience-bound objects.

```
verify(exchange, policy):
  input: credential G, mandate M, token T,
         service metadata S, request Q

  require trusted_issuer(G.issuer)
  require credential_status(G) == active
  require verify_signature(M, key=G.public_key)
  require M.agent == G.subject
  require M.audience == S.audience
  require verify_signature(S, key=trusted_service_key(S))
  require tls_endpoint_matches(S.endpoint, M.audience)
  require verify_signature(T, key=G.public_key)
  require T.cnf.jkt == thumbprint(G.public_key)
  require T.aud == M.audience
  require M.action in T.scope
  require now < M.expires_at and now < T.expires_at
  require nonce_not_seen(M.nonce, M.audience)
  require policy_accepts(M.constraints, Q)

  receipt = sign_receipt(S.receipt_key, hash(M), hash(Q), outcome)
  return accept(receipt)
```

This algorithm separates two roles that are often merged in demos. The mandate expresses delegated user authority. The token expresses runtime access. The service metadata expresses counterparty authenticity. The receipt expresses what happened. A valid token without a current mandate is insufficient. A valid mandate sent to the wrong service is insufficient. A service that cannot prove its metadata is not eligible to receive sensitive instructions.

12 Security Properties

The following properties are proof sketches over the formal model and reference verifier. They are not a universal proof of every deployment because issuer governance, UI consent, revocation liveness, policy authoring, and key custody remain outside the profile.

Replay resistance under verifier state. If the verifier records the mandate nonce, audience, and action after accepting an exchange, then a second presentation of the same mandate to the same audience for the same action is rejected by the $\neg \mathsf{Seen}$ predicate. Expiry bounds limit replay attempts against verifiers that have not yet synchronized state. This property requires consistent nonce storage across service replicas.

Audience separation. If an attacker forwards a mandate issued for audience S_1 to service S_2, acceptance requires $M.audience = T.aud = R_S.audience$. Unless S_2 can produce trusted metadata for S_1 and a token audience for S_1, the exchange fails the audience-binding predicate. HTTPS endpoint checks further bind the logical audience to the transport destination.

Stolen-token replay reduction. A copied token is insufficient without possession of the private key whose thumbprint appears in $T.cnf.jkt$. DPoP or mTLS proof-of-possession therefore reduces replay from a different runtime. The property does not protect against compromise of the runtime that legitimately holds the key.

Private/public audit boundary. If the public ledger receives only anchor hash, receipt hash, mandate hash, service key thumbprint, chain name, and issuance time, and if M and R contain high-entropy nonces, then the ledger does not reveal the mandate or receipt contents except through correlation and dictionary attacks against predictable fields. This is why Mandate treats the chain as a commitment layer rather than an authorization layer.

13 Threat Mitigation Matrix

Threat	**Mandate control**	**Residual risk**
Agent impersonation	Issuer-signed credential and mandate signature bound to the agent key.	Issuer governance and key custody remain deployment risks.
Bearer token replay	DPoP or mTLS-style confirmation binds the token to a private key.	A compromised runtime can still act until revocation or expiry.
Overbroad delegation	Mandates include action, audience, expiry, nonce, and constraints.	User interfaces must avoid tricking users into broad mandates.
Counterparty spoofing	The caller verifies signed service metadata before sending the mandate.	Trust anchors for service keys must be managed.
Replay	Audience binding, expiry, and nonce storage reject reused mandates.	Distributed nonce storage is required across service replicas.
Payment escalation	Mandates can require final approval and separate search, hold, and pay scopes.	Payment adapters need independent policy checks.
Audit leakage	Public chains receive hashes or status, not private mandates or receipts.	Hash correlation can still leak metadata if inputs are predictable.

14 Public Anchoring

Public chains are useful for neutral registries, revocation status, validation records, reputation, and receipt hash anchoring. They are not appropriate for full mandates or receipts because those objects can reveal user intent, travel plans, commercial terms, or payment constraints.

The initial Mandate Solana adapter anchors a compact memo payload through a normal Solana transaction instruction and the Memo program [16, 17]:

```
{
  "type": "mandate.receipt_anchor",
  "version": "0.1",
  "anchorHash": "...",
  "receiptHash": "...",
  "mandateHash": "...",
  "issuedAt": "..."
}
```

The memo does not authorize the agent. It only provides a public timestamped commitment to a private receipt or status record.

15 Privacy Classification

The core privacy rule is that on-chain data must be sufficient to verify a commitment but insufficient to reconstruct the user mandate or receipt. Table 2 classifies common fields.

If a mandate contains predictable fields, a bare hash can be vulnerable to dictionary reconstruction. Deployments SHOULD include high-entropy nonces or salts in private objects before anchoring commitments, and SHOULD avoid publishing repeated relationship identifiers when linkability is not required.

16 Reference Implementation

The reference implementation is split into packages:

- `@mandateprotocol/core`: mandate construction, canonical JSON, Ed25519 envelopes, hashes, receipt anchors, verification checks.
- `@mandateprotocol/solana`: Solana memo instruction and transaction helpers for receipt hash anchoring.
- `mandate-protocol`: Python package with the same core primitives.
- Shared test vectors: JavaScript and Python verification over identical signed mandates, receipts, and anchors.
- Website demo: browser WebCrypto proof flow showing issue, verify, deny, and Solana devnet anchor states.
- Agent Passport app: wallet-only onboarding, local agent-key generation, Agent Passport metadata, a Solana devnet NFT mint path, a shareable public passport route, an OpenRouter-backed sandbox travel-agent tool loop, wallet-signed approval, service verification, and signed receipts.

Table 2: Privacy classification for common Mandate fields.

Field	Publication class	Reason
Agent public key thumbprint	Public or registry	Needed for key discovery and revocation.
Issuer identifier	Public or registry	Needed for verifier trust decisions.
Mandate body	Never public	Can reveal user intent, destination, commercial terms, or constraints.
Payment credential or token	Never public	Even scoped payment credentials remain high-value secrets.
Receipt body	Private by default	Can reveal purchase details, supplier terms, or itinerary.
Mandate hash	Hash only	Useful for audit, but vulnerable to guessing if mandate fields are low entropy.
Receipt hash	Hash only	Useful for audit when the private receipt can later be disclosed selectively.
Revocation status	Public or hash only	Public status improves verifier liveness but can leak relationship graphs.
Reputation record	Public if consented	Useful in open marketplaces, but can create profiling risk.

16.1 Interactive Agent Passport Sandbox

The deployed companion artifact at `https://www.mandate.lol/app` makes the profile inspectable in a browser. The first flow creates a wallet-owned Agent Passport for a local agent key and exposes a shareable `/app/agent/<mint>` page. The second flow asks the agent to find the cheapest Dubai-to-San Francisco itinerary in a deterministic Mandate Air sandbox. Before the agent may create a hold, the interface presents a scoped mandate for wallet approval. The proof drawer then shows the Agent Passport, signed mandate, service metadata, tool trace, hold receipt, receipt hash, and optional public receipt anchor.

This artifact is not evidence that real airline booking is safe or that all wallets, RPC providers, and model providers behave identically. It is implementation evidence for three claims that are otherwise abstract in the paper: the user can inspect current delegated authority before execution; the agent can verify a counterparty before sending sensitive instructions; and the service outcome can be bound back to the mandate and receipt trail. The same surface is also a base for future interactive-steering studies in which users pause, narrow, revoke, approve, or contest an agent's authority during a long-running task.

17 Evaluation

The reference implementation includes reproducible scripts for policy-vector coverage, payload sizing, and verifier latency. The current evidence artifacts are generated by `npm run paper:evidence` and written to `paper/results/`. The shared policy fixture contains three positive authorization

scenarios and ten negative authorization cases. The positive cases cover a flight hold, a procurement quote request, and an MCP-style repository tool read. The negative cases cover expired mandate, expired token, mismatched token audience, missing scope, wrong key binding, revoked credential, replayed nonce, spoofed service audience, unsupported service action, and payment escalation attempt. JavaScript and Python both verify the same expected decision and primary failed check for every policy vector.

Scenario	**Allowed action**	**Reason it matters**
Travel hold	`flight.hold.create`	Separates search and hold from final payment authorization.
Procurement quote	`quote.request.create`	Tests business-to-business delegation without immediate payment.
MCP tool read	`repo.issue.read`	Exercises tool authorization where the service is an MCP-style resource.

Evaluation item	**Method**	**Expected artifact**
Cross-language determinism	Run JavaScript and Python over the same canonical JSON vectors.	Identical mandate hashes, receipt hashes, signatures, and anchor hashes.
Negative authorization cases	Mutate one field at a time: audience, scope, expiry, nonce, and key binding.	Verifier rejects with a specific failed check.
Counterparty verification	Replace service metadata key or endpoint.	Caller refuses to transmit sensitive mandate data.
On-chain anchoring	Submit a Solana devnet memo containing only the anchor payload.	Explorer-visible transaction with no private mandate or receipt data.
Performance	Measure local signature verification and policy checks across batches.	Latency and payload-size table.
Browser artifact	Run app smoke tests against the deployed `/app` surface and forced AI-provider tool execution.	Wallet onboarding, passport pages, approval gating, proof drawer, stream path, payment refusal, and OpenRouter-backed tool calls pass without using deterministic fallback.

On the current local benchmark machine (Node v22.22.2 on Apple M3), 10,000 iterations of two Ed25519 envelope verifications plus policy checks produced a mean latency of 0.1932 ms, median latency of 0.1827 ms, p95 latency of 0.2091 ms, and p99 latency of 0.2722 ms. Figure 2 visualizes the observed distribution summary. These numbers should be treated as implementation evidence, not a protocol limit.

Operation	Mean ms	Median ms	p95 ms	p99 ms
Two Ed25519 verifications + policy checks	0.1932	0.1827	0.2091	0.2722

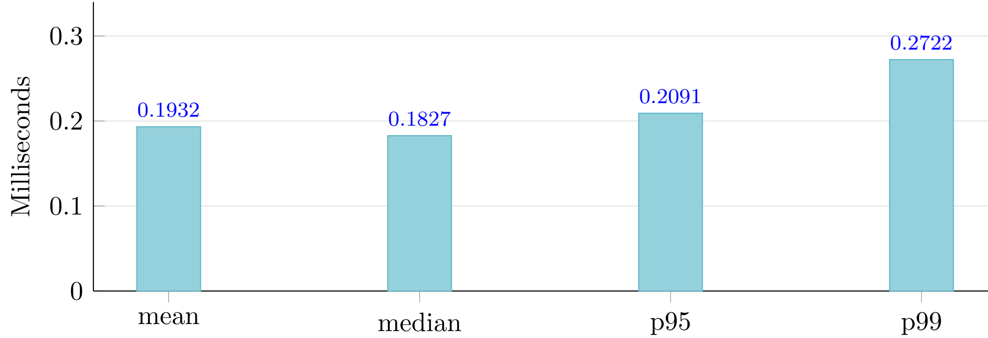

Figure 2: Verifier latency for 10,000 Node iterations of two Ed25519 envelope verifications plus policy checks on the local benchmark machine.

The same evidence run measured the canonical payload sizes shown below. Figure 3 emphasizes the privacy boundary: the public Solana anchor payload is substantially smaller than the private exchange bundle and carries only commitment material.

Object	Canonical bytes
Mandate envelope JSON	664
Bound token	179
Service metadata	109
Receipt envelope JSON	576
Solana anchor payload	409
Full private exchange bundle	2091

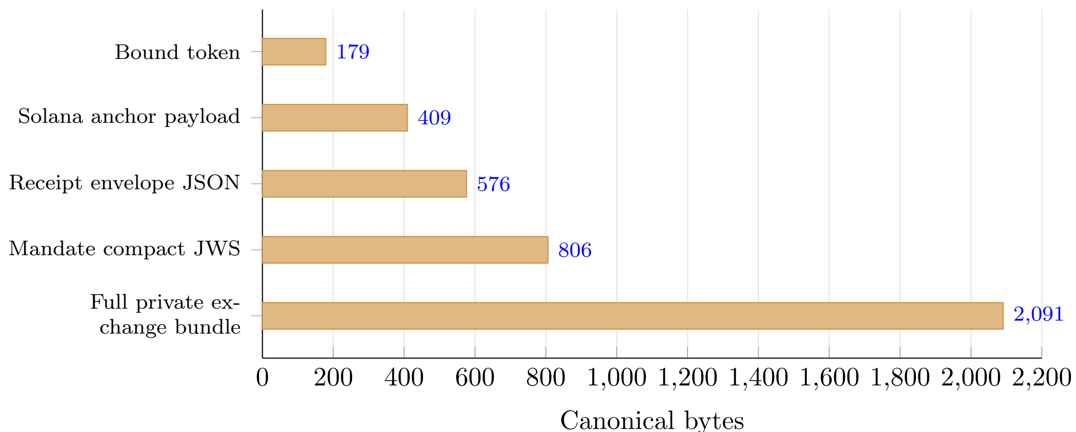

Figure 3: Canonical payload sizes. The public anchor is a compact commitment, while the full private exchange bundle remains off-chain.

18 Comparison to Adjacent Rails

Mandate is best read as a composition profile. It should not compete with the standards it uses. Table 3 summarizes the boundary.

Table 3: Comparison between Mandate and adjacent identity, authorization, payment, and registry rails.

Rail	What it provides	Mandate gap it leaves
OAuth bearer token	Scoped API access.	Stolen bearer replay and lack of user-mandate semantics.
DPoP / mTLS	Proof-of-possession token use.	Does not itself express who delegated what action to the agent.
W3C VC / DID	Portable signed claims and key discovery.	Does not define per-action runtime authorization or receipts.
MCP authorization	OAuth-based authorization for tools and resources.	Does not define a commerce mandate or bilateral service verification profile.
A2A	Agent discovery and task communication.	Requires a delegation and proof layer for high-risk actions.
AP2	Payment mandates and transaction binding.	Payment-specific; Mandate generalizes the mandate pattern to non-payment agent actions and receipts.
ACP delegated payment	Scoped payment credentials for merchant checkout.	Payment credential flow, not a full agent-to-agent authorization chain.
x402	HTTP payment and settlement pattern.	Payment trigger, not agent identity, delegation, or receipt audit.
ERC-8004 / ERC-8183	Optional public registry, reputation, escrow, and evaluator rails.	Public coordination layer, not private runtime authorization.

19 Related Work

Mandate composes OAuth security guidance, proof-of-possession tokens, verifiable credentials, DIDs, MCP authorization, A2A service metadata, and emerging agent payment protocols [12, 8, 4, 20, 19, 14, 1, 2, 3, 21]. It differs from pure identity registries by centering the signed user mandate and bilateral verification path.

AP2's mandate framework is the closest adjacent work because it introduces cryptographically constrained user authorization for agent payments [2]. Mandate borrows the core lesson that an agent should carry evidence of human authorization, but applies it beyond payments and adds a service-verification and receipt layer. ACP delegated payments and Stripe Shared Payment Tokens show that mainstream payment systems are moving toward constrained agent payment credentials [3, 18]. Mandate treats those credentials as adapters below the mandate layer.

Recent security papers make the need for runtime checks explicit. Work on AP2 runtime verification emphasizes context binding, consume-once semantics, and execution-time verification [11]. Red-team work against AP2-style shopping agents demonstrates that prompt injection can still manipulate agent behavior even when payment protocols include cryptographic mandates [6]. Secure

MCP proposals similarly motivate mutual authentication, fine-grained policy, and audit logging for tool-access systems [9]. These results support Mandate's design choice: cryptographic delegation is necessary, but it must be paired with runtime policy, prompt-injection defenses, and operational monitoring.

20 Limitations

Mandate does not solve prompt injection, economic manipulation, model alignment, issuer governance, key custody, or legal compliance. It also does not remove the need for explicit user consent at high-risk steps such as final payment, credential issuance, or irreversible actions. The profile assumes verifiers can maintain nonce state and revocation state with acceptable freshness. Open agent marketplaces also need governance for issuer trust, reputation abuse, dispute handling, and abuse reporting.

The current implementation is pre-alpha. Its evidence supports feasibility of the object model, cross-language hash determinism, a deployable browser proof flow, and low local verification latency. It is not a production security audit, and the benchmark is not a claim about all runtimes, hardware, wallets, RPC providers, model providers, or policy engines. Full public devnet NFT verification also depends on faucet-funded devnet SOL; the reference verifier is prepared, but public faucet availability is an external operational dependency rather than a protocol property.

21 Data and Code Availability

The reference implementation, website demo, paper source, and test vectors are maintained at `https://github.com/rishabhsai/mandate-protocol`. The deployed app artifact is available at `https://www.mandate.lol/app`. The arXiv source package contains the TeX source, bibliography, and processed BibTeX file required to rebuild the paper. The reproducible evidence commands are:

```
npm run test:vectors
npm run paper:evidence
```

The checked-in fixtures are public test material only. They must not be reused as live signing keys or production credentials.

22 Conclusion

Agent commerce needs a cryptographic record of delegated authority that is narrow, inspectable, revocable, and auditable. Mandate proposes a profile that makes each authorization decision explicit: who the agent is, who authorized it, what it may do, which service it may address, whether the token presenter owns the key, and what receipt proves the result. Blockchain anchoring can strengthen auditability, but it should remain a public commitment layer rather than the runtime authorization layer.

A Reference API Surface

The JavaScript core package exposes deterministic helpers for building the evidence chain:

- `createMandate`: constructs a versioned `UserMandate` object with id, issued time, expiry, audience, action, constraints, and nonce.

- `signEnvelope` and `verifyEnvelopeSignature`: produce and verify Ed25519 JWS-like envelopes over canonical JSON.
- `hashObject`: hashes canonical JSON objects for mandate, receipt, and anchor binding.
- `createReceiptAnchor`: constructs the public anchor payload from private mandate and receipt hashes.
- `verifyPolicy`: evaluates credential status, key binding, audience binding, scope binding, service action support, nonce replay, and validity windows.

The Python package mirrors the same primitive set so that shared test vectors can fail closed when canonicalization or hash behavior diverges across runtimes. The Solana package is deliberately narrower: it creates memo instructions and transactions for receipt-anchor payloads, but it does not authorize actions or inspect private mandates.

References

[1] A2A Protocol. Agent2agent protocol specification. https://a2a-protocol.org/latest/specification/, 2026.

[2] Agent Payments Protocol. Agent authorization framework. https://ap2-protocol.org/ap2/agent_authorization/, 2026.

[3] Agentic Commerce Protocol. Delegated payment spec. https://agentic-commerce-protocol.com/docs/commerce/specs/payment, 2026.

[4] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. Oauth 2.0 mutual-tls client authentication and certificate-bound access tokens. https://www.rfc-editor.org/rfc/rfc8705, 2020.

[5] Davide Crapis, Bryan Lim, Weixiong Tay, and Zuhwa Chooi. Erc-8183: Agentic commerce. https://eips.ethereum.org/EIPS/eip-8183, 2026.

[6] Tanusree Debi and Wentian Zhu. Whispers of wealth: Red-teaming google's agent payments protocol via prompt injection. https://arxiv.org/abs/2601.22569, 2026.

[7] Ethereum Improvement Proposals. Erc-8004: Trustless agents. https://eips.ethereum.org/EIPS/eip-8004, 2025.

[8] Daniel Fett, Brian Campbell, John Bradley, Torsten Lodderstedt, Michael Jones, and David Waite. Oauth 2.0 demonstrating proof of possession (dpop). https://www.rfc-editor.org/rfc/rfc9449, 2023.

[9] Xinyi Hou, Shenao Wang, Yifan Zhang, Ziluo Xue, Yanjie Zhao, Cai Fu, and Haoyu Wang. Smcp: Secure model context protocol. https://arxiv.org/abs/2602.01129, 2026.

[10] Michael Jones, Brian Campbell, Aaron Parecki, David Waite, and Werner Schwenkschuster. Oauth 2.0 protected resource metadata. https://www.rfc-editor.org/rfc/rfc9728, 2025.

[11] Qianlong Lan, Anuj Kaul, Shaun Jones, and Stephanie Westrum. Zero-trust runtime verification for agentic payment protocols: Mitigating replay and context-binding failures in ap2. https://arxiv.org/abs/2602.06345, 2026.

[12] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. Best current practice for oauth 2.0 security. https://www.rfc-editor.org/rfc/rfc9700, 2024.

[13] Qian'ang Mao, Jiaxin Wang, Ya Liu, Li Zhu, Cong Ma, and Jiaqi Yan. Sok: Security of autonomous llm agents in agentic commerce. https://arxiv.org/abs/2604.15367, 2026.

[14] Model Context Protocol. Model context protocol authorization specification. https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization, 2025.

[15] NIST National Cybersecurity Center of Excellence. Accelerating the adoption of software and ai agent identity and authorization. https://www.nccoe.nist.gov/publications/other/accelerating-adoption-software-and-ai-agent-identity-and-authorization-concept, 2026.

[16] Solana. Solana transaction and instruction documentation. https://solana.com/docs/core/transactions, 2026.

[17] Solana Program Library. Solana program library memo program. https://github.com/solana-program/memo, 2026.

[18] Stripe. Agentic commerce. https://docs.stripe.com/agentic-commerce, 2026.

[19] World Wide Web Consortium. Decentralized identifiers (dids) v1.0. https://www.w3.org/TR/did-1.0/, 2022.

[20] World Wide Web Consortium. Verifiable credentials data model v2.0. https://www.w3.org/TR/vc-data-model-2.0/, 2025.

[21] x402 Foundation. x402 protocol. https://www.x402.org/, 2025.