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

D. Borthwick
InsumerAPI

13 May 2026

Wallet State Attestation: Signed Booleans about On-Chain State
draft-borthwick-wallet-state-attestation-00

Abstract

This document defines Wallet State Attestation: a primitive in which an issuer reads on-chain state for a wallet address, evaluates one or more operator-defined conditions, and returns a cryptographically signed boolean (or structured fact profile) that any verifier can check offline using a published JWKS endpoint. The primitive enables condition-based access decisions without identity presentation, credential exchange, or trust in the issuer at runtime. This document describes the request and response surface, the signing algorithm posture, the JWKS discovery pattern, and the security and privacy considerations of the primitive. It does not define a new wire protocol; rather, it profiles existing IETF building blocks (JWT, JWKS, JOSE) for the wallet-state-attestation use case.

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

1.1. Motivation

Existing access-control primitives --- OAuth 2.0 [RFC6749], OIDC [OIDC-CORE], SAML, and the broader credential-presentation family --- answer the question "who is this user?" by accepting a presented artifact (token, assertion, credential) from a holder.

Wallet State Attestation answers a different question: "does this wallet currently satisfy a specified condition over on-chain state?" The holder presents nothing. An issuer reads canonical on-chain state for a specified wallet address, evaluates the operator-defined condition, and returns a signed boolean. Verifiers check the signature offline using a published JWKS endpoint.

This is a distinct primitive from identity verification, credential issuance, or holder-presentation flows. It belongs in a parallel category --- what this document terms wallet-state-attestation --- alongside, not within, the existing credential-presentation stack.

1.2. Position Relative to Identity Primitives

OAuth and OIDC prove who a subject is. Wallet State Attestation proves what a wallet currently holds, owns, or has been attested to in on-chain state.

These are complementary, not competing. A system may use OAuth to authenticate a human or service principal, then call a wallet-state attestation issuer to verify that a wallet associated with that principal satisfies an on-chain condition. The two primitives operate on different subjects (identity vs. wallet state) and answer different questions.

1.3. Position Relative to Credentials

Wallet State Attestation is *not* a Verifiable Credential (W3C VC) [VC-DATA-MODEL], an mDoc / ISO/IEC 18013-5 mobile document, an eIDAS attestation, a SAML assertion, or an OIDC ID token. These are credential-presentation primitives: an authority issues a credential, the holder presents it, the verifier checks the issuer’s signature over the holder’s presented bytes.

Wallet State Attestation does not involve credential presentation. The holder does not present anything. The issuer pulls public on-chain state directly, evaluates a condition, and signs the result. The signed payload is an observation about external state, not a credential about the holder’s identity.

This document treats the credential / VC / eIDAS / SAML stack as adjacent prior art for explicit non-conflation purposes. See Section 12 ("What This Is Not") for the full enumeration.

1.4. Position Relative to Other Documents

Wallet State Attestation is referenced or adopted in the following public artifacts (all informative; see References):

* *Knowledge Context Protocol (KCP) RFC-0004* [KCP-0004] --- defines an attestation_url / attestation_jwks mechanism in KCP YAML manifests for verifying agent eligibility before access. The pattern is mechanism-agnostic; wallet state attestation is one valid implementation shape.

* *Verifiable Intent (VI) environment.* constraint family* [VI-ENVIRONMENT] --- defines an environmental constraint family for autonomous-mode (L3) execution in agent commerce, establishing family-wide vocabulary (attestation_url, max_attestation_age), composition rules, and IANA registry mechanics for individual constraint types. The wallet-state-attestation primitive is the reference implementation shape used by VI's environment.wallet_state constraint type.

* *Agent Governance Vocabulary* [AGV] --- defines wallet_state as a foundation-layer signal type with named production issuers.

* *Open Agent Trust Registry (OATR)* [OATR] --- third-party trust registry of attestation issuers, including wallet-state-attestation issuers.

* *Draft ERC-8210 Agent Assurance Protocol* [ERC-8210] --- proposed IRiskHook verification taxonomy identifies wallet state attestation as the representative shape for pre-commitment eligibility verification (one of three categories in the verification framework, alongside post-hoc resolution evidence and behavioral reputation).

* *Universal Commerce Protocol (UCP) identity-linking* [UCP-IDENTITY] --- wallet attestation acknowledged as a reserved extensibility point for future non-OAuth identity-mechanism types.

Wallet state attestation is *not* dependent on or normatively related to any of these documents. They are listed as evidence of adoption and as informative context for readers seeking to understand where the primitive is in use.

1.5. Use Cases

Common deployments of wallet state attestation include:

* *Condition-based access* --- gating API access, content access, or commerce flows on whether a specified wallet holds a specified token, NFT, or attested status at request time, without exposing actual balance or transaction history.

* *Agent-to-agent trust verification* --- autonomous agents verifying counterparty wallet state (holdings, attested compliance status, etc.) before transaction execution.

* *Compliance gating* --- verifying on-chain compliance attestations (KYC status from external attestors, country attestations, sanctions clearance) against a wallet without handling personally-identifiable information.

* *Pre-commitment eligibility* --- agent commerce protocols verifying that an agent wallet satisfies pre-conditions (sufficient stake, sufficient balance, attested status) before contract commitment.

The primitive is read-only and does not interact with chain state beyond observation. Implementations issue attestations on demand; attestations are short-lived and expire (typically thirty minutes; see Section 5).

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:

Issuer --- The party that reads on-chain state, evaluates conditions, and signs the resulting attestation. The issuer is the sole signer of attestations within its declared scope (see Section 10.1).

Holder --- The wallet whose on-chain state is observed. The holder does not present anything in this primitive; the issuer reads chain state directly. The holder need not be aware of any individual attestation.

Verifier --- The party that consumes the attestation, checks the signature against the issuer’s published JWKS, and acts on the result.

Condition --- An operator-defined predicate over wallet state. Examples include: "balance of token T at address A is greater than or equal to value V", "wallet A owns NFT N", "wallet A has a valid on-chain attestation of type T from attestor X".

Attestation --- The signed payload returned by the issuer in response to a request. Distinguish explicitly from a credential (which is presented by a holder), from a JWT claim in the credential sense (which carries identity information), and from a VC (which is issued by an authority over a holder’s presented identity). An attestation in this document’s sense is an issuer-signed observation about public on-chain state.

Fact profile --- A multi-condition, multi-dimension attestation payload consisting of a structured collection of independently-signed condition results. Used for richer wallet-state queries where a single boolean is insufficient. Distinguish explicitly from a "trust score" or "reputation rating" --- a fact profile contains observed evidence organized by dimension, not an opinion or aggregate.

Block reference --- Chain-specific freshness evidence included in every attestation. EVM chains MUST include block number and block timestamp. XRPL MUST include ledger index and ledger hash. Solana MUST include slot. Bitcoin MUST include block height and block hash (chain-tip evidence; see Section 9 for Bitcoin-specific considerations).

Condition hash --- A SHA-256 hash of the JSON serialization of the evaluated condition, with top-level keys sorted in lexical order, represented as a 0x-prefixed lowercase hexadecimal string, included in the response for tamper-evidence over the condition itself.

3. Architecture Overview

The primitive operates as a three-step pipeline: *read -> evaluate -> sign*.

1. The issuer receives a request specifying a wallet address, a chain identifier, and one or more conditions.

2. The issuer reads canonical on-chain state for the specified wallet at the current chain tip (or, where supported, at a specified block).

3. The issuer evaluates each condition against the observed state.

4. The issuer signs the result and returns the attestation.

The attestation is *deterministic*: the same inputs evaluated against the same chain state at the same block produce the same output. A verifier can independently re-derive the attestation by reading the same chain state and applying the same condition logic.

The primitive is *read-only*: the issuer does not modify chain state, does not request a wallet signature from the holder, and does not custody any assets. The issuer is a passive reader of public chain state.

For each chain referenced in the request, the issuer captures a chain-tip reference (block number and block timestamp for EVM chains; ledger index and, where available, ledger hash for XRPL; slot for Solana; block height and block hash for Bitcoin) at attestation time. The chain-tip reference is carried inside each per-condition result and is therefore covered by the issuer's signature. If a required chain observation cannot be acquired, the issuer MUST NOT sign the attestation; see Section 10.3.

This document does not specify how the issuer reads chain state (RPC routing, indexer choice, archive node selection), how it manages signing keys (HSM, software, federated), or how it handles chain-specific edge cases (reorgs, finality, data availability). These are implementation concerns, deliberately left out of scope.

4. Request Shape

A request is a JSON object with the following structure:

```
{
  "wallet": "<address>",
  "chainId": "<chain identifier>",
  "conditions": [
    {
      "type": "<condition type>",
      "...": "<condition-specific parameters>"
    }
  ],
  "options": {
    "format": "json",
    "proof": "none"
  }
}
```

wallet --- A chain-specific address string. EVM addresses use the standard 0x-prefixed hexadecimal form. Solana addresses use base58. XRPL addresses use the r-address format. Bitcoin addresses MAY use any standard format (P2PKH, P2SH, bech32 / SegWit, bech32m / Taproot). Implementations MAY accept the wallet in a chain-specific field (e.g., solanaWallet, xrplWallet, bitcoinWallet) when the address format is ambiguous.

chainId --- A chain identifier. EVM chains use a numeric chain identifier consistent with CAIP-2 [CAIP-2], expressible as integer (e.g., 1 for Ethereum mainnet) or string. Non-EVM chains use a string identifier (solana, xrpl, bitcoin).

conditions --- A list of one or more condition objects. Each has a type field discriminating the condition category, and type-specific fields specifying the parameters of that category. This document does not enumerate all possible condition types. Common categories include token balance evaluation, NFT ownership evaluation, on-chain attestation reference (e.g., reference to an Ethereum Attestation Service entry), and identity-registry membership evaluation. Implementations MAY define additional condition types.

options.format --- Selects the response envelope. Values: "json" (default; raw JSON object) or "jwt" (RFC 7519 JWT token).

options.proof --- Requests an optional cryptographic proof layer. Values: "none" (default) or "merkle" (where supported by the underlying chain; returns [EIP-1186] Merkle storage proofs anchored to the block header).

5. Response Shape

A response contains a signed attestation object together with the signature and key identifier. The signed attestation object has the following structure:

```
{
  "id": "<opaque attestation identifier>",
  "pass": true,
  "results": [
    {
      "condition": 0,
      "label": "<operator-provided label>",
      "type": "<condition type>",
      "chainId": "<chain identifier>",
      "met": true,
      "evaluatedCondition": { /* condition object */ },
      "conditionHash": "0x<sha256 of evaluatedCondition>",
      "blockNumber": "<chain-specific block reference>",
      "blockTimestamp": "<ISO 8601 timestamp>"
    }
  ],
  "attestedAt": "<ISO 8601 timestamp>"
}
```

The signature is computed over a JSON serialization of this object with field order id, pass, results, attestedAt. The signature, the key identifier (kid), expiry timestamp (expiresAt), and any deployment-specific wrappers are transmitted alongside the signed object but are NOT part of the signed bytes. A typical wire response carries the signed object plus these adjacent fields:

```
{
  "attestation": {
    /* signed object above */
    "expiresAt": "<ISO 8601 timestamp>"
  },
  "sig": "<base64 P1363 ECDSA signature>",
  "kid": "<key identifier for JWKS lookup>"
}
```

Field semantics (signed object):

id --- An opaque attestation identifier. Implementation-specific format. Useful for correlation and logging; not used for verification.

pass --- The aggregate result. Default semantics: logical AND of all per-condition met values. Operators MAY specify alternate combination logic in extensions.

results --- An array of per-condition result objects. Each MUST include the per-condition met boolean, the condition type, the verbatim evaluatedCondition (as it was evaluated, after any

normalization), and the conditionHash (SHA-256 of the JSON serialization of evaluatedCondition with top-level keys sorted lexically). Each result MUST include chain-specific freshness evidence: blockNumber and blockTimestamp for EVM chains; ledgerIndex and (where available) ledgerHash for XRPL; slot for Solana; blockHeight and blockHash for Bitcoin. All such freshness fields are inside the signed results array and are therefore covered by the issuer's signature. Implementations MAY include additional informational fields (condition index, label, etc.) echoed from the request.

attestedAt --- ISO 8601 timestamp of when the attestation was signed.

Field semantics (adjacent, unsigned):

expiresAt --- ISO 8601 timestamp of when the attestation expires. Verifiers MUST check expiry. Recommended default: thirty minutes after attestedAt. Implementations MAY use shorter expiries; longer expiries SHOULD be justified by the use case. expiresAt is transmitted with the attestation but is not part of the signed payload; verifiers concerned with tamper-evidence over the expiry SHOULD use the JWT format (Section 8), where exp is a signed claim.

kid --- Key identifier (RFC 7517) for the public key used to sign this attestation. Verifiers use this to look up the appropriate public key in the issuer's JWKS. kid is transmitted alongside the signed payload but is not part of the signed bytes; verifiers identify the verification key by trying the issuer's published JWKS entries indexed by the transmitted kid. The JWT format embeds kid in the protected header where it is covered by the JWT signature.

sig --- The signature. Default: ECDSA P-256 signature over the JSON serialization of the signed object, in base64-encoded P1363 (raw R || S) format. See Section 6.

The signature scope includes the evaluatedCondition and conditionHash fields nested inside results, ensuring the signature is tamper-evident over both the result and the condition that produced it.

6. Signing Algorithm

The default signing algorithm is *ECDSA with P-256 curve and SHA-256 (ES256)* per [RFC7518].

The signed bytes are the JSON serialization of the signed object (id, pass, results, attestedAt) using deterministic field-order serialization in that order. Verifiers MUST verify the signature

against the exact bytes the issuer produced; verifiers reconstructing the preimage from a parsed JSON object MUST preserve the original field order. Issuers and verifiers SHOULD treat the signed bytes as opaque transport-layer material, not a re-serializable JSON object. For nested objects whose key order is not externally meaningful (notably the evaluatedCondition object hashed into conditionHash), keys are sorted in lexical order before serialization. See [RFC8785] for a general canonical-JSON scheme; this document does not require RFC 8785 conformance for the outer signed payload, because the outer field set is fixed and ordered by this specification.

The signature output is the raw P1363 (R || S) representation of the ECDSA signature, base64-encoded (88 characters for P-256). Implementations producing DER-encoded ECDSA signatures MUST convert to P1363 before transmission.

Implementations MAY support additional algorithms (ES384, EdDSA / Ed25519). Algorithm selection is encoded in the JWKS entry indexed by kid. Verifiers MUST consult the JWKS entry to determine the algorithm before verifying.

7. JWKS Discovery

The issuer publishes its public key set at a discoverable HTTPS endpoint following [RFC7517]. The recommended path is /.well-known/jwks.json per [RFC5785].

Each JWKS entry MUST include kid, kty, crv, alg, x, and y fields appropriate for the curve and algorithm. Issuers MAY publish multiple active keys simultaneously to support key rotation.

Key rotation: when the issuer rotates signing keys, the new public key MUST be published in JWKS before the issuer signs any attestation with the corresponding private key. Retired public keys SHOULD remain published in JWKS for a duration at least equal to the longest possible outstanding attestation expiry (typically the default thirty-minute window).

Verifiers MUST cache the JWKS appropriately (HTTP cache headers as published by the issuer; the issuer SHOULD set Cache-Control: max-age=3600 or shorter).

8. JWT Format

When options.format is "jwt", the response is wrapped in an RFC 7519 JWT.

Protected header: {"alg": "ES256", "typ": "JWT", "kid": "<key id>"} per [RFC7519]. The kid in the protected header is covered by the JWT signature.

Standard claims:

* iss --- issuer identifier (e.g., the issuer's base URL)

* sub --- wallet address (the subject of the attestation)

* jti --- JWT identifier (typically the attestation id)

* iat --- issuance timestamp (Unix seconds)

* exp --- expiry timestamp (Unix seconds)

Custom claims carrying the attestation payload:

* pass --- aggregate result

* results --- per-condition result array

* conditionHash --- array of per-condition conditionHash values, in results order

* blockNumber --- chain block reference (typically the first result's blockNumber, where chain-homogeneous)

* blockTimestamp --- chain block timestamp (typically the first result's blockTimestamp)

The JWT signature is computed by the issuer using the corresponding private key. Verifiers verify using any standard JWT library that supports ES256 and JWKS discovery.

The JWT format provides signed coverage of exp and kid (via standard claim and protected header, respectively); deployments requiring tamper-evident expiry or signed key-identifier binding SHOULD use the JWT format rather than the JSON format described in Section 5.

9. Optional Merkle Proofs

When options.proof is "merkle" and the underlying chain supports Merkle storage proofs (typically EVM chains for storage-slot-mappable conditions), the response MAY include [EIP-1186] Merkle storage proofs anchored to the block header for each per-condition result.

Merkle proofs permit trustless verification: a verifier can independently check the Merkle proof against the block reference without trusting the issuer's evaluation. When a Merkle proof is included (proof.available is true in the per-result proof object), the proof object MUST include the block reference and the storage slot path.

Not all conditions or chains support Merkle proofs. Implementations MUST indicate proof availability per result and MAY return proof.available: false with a reason field when proof generation is unsupported for the condition or chain (e.g., NFT ownership conditions, non-storage-slot-mappable conditions, non-EVM chains).

Privacy trade-off: Merkle proofs reveal raw on-chain values (e.g., the actual token balance). The default "none" mode preserves privacy by returning only the boolean result. Operators MUST consider whether the trustlessness benefit of Merkle proofs is worth the privacy cost in each deployment.

Bitcoin, XRPL, and Solana have different state-proof models and may not support Merkle proofs in the [EIP-1186] sense. Implementations targeting those chains MAY define chain-appropriate proof shapes or MAY return proof.available: false.

For Bitcoin specifically: the UTXO model does not permit reproducible balance-at-block queries. Bitcoin implementations anchor the attestation to the chain tip (block height + block hash) at observation time as the cryptographically-bound freshness evidence appropriate to the UTXO model. The blockHeight and blockHash are covered by the issuer's signature inside the result and bind the observation to a recent chain tip rather than to a specific block's historical state. Issuers MUST refuse to sign Bitcoin attestations when the chain-tip lookup fails; see Section 10.3.

10. Security Considerations

10.1. Single-Issuer Architecture

This document describes a single-issuer attestation primitive. The issuer is the sole signer of attestations within its declared scope. This is a deliberate design choice for several reasons:

1. *Deterministic signature surface.* Verifiers can independently re-derive an attestation from chain state without resolving issuer trust hierarchies, federation policies, or multi-party signing logic.

2. *No re-signing or wrapping.* Attestations cannot be re-signed by intermediaries, aggregated under a wrapper signature, or repackaged in derivative forms. The original observation's signature is the authoritative artifact end-to-end.

3. *Unambiguous attribution.* For audit, dispute resolution, and accountability, the issuer is unambiguously identified by the JWKS endpoint and signing key.

Multi-issuer federation, key delegation, cross-issuer attestation aggregation, and threshold signing are explicitly out of scope for this document. Future Internet-Drafts MAY define such mechanisms as separate primitives, but they are distinct from the single-issuer wallet-state attestation defined here.

10.2. Freshness Anchors and Replay Considerations

The cryptographically-bound freshness anchors are the block reference inside each per-condition result and the attestedAt timestamp in the outer signed payload. Both are covered by the issuer's signature. Verifiers requiring strict tamper-evident freshness derive their policy from these signed anchors (e.g., "reject if attestedAt is older than thirty minutes", or "reject if blockTimestamp is older than sixty seconds").

The expiresAt field is an issuer-provided TTL hint, transmitted alongside the attestation but outside the signed payload in the JSON format described in Section 5. Verifiers MAY honor expiresAt directly as a convenience. Deployments requiring tamper-evident expiry SHOULD use the JWT format described in Section 8, where exp is a signed claim.

For replay-sensitive deployments, implementations MAY include a nonce binding or audience binding in custom JWT claims (when the JWT format is used). This document does not normatively specify these extensions; deployments requiring replay protection beyond freshness checking SHOULD define them explicitly in their integration profile.

10.3. Refuse-on-Observation-Failure

Issuers MUST refuse to sign an attestation when any required observation fails: chain-tip lookup, balance query, on-chain attestation registry lookup, or any other data source the condition depends on. A successfully signed attestation therefore implies that all underlying observations were acquired against a current chain reference at attestation time. Partial or degraded observations MUST NOT be signed.

Verifiers MAY cross-check the included block reference against an independent chain source if the deployment requires it.

10.4. What This Primitive Does Not Protect Against

* *Chain-level reorganizations after attestation.* If a chain reorgs after an attestation has been signed, the historical state observed in the attestation may no longer be canonical. Operators handling this concern SHOULD use chains with strong finality, SHOULD use sufficiently old block references, or SHOULD include reorg-handling logic in their verification flow.

* *Off-chain state changes.* This primitive observes on-chain state only. It does not attest to off-chain identity, off-chain agreements, or off-chain commitments.

* *Malicious issuer.* A compromised or malicious issuer could sign false attestations. This is mitigated by: (a) JWKS publication of signing keys, (b) optional Merkle proofs for trustless verification (where supported), (c) condition hashes for tamper-evidence over the evaluated logic, and (d) deterministic re-derivation by independent verifiers.

10.5. Algorithm Agility

The default ES256 algorithm is widely supported and provides 128-bit security. Implementations supporting additional algorithms MUST include the algorithm in the JWKS entry indexed by kid. Verifiers MUST check the algorithm before verifying.

Algorithm migration: when the cryptographic landscape changes (e.g., post-quantum migration), issuers can rotate to new algorithms via JWKS without breaking the verification protocol. The protocol surface remains stable.

11. Privacy Considerations

11.1. Boolean by Default

The default response mode returns only the result of condition evaluation, never the underlying on-chain value. A verifier learns whether a wallet satisfies a condition (e.g., "holds at least 100 USDC") but does not learn the actual balance.

This is the privacy-preserving default and SHOULD be used unless specific deployment requirements justify a less-private mode.

11.2. Merkle Mode Trade-Off

When options.proof is "merkle", raw on-chain values are revealed for trustless verification. Operators MUST consider the privacy implications of revealing actual balances or asset holdings in each deployment.

11.3. No PII

The primitive operates on public on-chain state only. No personally-identifiable information is collected, processed, signed, or transmitted.

11.4. Wallet Address Handling

The wallet address is the subject of the query but is public information on the underlying chain. Issuers SHOULD NOT log or retain wallet addresses beyond what is required for service delivery, abuse prevention, and operational integrity. Implementations targeting privacy-conscious deployments SHOULD document their address-handling policy.

11.5. Awareness of Observation

This primitive observes public chain state without holder participation. The holder is not required to consent to nor be aware of any individual attestation, consistent with the public nature of blockchain state. Deployers operating in regulated or consumer-facing contexts SHOULD consider whether their deployment context requires additional transparency mechanisms; such mechanisms are out of scope for this document.

12. What This Is Not

This section explicitly enumerates adjacent primitives that wallet state attestation is *not*, to prevent conflation:

* *Not a Verifiable Credential (W3C VC) [VC-DATA-MODEL].* VCs are issued by an authority, presented by a holder, and verified against the holder’s presented bytes. Wallet state attestations are pulled by the issuer from public chain state without holder presentation.

* *Not an mDoc or eIDAS attestation.* mDocs (ISO/IEC 18013-5) are pre-issued, batched, and stored by the holder for later presentation. Wallet state attestations are issued on demand, are short-lived, and are not stored by the holder.

* *Not a SAML assertion or OIDC ID token.* These primitives carry identity claims about a user. Wallet state attestations carry observations about wallet state, with no identity component.

* *Not a reputation score, trust score, or credit rating.* A wallet state attestation contains observed facts (boolean results, optionally with structured fact profiles). It does not contain opinions, scores, ratings, or aggregated judgments. A separate primitive built on top of wallet state attestations might produce a score; this primitive does not.

* *Not a settlement attestation, delivery attestation, or transaction proof.* Wallet state attestation is read-only state observation at observation time. It does not attest to action confirmation, transaction completion, or delivery of goods or services.

* *Not an oracle in the price-feed sense.* This primitive observes specific wallet state on demand for a specific request; it does not publish continuous data streams or aggregate market data.

These distinctions are essential to keeping wallet state attestation as a distinct primitive in the agent-commerce, condition-based-access, and verification ecosystems.

13. IANA Considerations

This document has no IANA actions.

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/rfc/rfc5785>.

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

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

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/rfc/rfc7519>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

14.2. Informative References

[AGV] "Agent Governance Vocabulary", n.d., <https://github.com/aeoess/agent-governance-vocabulary>.

[CAIP-2] "Chain Agnostic Improvement Proposal 2: Blockchain ID Specification", n.d., <https://github.com/ChainAgnostic/CAIPs/blob/main/CAIPs/caip-2.md>.

[EIP-1186] "Ethereum Improvement Proposal 1186: RPC-Method to get Merkle Proofs", n.d., <https://eips.ethereum.org/EIPS/eip-1186>.

[ERC-8210] "Draft Ethereum Improvement Proposal 8210: Agent Assurance Protocol", n.d., <https://ethereum-magicians.org/t/erc-8210-agent-assurance/28097>.

[KCP-0004] "Knowledge Context Protocol RFC-0004: Trust, Provenance, and Compliance", n.d., <https://github.com/Cantara/knowledge-context-protocol/blob/main/RFC-0004-Trust-and-Compliance.md>.

[OATR] "Open Agent Trust Registry", n.d., <https://github.com/FransDevelopment/open-agent-trust-registry>.

[OIDC-CORE] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", n.d., <https://openid.net/specs/openid-connect-core-1_0.html>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/rfc/rfc6749>.

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

[UCP-IDENTITY] "Universal Commerce Protocol: Identity Linking", n.d., <https://github.com/Universal-Commerce-Protocol/ucp/blob/main/docs/specification/identity-linking.md>.

[VC-DATA-MODEL] Sporny, M., "Verifiable Credentials Data Model", n.d., <https://www.w3.org/TR/vc-data-model-2.0/>.

[VI-ENVIRONMENT] Borthwick, D. and M. Msebenzi, "Verifiable Intent --- environment.* Constraint Family", May 2026, <https://datatracker.ietf.org/doc/draft-borthwick-msebenzi-environment-state/>.

Acknowledgments

The author thanks the IETF community for ongoing discussion of attestation primitives in the agentic and wallet-state space.

Author’s Address

Douglas Borthwick
InsumerAPI
New Canaan, CT
United States of America
Email: douglas@insumermodel.com