# Universal Manifest v0.5 — EXT-T1

## Tier-1 Binding Profile

**v0.5.0. Published as tag v0.5.0 of UniversalManifest/universal-manifest, with the same wire files as release candidate 1 (v0.5.0-rc.1). It is a mechanism release, not a completed implementation or conformance qualification. Immutable versions in UniversalManifest/universal-manifest are normative; main is the working version. Earlier versions were reviewed in the OMA3 Technical Working Group, and publication asserts no OMA3 ratification.**

## 8 November 2026


**This document:** /spec/v0.5/ext-t1/

**Document set:** Universal Manifest v0.5.0 · What changed in v0.5.0

**Artifacts:** JSON Schema · JSON-LD context

**History:** https://universalmanifest.net/spec/ (version index and changelog)



**Author:** Grig Bilham — inventor, architect, and author of the Universal Manifest.



Copyright © 2026 SumSet Tech LLC and the Universal Manifest contributors. This specification is available under the Apache License, Version 2.0. It is free and remains free. Earlier published revisions retain their original licence notices.


---

**v0.5.0.** Published as tag `v0.5.0` of `UniversalManifest/universal-manifest`, with the same wire files as release candidate 1 (`v0.5.0-rc.1`). Wire version `0.5`; namespace `https://universalmanifest.net/ns/v0.5`; reading routes `/spec/v0.5/`. It is a mechanism release, not a completed implementation or conformance qualification. Feature-specific PREVIEW and conditional requirements remain in force.

This is a **companion profile** to the Universal Manifest v0.5 Base specification. It specifies the credential-binding mechanics that turn the Base's **Tier-1 requirement** into running code. Read the Base first; this document assumes its terminology, envelope, and evaluation sequence.

# EXT-T1: Tier-1 Binding Profile

**Companion to:** Base specification (Tier-0 / Tier-1 Core). **Registry categories:** `binding`, `trust`.
**Status:** Production-candidate normative content (carried forward and extended from v0.3). The

Tier-1 binding core is not PREVIEW; the per-facet floor subsections **T1.3.1–T1.3.3 are PREVIEW** (their enforcement profiles may change before the schema locks).

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 [RFC 2119] [RFC 8174] when, and only when, they appear in ALL CAPITALS, as shown here.

## T1.0. Scope and Relationship to the Base

The Base specification *requires*, for any claim relied upon at Tier 1 or above, a verified `holderBinding` (Base §6.4.2, Base §4.2), and it carries the binding *field surface* on claims (Base §1.4.3) and on receipts (Base §3.3.1.1). This profile specifies the *mechanics* the Base names:

- **Holder binding** (T1.1) — proving the presenting subject is the credentialed holder, in three modes.
- **Presentation proof** (T1.2) — proof-of-possession binding a presentation to a specific verifier and moment.
- **Liveness attestation** (T1.3) — evidence of recent interactive human authentication.
- **Stage-2 verification sub-steps** (T1.4) — how the above weave into the Verify stage of the evaluation sequence.
- **The end-to-end claim-proof verification path** (T1.5) — the seven-step chain that ties outer-signature, claim-issuer, and attester key-authorization together, and the `claimProof` verification procedure.
- **Security considerations and backwards compatibility** (T1.6).

This profile is **OPTIONAL to implement, but REQUIRED for any implementation claiming Tier 1 or higher assurance** (Base §4.7). An evaluator that does not implement it records the binding statuses as `"absent"` and caps every relied-upon claim at Tier 0.

Non-goals: this profile does not define the Tier-2 cryptographic cross-DID proof (EXT-T2) or the Tier-3 ceremony (EXT-T3); it specifies the Tier-1 layer those tiers build upon.

## T1.1. Holder Binding

Manifests without `holderBinding` on their claims MUST NOT be treated as providing identity assurance above Tier 0, regardless of the trust tier declared by the holder. Every claim intended to

be relied upon at Tier 1 or above MUST carry `holderBinding`. **Exception — `identity.crossDidBinding` claims.** For a binding claim whose `boundDids` includes the manifest `subject` (Base §6.4.4), presenter binding is established by the subject-signature key-authorization check (T1.5.2 step 2) together with attester or proof verification (T1.4 sub-step 2d); a separate `holderBinding` member is OPTIONAL on such claims.

The `holderBinding` object supports three modes:

- `"sd-jwt-kb"` — SD-JWT Key Binding per [RFC9901]. The credential contains a `cnf` (confirmation) claim ([RFC7800]) with a JWK Thumbprint ([RFC7638]) binding the credential to a specific holder key. The `cnfThumbprint` field is REQUIRED when this mode is used. **This is the mandatory-to-implement baseline.**
- `"bbs-holder-commitment"` — BBS+ holder binding per the W3C Data Integrity BBS Cryptosuites. The issuer commits to a holder-private key during credential issuance; the holder derives a zero-knowledge proof demonstrating possession of the bound key without revealing it. The derived proof bytes do not link presentations, but other carried values can (Base §7.1). The `pseudonymScope` field is OPTIONAL; when present it enables scope-exclusive pseudonyms per verifier. RECOMMENDED for privacy-sensitive deployments.
- `"reciprocal-control"` — Direct DID binding. Both the manifest subject and the bound DID sign the same challenge (the manifest `@id`). The `boundDid`, `subjectProof`, and `boundDidProof` fields are REQUIRED.

### T1.1.1. Per-Mode Verification Procedure

For each mode, the evaluator performs the following verification during Stage-2 sub-step 2a (T1.4):

**`sd-jwt-kb`.**

1. Locate the SD-JWT and its Key Binding JWT (KB-JWT) in the claim's `claimProof`.
2. Confirm the credential's `cnf` JWK thumbprint ([RFC7638]) equals the declared `cnfThumbprint`.
3. Validate the KB-JWT signature with the confirmed holder key.
4. Confirm the KB-JWT `aud` equals the presentation `audience` and its `nonce` equals the presentation `challenge` (T1.2), and that its `sd_hash` covers the disclosed credential. When the same KB-JWT also serves as the manifest's `presentationProof`, its payload additionally carries the manifest-hash binding (`um_manifest_hash` — T1.2).
5. On any failure, record `holderBindingStatus: "failed"` and cap the claim at Tier 0.

**`bbs-holder-commitment`.** The binding carries the REQUIRED fields `commitment` (the issuer's commitment to the holder secret) and `proofValue` (the holder's zero-knowledge proof of possession), and the OPTIONAL `pseudonymScope` ([VC-DI-BBS]).

1. Verify the BBS+ derived `proofValue` demonstrates possession of the secret committed in `commitment`, against the issuer's BLS12-381 public key.
2. If `pseudonymScope` is present, confirm the pseudonym is bound to that scope (and not reused across verifiers — T1.6.1).
3. On failure, record `holderBindingStatus: "failed"` and cap the claim at Tier 0.

**`reciprocal-control`.** Confirm `subjectProof` and `boundDidProof` are each valid signatures over the manifest `@id` by the manifest `subject` key and the `boundDid` key respectively; on failure record `holderBindingStatus: "failed"`. (See the limitation in T1.6.3.)

Verifying an `sd-jwt-kb` binding presupposes an interactive presentation: step 4 compares the KB-JWT `aud` and `nonce` against the `audience` and `challenge` of the presentation (T1.2). In non-interactive contexts (offline or cached presentation), holder binding at Tier 1 or above is carried by the `bbs-holder-commitment` or `reciprocal-control` modes, which verify without a verifier-issued challenge.

The receipt field `holderBindingStatus` records the outcome as one of `"verified"`, `"failed"`, `"unsupported-mode"`, or `"absent"` (Base §3.3.1.1); `"absent"` indicates the claim carried no `holderBinding`.

## T1.2. Presentation Proof

The `presentationProof` field at the manifest root provides proof-of-possession at verification time, binding the manifest to a specific verifier and moment. It prevents manifest replay.

When presenting a manifest in response to a verifier-issued challenge (an *interactive presentation*), the presenter MUST include a `presentationProof` containing:

- `proofType` — One of `"sd-jwt-kb"`, `"bbs-derived"`, or `"did-auth"`.
- `challenge` — The verifier-issued nonce. MUST be unique per presentation request.
- `audience` — The verifier's identifier (DID or origin). MUST match the requesting verifier.
- `created` — RFC 3339 timestamp of proof generation.
- `proofValue` — Base64url-encoded proof (KB-JWT, BBS+ derived proof, or DID Auth signature).

`presentationProof` is excluded from the manifest signing input (removed alongside `signature` in step 2 of Base §1.6.3). Its `proofValue` MUST bind four values — the manifest's **signing-input hash** (SHA-256 of the Base §1.6.3 canonical bytes, as its 32 raw bytes), `challenge`, `audience`, and `created` — per the mechanics of the declared `proofType`:

- **`did-auth`** — `proofValue` is a DID Auth signature ([DID-CORE]) and MUST be computed over the concatenation of: the manifest's signing-input hash (its 32 raw bytes), then the UTF-8 bytes of `challenge`, `audience`, and `created`, in that order.
- **`sd-jwt-kb`** — `proofValue` carries the Key Binding JWT ([RFC9901]). A KB-JWT signs its own `BASE64URL(header).BASE64URL(payload)` input, so the binding lives in the KB-JWT payload rather than in a byte concatenation: the payload's `nonce` MUST equal `challenge`, its `aud` MUST equal `audience`, its `iat` MUST correspond to `created`, and the payload MUST additionally carry the manifest's signing-input hash in a `um_manifest_hash` claim (base64url of the 32 hash bytes). This is a KB-JWT-payload rule only; it defines no new Universal Manifest wire member.
- **`bbs-derived`** — `proofValue` is a BBS+ derived proof ([VC-DI-BBS]); the four values, in the order stated above, form the BBS presentation header the derived proof is bound to.

Binding the proof to the signing-input hash ties the presentation to the exact signed manifest without requiring the holder to re-sign per presentation.

A manifest presented without a `presentationProof` is valid only for offline or cached integrity display, not for interactive verification or attribution of an unresolved keyRef identity. A genuinely absent proof records `presentationProofStatus: "absent"`; a present proof whose verification was not executed records `"not-evaluated"`. When the evaluator issued a challenge, it MUST refuse interactive verification if the proof is absent, failed or not-evaluated. An executed verification records `"verified"` or `"failed"` according to its actual result. The v0.5 values are `"verified"`, `"failed"`, `"absent"` and `"not-evaluated"`; historical `"missing-required"` records remain old-version diagnostics. A did-auth proof in v0.5 MUST apply all six Profile A predicates to its exact signed bytes. Valid embedded-key mathematics does not establish holder or keyRef-derived identity authority.

**Implementation status of the `proofType` values (at-risk).** Of the three `proofType` values, only `did-auth` was exercised by the earlier v0.4 fixture suite and the reference implementation. `sd-jwt-kb` (whose manifest binding depends on the still-unratified decision UM-v04-DEC-07) and `bbs-derived` are **specification-only**: no fixture and no implementation exercises them yet, so interoperability for the two is not demonstrable at this stage, and an implementation MUST fail them closed (as the reference implementation does) rather than report an unverified proof as verified. Both stay **PREVIEW / at-risk** in v0.5: the maintainers have not ratified UM-v04-DEC-07,

and its ratification and the fixtures are later-version work; a conformance claim relying on interactive presentation SHOULD state which `proofType` it implements.

## T1.3. Liveness Attestation

A `livenessAttestation` proves human presence at `attestedAt`. It does **not** prove human presence at verification time. Evaluators MUST NOT treat a stale liveness attestation as equivalent to current human presence.

Four freshness classes are defined. An attestation past its `validUntil` is `"unknown"` regardless of `attestedAt`; otherwise the classes are evaluated in the order listed and the first match applies:

- `"live"` — Attested within 60 seconds. Suitable for high-stakes transactions.
- `"recent"` — Attested within 4 hours. Suitable for spatial platform entry.
- `"stale"` — Attested more than 4 hours ago but within the attestation's `validUntil`. Evaluators SHOULD require re-attestation for sensitive operations.
- `"unknown"` — No liveness attestation present (default for offline mode), or an attestation whose `validUntil` has passed. An attestation past its `validUntil` MUST be treated as `"unknown"`.

The `livenessAttestation` is a root-level member of the manifest (carried among the optional credential-binding members of the abstract data model — EXT-OPT). When present it MUST contain `proofType` (a string identifying the liveness method, e.g. `"webauthn-uv"`; extensible via the profile registry — EXT-OPT), `attestedAt` (RFC 3339, when human presence was attested), `validUntil` (RFC 3339, after which the attestation is `"unknown"`), and `proofValue` (Base64url-encoded; MUST be computed over the UTF-8 bytes of the manifest `subject` concatenated with the UTF-8 bytes of `attestedAt`, signed by the attesting authenticator or platform key). It MAY contain `method` (the authentication method, copied into the receipt), `userVerified` (boolean, copied into the receipt), and `attester` (the DID of the attesting authority). **Signature coverage:** the `validUntil`, `method`, and `userVerified` members are covered by the holder's manifest signature, not by the attester's `proofValue` (Base §6.6); an evaluator MUST treat them as holder-asserted except where the declared `proofType`'s mechanics attest them.

WebAuthn user-verification (a WebAuthn assertion with the user-verified flag set, `proofType: "webauthn-uv"`) is the RECOMMENDED liveness method. WebAuthn user-verification follows the explicitly versioned WebAuthn assertion-validation requirements cited by this method [WEBAUTHN]. BE/BS processing refers to WebAuthn Level 3 §§6.1 and 6.1.3; Level 2 reserves those bits and does not establish their later flag semantics. Level 3 is a W3C Recommendation

dated 25 August 2026. For `proofType: "webauthn-uv"`, `proofValue` carries the WebAuthn assertion — a WebAuthn assertion signs `authenticatorData || SHA-256(clientDataJSON)`, not a raw byte concatenation — and the challenge of the assertion's client data MUST be (a digest of) the `subject || attestedAt` concatenation above; the evaluator MUST derive `userVerified` from the validated assertion's UV flag — the manifest-level `userVerified` field is a display echo and MUST NOT be the floor input (T1.3.1) when the assertion is validatable. Evaluators verifying a `livenessAttestation` MUST validate `proofValue` against the named `proofType`'s mechanics before treating the attestation's freshness class as evidence of human presence. The receipt field `livenessStatus` is an object containing `freshnessClass` (one of `"live"`, `"recent"`, `"stale"`, `"unknown"`) and, when present, the `method` and `userVerified` values copied from the attestation.

### T1.3.1. Per-Facet Liveness Floor (`requiredLiveness`) PREVIEW

A facet MAY declare a `requiredLiveness` floor (the OPTIONAL facet member named in Base §6.6). Its value is an **object**, because a liveness floor is a freshness class **plus** a boolean — a compound, unlike the closed integer `requiredTrustTier`:

- `minFreshness` — REQUIRED; one of `"live"`, `"recent"`, `"stale"`, ordered `live` > `recent` > `stale`. `"unknown"` is **not** a valid floor value (a floor of `"unknown"` would be vacuous).
- `userVerified` — OPTIONAL boolean; default `false`. When `true`, the floor additionally requires the attestation's `userVerified` to be `true`.

The floor comparison order is `live` > `recent` > `stale` > `unknown`. A facet carrying `requiredLiveness` MUST NOT be processed unless a **validated** `livenessAttestation` yields a `freshnessClass` at least as fresh as `minFreshness`, and — when `userVerified: true` is required — the attestation's `userVerified` is `true`. Enforcement is sequenced in the Consent stage (Base §3.1.4) and recorded in the receipt.

The floor is **fail-closed**:

- **`userVerified` absent ≡ `false`.** For a floor requiring `userVerified: true`, an attestation lacking `userVerified: true` (including absent) MUST fail the floor.
- **Proof validation before freshness.** An attestation whose `proofValue` does not validate under its `proofType` MUST be treated as `"unknown"` for floor purposes regardless of `attestedAt`; the floor check is sequenced strictly after proof validation, so a forged-but-fresh-looking attestation cannot bypass the floor.
- **Absent or expired root attestation ⇒ no floor satisfied.** If the root `livenessAttestation` is absent, or evaluates to `freshnessClass: "unknown"` because it is past its `validUntil`,

then **no** `requiredLiveness` floor is satisfied — including a `minFreshness: "stale"` floor — and the facet is withheld. Because `unknown` is ordered lowest, no floor value can be met by an `unknown` root attestation.

A `requiredLiveness` floor is orthogonal to `requiredTrustTier`: it can only *raise* the unlock bar for its facet and never lowers any trust-tier floor.

### T1.3.2. Authenticator Provenance PREVIEW

The `userVerified` flag proves a user-verification *gesture occurred*; it does **not** prove the provenance of the authenticator (a synced/exportable credential and a device-bound/non-exportable credential can both set `userVerified: true`), nor *which* human acted (subject-binding is holder binding, not this flag). The bare `userVerified` flag MUST NOT be relied on as evidence of hardware provenance. A deployment requiring a device-bound authenticator MUST express that requirement through `deviceAttestation` (EXT-OPT §O9.2 — the actual hardware-provenance structure), **not** through the liveness `proofType` (which names the method only); evaluators SHOULD additionally consult WebAuthn backup-eligibility / backup-state (BE/BS) flags where available. The receipt echoes whichever provenance signal the floor actually relied on: a device-bound requirement enforced via `deviceAttestation` is recorded as such, and the existing `livenessStatus` echo covers the liveness side.

### T1.3.3. Locked-Tier Portable-Unlock Profile (`um:profile:trust:locked-tier-portable-unlock`) PREVIEW

This subsection registers a thin **conformance-profile label**, `um:profile:trust:locked-tier-portable-unlock` (a `trust`-category profile per the registration mechanism — EXT-OPT §O6). The label **names a bundle of existing EXT-T1 requirements at locked-tier floors**; it does **not** define any new ceremony or cryptographic mechanism. A facet (or the consent gating it) that conforms to this profile applies, together:

- the existing per-facet liveness floor `requiredLiveness: {minFreshness: "live", userVerified: true}` (§T1.3.1) as the **user-verification floor** — no new field;
- for the highest-assurance (device-bound) rung, the existing `deviceAttestation` structure (EXT-OPT §O9.2) plus WebAuthn backup-eligibility/backup-state (BE/BS) flags (§T1.3.2) — already the prescribed route for a device-bound requirement, no new field;
- the new per-facet **assurance-class floor** `requiredAssuranceClass` (defined below);

- the unlock-window session policy (EXT-OPT §O4.5) when the locked facet is opened in a time-boxed window; and
- the receipt assurance echo and the below-floor refusal status `assuranceInsufficient` (Base §3.3.1, Base §3.3.1.1).

The label is colon-namespaced (`um:profile:trust:…`), consistent with the `um:profile:`/`um:reason:` registries; it is **not** the dot-form used for agent-delegation and consent `scope` operation tokens.

**The `requiredAssuranceClass` floor.** A facet MAY declare an OPTIONAL `requiredAssuranceClass` member — a **closed, UM-neutral, ordered enum** naming the minimum *authenticator assurance class* the unlock gesture's provenance MUST reach:

- `"software"` — a software-held key / passphrase-derived key (the lowest rung; the default);
- `"hardware-uv"` — a hardware-backed authenticator performing user verification (e.g. a platform authenticator with a hardware-protected key);
- `"hardware-bound"` — a device-bound, non-exportable hardware authenticator (the highest rung; the device-bound provenance of §T1.3.2).

The ordering is `software` < `hardware-uv` < `hardware-bound`. When absent, the floor defaults to `"software"` (the lowest value, so an absent floor never raises the bar). `requiredAssuranceClass` is modeled **in spirit on `requiredTrustTier`** (Base §6.4.5): as with `requiredTrustTier`, the floor **can only raise** the bar (it never lowers the effective assurance requirement; `requiredAssuranceClass` is a facet-level member — this draft defines no manifest-level assurance floor), and it is **fail-closed** — if the asserted or achieved assurance class is below the facet's floor, the evaluator **MUST NOT** downgrade and **MUST** withhold the facet (content never released), recording the facet status `assuranceInsufficient` (Base §3.3.1), exactly parallel to how an unsupported `requiredTrustTier` records `trustTierUnsupported`.

**Evidence→class determination.** The class the unlock gesture *achieved* is determined by the **evaluator**, from evidence it has itself verified — never from an unverified assertion — under the following rule, evaluated top-down with the first match applying:

- `hardware-bound` — a **verified** `deviceAttestation` (EXT-OPT §O9.2) on the presented manifest whose attestation evidence shows device-bound, non-exportable provenance for the authenticator that performed the gesture (the §T1.3.2 route; WebAuthn backup-eligibility/backup-state flags, where available, corroborate non-exportability);
- `hardware-uv` — a **validated** `livenessAttestation` with `userVerified: true` whose `proofType` mechanics attest hardware-backed user verification (e.g. `webauthn-uv` performed

by a platform authenticator), without verified device-bound provenance;

- `software` — every other case, including absent, unverifiable, or software-only evidence. Unverifiable evidence never raises the class (fail closed; the default rung is also the floor's default value).

The `assuranceStatus.assertedClass` receipt echo (Base §3.3.1.1) records the class **the evaluator determined** under this rule — it is evaluator-asserted, not a holder self-declaration; a holder-supplied class assertion, where a profile defines one, is a hint and MUST NOT raise the determined class. A deployment MAY substitute a stricter determination rule by local policy but MUST NOT weaken this one (for example, by treating software-only evidence as `hardware-uv`).

`requiredAssuranceClass` is **orthogonal** both to `requiredTrustTier` (trust tiers grade claim-verification / Sybil-resistance; the assurance class grades unlock-gesture strength — no formal mapping is asserted between the two axes) and to `requiredLiveness` (which it composes beside: the liveness floor grades *freshness of presence*, the assurance class grades *provenance strength of the authenticator*; a locked facet typically carries both). Because the floor is a property of the **facet / consent**, not of the unlock window or any spend ledger, a future single-use consent on a locked facet can require both a single-use spend constraint **and** this assurance floor without either field touching the other.

The following table maps each enum value to comparable assurance levels in external frameworks — NIST SP 800-63B-4 [NIST-800-63B], FIDO authenticator certification [FIDO-CERT], and eIDAS 2.0 [eIDAS2]. **This mapping is INFORMATIVE only**; the enum is the normative artifact, and UM asserts no conformance claim against NIST, FIDO, or eIDAS levels.

| `requiredAssuranceClass` **(normative)** | **NIST SP 800-63B-4 (informative)** | **FIDO authenticator certification (informative)** | **eIDAS 2.0 level of assurance (informative)** |
|---|---|---|---|
| `software` | ≈ AAL1 | FIDO L1 | Low |
| `hardware-uv` | AAL2 | FIDO L2 | Substantial |
| `hardware-bound` | AAL3 (hardware authenticator, verifier-impersonation resistance) | FIDO L3+ | High (qualified signature creation device / WSCD) |

**Schema note:** invalid-enum or wrong-type `requiredAssuranceClass` fixtures are schema-detectable by the v0.5 facet shape validator, matching the precedent set by `requiredLiveness` (§T1.3.1). The evaluator still owns cross-field and runtime assurance-floor checks, including below-floor unlock attempts.

## T1.4. Stage-2 Verification Sub-Steps

When the manifest relies on the credential-binding mechanisms of this profile for Tier 1 or higher assurance, the evaluator MUST perform the following sub-steps as part of the **Verify** stage (Base §3.1.2). Each sub-step records its outcome in the credential-binding receipt fields (Base §3.3.1.1). Evaluators that do not implement credential binding skip these sub-steps without claiming success. A present presentation proof that was not evaluated records `presentationProofStatus: "not-evaluated"`; only actual absence records `"absent"`. The existing `holderBindingStatus` and `crossDidBindingStatus` vocabularies remain as specified by Base. Because `livenessStatus` is an object, its no-implementation form is `livenessStatus.freshnessClass: "unknown"` (or the field is omitted).

1. **(2a) Holder-binding verification.** For each claim carrying `holderBinding` (T1.1), verify the binding according to its `mode`. Record `holderBindingStatus`. A claim relied upon for assurance whose holder binding fails or is absent MUST NOT be granted trust above Tier 0.
2. **(2b) Presentation-proof verification.** If the manifest carries a `presentationProof` (T1.2), verify that `challenge` matches the verifier-issued nonce, `audience` matches this verifier, and `proofValue` validates the binding of the signing-input hash, `challenge`, `audience`, and `created` under the declared `proofType`'s rule (T1.2). Record `presentationProofStatus` according to the check actually performed. A failed presentation proof MUST cause the manifest to be treated as replay-suspect and rejected for interactive verification. When the evaluator issued a challenge and no `presentationProof` is present, record `presentationProofStatus: "absent"` and reject interactive verification. A present proof that cannot be evaluated records `"not-evaluated"` and also refuses interactive verification.
3. **(2c) Liveness-attestation verification.** If a `livenessAttestation` (T1.3) is present, verify its proof and compute its `freshnessClass` from `attestedAt` and `validUntil` per T1.3 (an attestation past its `validUntil` is `"unknown"` regardless of `attestedAt`). Record `livenessStatus`. Evaluators MUST NOT treat a `stale` or `unknown` freshness class as current human presence. An evaluator that cannot validate the attestation's `proofType` records the temporal `freshnessClass` with the attestation treated as **unvalidated**: it MUST NOT satisfy any `requiredLiveness` floor (T1.3.1) and MUST NOT be treated as evidence of human presence.
4. **(2d) Cross-DID binding-proof verification.** For each `identity.crossDidBinding` claim (Base §6.4.4), verify the attester authorization (when attester-asserted) and any Tier 2 `bindingProof` (EXT-T2) or Tier 3 `ceremonyProof` (EXT-T3) per the claim-proof process (T1.5). Record each result in `crossDidBindingStatus`, one of `"verified"`, `"failed"`, `"absent"`, or `"trustTierUnsupported"` (the last when the claim declares a

`requiredTrustTier` the evaluator has no implemented verification profile for — Base §6.4.5); `"absent"` indicates the claim carried no binding proof to verify.

These sub-steps are the normative record of how the binding mechanics weave into the evaluation sequence; the binding modes and proof procedures they invoke are defined in this profile and in EXT-T2 / EXT-T3.

## T1.5. Claim Proof Process — End-to-End Verification Path

This is the normative end-to-end procedure an evaluator follows to determine whether a claim carried in a manifest is trustworthy at Tier 1 or above. It ties together the outer manifest signature verification (Base §1.6), the `claimProof` field (Base §6.4.3), and the cross-DID binding attestation (Base §6.4.4) into a single verification chain.

### T1.5.1. `claimProof` Verification Procedure

When claiming Tier 1 assurance or higher, the evaluator MUST perform the verification steps below; at Tier 0 these checks are OPTIONAL.

- When `claimProof` is present as an **embedded object**, the evaluator MUST verify the VP proof chain: (a) the VC signature validates to the stated issuer, (b) the VP signature validates to the holder, (c) the holder DID matches the manifest subject.
- When present as a **URI string**, the evaluator MAY fetch the VP for verification when network access is available. Evaluators that cannot resolve the URI SHOULD record the claim as `"unverified"` with a reason such as `um:reason:trust:claimproof-unresolved`, rather than `"verified"`.
- When `claimProof` is an **array**, each entry is independently verified; the claim is backed by the conjunction of all valid proofs.

VPs used as `claimProof` SHOULD include domain (audience binding) and challenge (nonce) parameters to prevent cross-manifest replay. Evaluators MUST enforce size limits of at most 50 KB per embedded VP and at most 500 KB total VP payload across all claims in one manifest, and MUST document any stricter limit in their conformance claim.

### T1.5.2. Seven-Step Verification Chain

For each claim requiring Tier 1 or higher trust, the evaluator MUST execute the following seven-step procedure:

1. **Verify outer manifest signature** per Base §1.6. If the signature is invalid, the entire manifest is rejected.
2. **Confirm manifest signing key authorization.** Resolve the manifest `subject` DID document. Confirm that the signing key (referenced via `signature.keyRef`) appears in the subject's DID document under the `authentication` or `assertionMethod` verification relationship, per [DID-CORE] §5.3. If the manifest signing key is not authorized for the subject, the evaluator MUST record this as a verification failure.
3. **Resolve the claim issuer DID.** For each claim carrying `claimProof`, resolve the `claims[].issuer` DID document per W3C DID Resolution.
4. **Verify the claim proof chain.** If `claimProof` is present:
    1. Identify the proof type. If `proofType` is absent, infer from the proof structure.
    2. Verify the proof material per the declared proof type, following the VP proof-chain verification in T1.5.1.
    3. If `verificationMethod` is present, resolve the URI to the issuer's DID document. Confirm the key appears under the `assertionMethod` verification relationship. This is the key-authorization step: it proves the signing key was authorized by the stated issuer DID to issue credentials. (See [DID-CORE] §5.3 and [VC-DATA-MODEL] §4.12, Securing Mechanisms.)
    4. If `statusRef` is present on the proof entry, check revocation status per the statusRef resolution schema (EXT-OPT).
    5. If `claimProof` is an array, each entry is independently verified; the claim is backed by the conjunction of all valid proofs.
5. **Verify attester key authorization** (for attester-asserted `identity.crossDidBinding` claims; skipped when the binding carries only a cryptographic `bindingProof` or `ceremonyProof`, whose verification is defined in EXT-T2 and EXT-T3). Resolve the `attester` DID document. Confirm the attester's signing key appears under the `assertionMethod` verification relationship. Confirm the attester is on the evaluator's configurable trust list per Base §6.4.4.
6. **Confirm key revocation status.** For all signing keys encountered in steps 1–5 (manifest signer, claim issuer, attester), confirm that no key relied upon has been revoked, to the extent revocation information is available. Key revocation is determined per key source: for DID-based keys, the verification-method state in the key's DID document; for keys carried by a

credential, the proof entry's own `statusRef`. This is distinct from the manifest-instance status resolved via `signature.statusRef`. Inner key revocation (claim-issuer and attester keys) SHOULD be checked when revocation information is available.

7. **Compose per-claim outcomes.** Aggregate the verification outcomes into the manifest's trust decision per Base §6.4.5 (`requiredTrustTier` floors) and emit the receipt per Base §3.3.

### T1.5.3. Key Purpose Requirements

For each step that names a verification relationship, the evaluator MUST confirm the signing key appears under the matching W3C DID Core verification relationship:

- **Manifest signature:** `authentication` or `assertionMethod` on the manifest subject's DID document.
- **VC issuance (claimProof embedded VP):** `assertionMethod` on the VC issuer's DID document.
- **Cross-DID binding attestation:** `assertionMethod` on the attester's DID document.
- **Agent delegation signing:** `capabilityDelegation` on the delegator's DID document (Base §6.5).

### T1.5.4. Trust Policy vs. Cryptographic Verification

The verification chain separates two concerns that MUST NOT be conflated:

- **Cryptographic verification** (steps 1–2, 4–6): mathematical proof that signatures are valid and keys are authorized. Deterministic.
- **Trust policy** (step 5 trust list, step 7 composition): which DIDs and attesters the evaluator trusts. Local configuration. The specification MUST NOT mandate a centralized trust registry; each evaluator defines its own trust policy.

**Non-normative note:** OMATrust's Key Binding attestation type operationalizes the key-authorization pattern for the EVM ecosystem. UM cites W3C DID Core and VCDM 2.0 as the normative sources for the verification-relationship model.

## T1.6. Security Considerations

### T1.6.1. BBS+ Pseudonym Scope

If `pseudonymScope` is reused across verifiers, cross-verifier correlation becomes possible. Issuers SHOULD generate unique scope URIs per verifier relationship. Evaluators MUST NOT use a shared pseudonym scope for multiple independent verifier relationships.

### T1.6.2. ZK Proof System Trusted Setup

Tier 2 profiles using Groth16 require a trusted setup ceremony (EXT-T2). The ceremony parameters MUST be publicly auditable. Verifiers MUST verify proofs against the published verification key, not against a key provided by the prover.

An implemented circuit profile MUST bind the circuit identifier and version to the exact verification-key bytes, their integrity digest, proof system, public-input schema and ordering, and required setup/trust evidence. The evaluator obtains that binding from a trusted versioned profile or deployment configuration pinned before evaluating the untrusted proof. A circuit URI in a proof is a selector, not a trust anchor or permission for an arbitrary network fetch. A prover-supplied or fetched key MUST NOT become authoritative merely because the proof verifies under it. Unknown, mismatched, or unavailable required bindings yield unsupported or failed verification as applicable and grant no higher tier. Offline verification against a trusted pinned key remains permitted. This rule selects no new circuit, fetch protocol, digest algorithm, or proof system; each implemented profile must define those parameters explicitly.

### T1.6.3. Reciprocal Control Limitations

Reciprocal control (two DIDs signing the same challenge) proves the same entity or cooperating entities control both keys. It does **not** prove they are the same natural person. For `sameSubject` assurance, combine reciprocal control with issuer-attested binding (Tier 1) or linked-secret proof (Tier 2 — EXT-T2).

### T1.6.4. Backwards Compatibility (v0.3 manifests)

v0.5 evaluators encountering a v0.3 manifest (no `holderBinding`, no `presentationProof`, no `livenessAttestation`) MUST:

1. Parse and process the manifest through the six-stage evaluation sequence.
2. Record `holderBindingStatus: "absent"` in the receipt.
3. Record `presentationProofStatus: "absent"` in the receipt.
4. Record `livenessStatus.freshnessClass: "unknown"` in the receipt.
5. Cap the effective trust tier at Tier 0 regardless of `requiredTrustTier` declarations.

A v0.3 manifest that declares no manifest-level `requiredTrustTier` (or declares Tier 0) is accepted but flagged as "unbound" at Tier 0; evaluators decide their own policy for unbound manifests. Where a v0.3 manifest declares a manifest-level `requiredTrustTier` that cannot be satisfied because binding material is absent, Base §6.4.5 governs the outcome: `accepted-partial` if other items can still be processed, or `rejected` when the unsupported tier applies at the manifest level (or no processable items remain — Base §3.1.5). There is no display-only outcome branch; whether an application still renders anything after a `rejected` evaluation is an application-layer choice that does not change the receipt outcome. Capping the effective trust tier at Tier 0 (step 5) does not by itself reject the manifest; the declared floor does.

v0.5 evaluators SHOULD emit an `um:reason:trust:unbound-claims` warning in the receipt when processing manifests without holder binding. *(Worked example: unbound-claims warning — see Cookbook.)*

## References

Cited in this profile (full descriptors in the published artifact):


- **[RFC9901]** SD-JWT (Selective Disclosure for JWTs), incl. Key Binding JWT.
- **[RFC7800]** Proof-of-Possession Key Semantics for JWTs (`cnf`).
- **[RFC7638]** JSON Web Key (JWK) Thumbprint.
- **[VC-DI-BBS]** Verifiable Credentials Data Integrity BBS Cryptosuites.
- **[DID-CORE]** Decentralized Identifiers (DIDs) v1.0. https://www.w3.org/TR/did-core/
- **[VC-DATA-MODEL]** Verifiable Credentials Data Model v2.0. https://www.w3.org/TR/vc-data-model-2.0/
- **[WEBAUTHN]** Web Authentication: An API for accessing Public Key Credentials. **Level 2** (W3C Recommendation, https://www.w3.org/TR/2021/REC-webauthn-2-20210408/) defines the earlier assertion-validation requirements. **Level 3** (W3C Recommendation, 25 August 2026, https://www.w3.org/TR/2026/REC-webauthn-3-20260825/) defines BE/BS in §§6.1 and 6.1.3. The method's explicitly cited edition determines the validation requirements.

- **[NIST-800-63B]** *(informative)* NIST SP 800-63B-4, Digital Identity Guidelines — Authentication and Authenticator Management (final, 2025). Defines Authenticator Assurance Levels (AAL 1–3). Cited only by the informative assurance-class mapping (§T1.3.3).
- **[FIDO-CERT]** *(informative)* FIDO Alliance Authenticator Certification Levels (L1–L3+). Cited only by the informative assurance-class mapping.
- **[eIDAS2]** *(informative)* Regulation (EU) 2024/1183 (eIDAS 2.0), levels of assurance Low / Substantial / High. Cited only by the informative assurance-class mapping.

✧ ✧ ✧

*Companion to the Base specification. For Tier-2 cryptographic binding, see EXT-T2. For worked examples, see the Cookbook.*