MINTID

MintID Verifier & SDK Integrator Manual

How to become a relying party on the MintID network and integrate the verifier SDK: registration, the presentation pipeline, and operations.

VERSION 0.1 · 2026-08-09 · PUBLIC EDITION

CONTRACT VERIFIER-SERVICE.V1 1.1.0 · SPEC-1 V1.3 PIN · ENGINE: STUB (PRE-F3)

VERIFIER FEES ARE OPEN PARAMETERS — NO FIGURE HEREIN IS A RATIFIED PRICE

> **Status and scope.** Public edition (v0.1). The network is pre-genesis (internal testnet). The anonymous-credential **proof engine is not yet live**: current SDK builds carry a pinned stub engine (`proof-core 0.1.0+stub.pre-f3`) that refuses every proof, fail closed. You can build and test a complete integration against injectable test doubles today; you cannot accept real anonymous proofs until the cryptographic milestone (Epic F3) ships with its audits. Verifier registration costs and any verifier bond are **open policy parameters** — nothing in this manual is a ratified price.

---

1. The verifier role in five minutes

A **verifier agency** is a relying party — a bank, marketplace or platform — that accepts privacy-preserving identity presentations. The holder proves *predicates* ("over 18", "KYC grade ≥ A3", "credential active") without revealing attributes, documents or identifiers.

What a verifier may do: request minimal claims, validate current chain state and proof, enforce nonce and expiry.

What a verifier must never do: request unapproved claims, use wildcard origins, replay a proof, or write proof activity to the chain. Verifiers validate cryptographic evidence only — they never inspect passports, biometrics, names, addresses or KYC case files.

What you receive on an accepted presentation: the issuer identity and the boolean policy outputs you asked for — never the exact assurance grade, the expiry date, the credential ID, or any personal attribute (R6). What you may retain is a closed five-field decision record (§6) — nothing else, structurally.

What it costs: verification itself is **free by protocol design** (R15/R18): presentations are off-chain and carry no usage tax. Registration cost and an optional verifier bond are open parameters that must be finalized before genesis — no band has been published.

The one rule that shapes everything: the decision pipeline exists in exactly **one** implementation (SPEC §14.4). Client SDKs are transport-only; nothing you deploy can weaken an acceptance condition, and the 10-second presentation lifetime is a constant, not a setting.

2. Becoming a verifier — installing an SDK does not make you one

Registration is on-chain, KYC-backed and council-approved:

```
Verifier KYC completed
  → accepted issuer produces verifier-KYC attestation commitment
  → verifier registers root request-signing key and exact domain/origin
  → verifier proves domain control
  → council approves
  → verifier is active on chain
```

No KYC documents enter the chain; it stores only a digest of the domain-control evidence. Before your first presentation you need all of:

1. **On-chain registration** (`MsgRegisterVerifier`, council/authority-driven) with your record in state **ACTIVE**. States: `PENDING → ACTIVE → ENHANCED_MONITORING / SUSPENDED / REVOKED`.
2. **Each exact origin registered and kept current** (`MsgRegisterVerifierOrigin` / `MsgRenewVerifierOrigin`). Origins are strict: `https://host[:port]` only, lowercase, no wildcards (R24), no IP literals, no paths, no default `:443`, and **no canonicalization — a non-canonical spelling is rejected, never rewritten**. One verifier per origin, globally unique. Domain control is revalidated periodically via a well-known HTTPS endpoint and/or DNS proof; a lapsed proof rejects presentations with `revalidation_expired`.
3. **Request-signing keys registered** (`MsgRotateVerifierKey`, self-service from your `operator_account`). Ed25519 only at launch; `key_id = SHA-256(public_key)`; only an ACTIVE key verifies.
4. **Chain access**: any RPC node(s) for state queries — the SDK treats them as **malicious** and proves every answer with ics23 against a trust anchor you designate. There is no obligation to run a trusted full node, but never point the anchor at an arbitrary public RPC.
5. **A durable nonce store**: crash-durable, atomic, exactly-once consumption. The bundled `SqliteNonceStore` (`synchronous=FULL`) qualifies.
6. **A disciplined clock** (`now_unix`): skew can only make you *more* rejecting, never widen acceptance.
7. **A retention posture**: keep the decision log and nothing else; any schema extension is a privacy review.

3. The SDK: one core, transport-only clients

Per ADR-0003 there is **one verified core** and thin clients — a per-language reimplementation is rejected permanently.

MODE	WHAT YOU RUN	FOR WHOM
Embedded	Your Python backend imports `mintid-verifier-sdk` and calls `validate_presentation` in-process	Python backends; lowest latency
Service	The same SDK wrapped in a thin first-party HTTP service you self-deploy	Any backend language; operational isolation
Client	`@mintid/verifier-client` (TypeScript, Node ≥ 20, ESM, zero runtime deps) against **your own** service instance	Node/TS backends (PHP client is next)

Service mode is embedded mode plus transport: it adds no authority, exposes no extra options, and runs the identical nine-condition pipeline — **there is no "lite" verification path**. Pointing your client at somebody else's verifier service would delegate the *decision* itself; the protocol's paid-services permission (R26) explicitly does not cover outsourcing the decision.

Current packages (all `0.1.0`, private, unpublished): `mintid-verifier-sdk` (Python ≥ 3.12, Rust engine compiled in via maturin), `@mintid/verifier-client`, and `@mintid/verifier-mcp` (an MCP server over your own service, for agent tooling: `mintid-verifier-mcp --service-url http://127.0.0.1:8471`). Design partners consume from the repository at a tag; the target channel is `pip install mintid-verifier-sdk` — no Rust toolchain needed.

You supply infrastructure, never policy. The five ports: `ChainReader` (which nodes — never whether reads are proven), `NonceStore` (backend — never the exactly-once semantics), `ProofVerifier` (no freedom: the wheel's pinned engine), `DecisionLog` (where records go — never what they contain), and the clock.

4. The presentation flow, end to end

```
your backend                          the chain
    │  1. create challenge                ▲
    ▼                                     │ proven reads (ics23)
[verifier SDK / your service] ────────────┘
    │  2. §10.5 signed challenge (10 s lifetime)
    ▼
 holder wallet ── proof bound to your exact challenge
    │  3. envelope (proof + public outputs)
    ▼
[verifier SDK] — nine conditions, in order, fail closed
    │  4. PresentationDecision + one minimal decision record
    ▼
your business logic
```

Issuing a challenge (Python):

```
request = PresentationRequest(
    session_id=secrets.token_bytes(16).hex(),
    nonce=secrets.token_bytes(32),
    verifier_id=MY_VERIFIER_ID,          # your on-chain registration
    audience="https://checkout.example", # one of your registered exact origins
    request_key_id=MY_REQUEST_KEY_ID,    # registered for that origin
    expires_at_unix=now + PRESENTATION_LIFETIME_SECONDS,   # = 10, a constant
    finalized_chain_height=chain.latest_height(),
    issuer_policy=IssuerPolicy(),        # or restrict accepted issuers / state minimum
    claim_policy=(ClaimPredicate(claim="age_over", value=18, operator=">="),),
)
signed_body = request.to_json()          # canonical; sign with your registered key
```

Validating the holder's response (Python entrypoint):

```
decision = validate_presentation(
    request, envelope,
    chain=chain, nonces=nonces, proofs=proofs, decisions=log,
    now_unix=now, max_height_lag=20,
)
decision.accepted      # bool
decision.reason_code   # closed vocabulary (§5)
decision.verified      # VerifiedPresentation | None
```

TypeScript, against your own service:

```
const client = new VerifierClient("http://127.0.0.1:8471");
const challenge = await client.createChallenge({
  claim_policy: [{ claim: "age_over", value: 18, operator: ">=" }],
});
const decision = await client.validatePresentation({
  request: challenge.request, envelope: holderResponse,
});
```

The nine acceptance conditions (SPEC §10.5) — always run in full, in order, fail closed:

1. verifier and exact origin are active on-chain;
2. signed request key is registered for that origin;
3. nonce has not been used;
4. response arrives before the 10-second deadline;
5. presentation is bound to the origin, verifier ID, request key, nonce, policy, and chain height;
6. every requested predicate is proven;
7. issuer is active and status root is current;
8. chain state is rechecked immediately before decision;
9. nonce is irreversibly consumed in the verifier's own durable store.

Hard numbers: presentation lifetime **10 s** (R14, non-configurable); nonce **256-bit**, session id 128-bit; challenge height must sit within `[latest − max_height_lag, latest]` (default lag 20 blocks, local policy, never per-request); issuer status roots every **30 s**, rejected past **45 s** of age — stale is always **fail-closed**. Binding is total: the policy digest commits to the whole canonical request body, so tampering with any part of the challenge breaks condition 5.

5. Reason codes – the closed vocabulary

CODE	COND.	MEANING	TYPICAL HANDLING
`accepted`	–	all nine conditions passed	proceed
`request_malformed`	pre	challenge violates the §10.5 shape	fix challenge construction
`verifier_not_active`	1	your registration is not ACTIVE on proven state	check registry state
`origin_not_registered`	1	origin does not resolve to you (no trust-by-name)	register/renew the origin
`revalidation_expired`	1	domain-control proof lapsed	`MsgRenewVerifierOrigin`
`request_key_not_active`	2	signing key unregistered or rotated out	fix key rotation
`nonce_replayed`	3	nonce already consumed	fresh challenge; investigate
`deadline_exceeded`	4	response at or after the 10 s deadline	fresh challenge
`proof_invalid`	5	proof failed or bound to a different context	reject; possible tampering
`predicate_missing`	6	a requested predicate is not proven	reject
`issuer_not_accepted`	7	issuer outside your accepted-issuer policy	your policy decision
`issuer_not_active`	7	issuer not operational (or below your minimum)	reject; retry later is fine
`status_root_stale`	7	proof built on a superseded/expired root	holder retries fresh
`height_stale`	8	challenge height fell out of the lag window	fresh challenge
`state_unproven`	1/8	a chain read could not be proven – including absence	**your pager, not your bug** – never bypass
`nonce_race_lost`	9	a concurrent presentation won the atomic gate	reject; exactly one winner

Reserved and structurally unreachable: `origin_invalid`, `domain_validation_failed`. The only policy knob in the vocabulary: with the default `issuer_state_minimum: "ACTIVE"`, an issuer under enhanced monitoring rejects as `issuer_not_active`; opting into `"ENHANCED_MONITORING"` accepts it.

6. Privacy and retention: the decision record

What you keep per decision (accept *or* reject) is exactly one `DecisionRecord`:

FIELD	CONTENT
`session_id`	your challenge's session
`accepted`	boolean
`reason_code`	one code from §5
`bound_context_digest`	32-byte digest of the bound context
`decided_at_unix`	decision time

Proof bytes, claim values and personal data **structurally do not fit** — tests assert field-set equality, and no presentation is ever broadcast, indexed or committed to the chain (R18). Expect a quarterly verifier privacy and domain-control review; the network's transparency reporting covers issuer/verifier states, never holder data.

7. Running service mode

```
python -m verifier_core.service \
  --origin https://checkout.example \
  --verifier-id-hex <64 hex> \
  --request-key-id-hex <64 hex> \
  --state-rpc  http://your-rpc-node:26657 \
  --anchor-rpc http://your-own-node:26657 \
  --nonce-db   /var/lib/mintid/nonces.sqlite \
  --decision-log /var/log/mintid/decisions.jsonl \
  --signer your_pkg.signing:HsmRequestSigner
```

- Identity (origin, verifier id, request key) is **deployment configuration bound to your on-chain registrations — never accepted from HTTP callers** (bIP-0001).
- HTTP semantics: a parseable body always gets **200 + decision** (accept or reject) plus one decision record; **400** only for unparseable bodies; **503** means proven chain state was unavailable — fail closed, retry, never work around. Clients throw only on transport; **branch on data, not on exceptions**.
- Defaults: `--host 127.0.0.1 --port 8471`. The built-in runner is a single-threaded reference server for devnet/integration — front it with your own web tier.
- `--insecure-unsigned-challenges` (mutually exclusive with `--signer`) is devnet-only; production request signing is HSM territory (MINT-48).

8. Conformance and versioning

- The contract `verifier-service.v1` is frozen at **1.0.0** (bIP-0001, accepted 2026-07-21), currently **1.1.0** after an additive build-info pin (MINT-173). Three operations: `GET /v1/build-info`, `POST /v1/challenges`, `POST /v1/presentations`.
- The conformance suite (`mintid-presentation-conformance` 1.0.0, 20 vectors) covers one accepted case plus ≥1 rejection per acceptance condition and every producible reason code; Python replays it through library and HTTP surfaces, TypeScript over real HTTP; vectors are byte-compared against regeneration in CI.
- **Decision semantics are major**: any change to accepted behavior or the reason vocabulary is a major version everywhere. Any two artifacts with the same contract major and suite version are **decision-equivalent** in any language.
- Check what you are running at runtime: `from verifier_core.build import build_info` — today it reports `sdk 0.1.0`, `engine proof-core 0.1.0+stub.pre-f3`, `SPEC-1 v1.3`, suite `1.0.0`. Note the SDK pins SPEC-1 v1.3 while the frozen master is v2.0 (2026-08-02); §14.4 was appended without changing R1–R28 semantics.

9. Trying it on the internal testnet

The project operates an internal testnet: a single validator plus a registered reference verifier (operated by Conectia PRO, bootstrapped via genesis with placeholder KYC linkage). Its limitations are explicit and by design: the proof engine is the pre-F3 stub (**every presentation refuses**); challenges are unsigned; real admission remains council-driven (R13). Challenges, proven state reads, origin resolution, staleness and decision records are all real and testable today. The testnet's endpoints are not published; clients must always point at **your own** service instance — the testnet is MintID's own deployment, useful as a wire reference only.

Annex – Requirement quick reference

ID	ONE LINE
R2	No raw KYC, credential serials or presentation logs on-chain, ever.
R6	Grades are private; you receive threshold proofs (`grade ≥ A3`), never the exact grade.
R9	Issuers heartbeat every ≤30 s; reject anything stale.
R13	Verifiers are registered, KYC-approved, and prove control of every origin and key.
R14	Presentations live 10 seconds, fully bound to your challenge.
R15	Verification is online against latest finalized state – and free.
R18	Presentations stay off-chain; you retain only the minimal decision record.
R21/R22	Predicates, not attributes; one session may prove several predicates.
R23	Holders may pin your (ID, origin, key fingerprint) locally; changes require their re-consent.
R24	Exact origins only; wildcards prohibited; periodic domain revalidation.
R26	Paid proof-verification APIs are allowed but never buy influence – and never outsource the decision.
§14.4	One decision pipeline, transport-only clients, no weakening knobs, nothing browser-side.

Manual v0.1 (public edition) – 2026-08-09. Sources: `master/SPEC-1-…` *v2.0 (§10.5, §11, §14),* `docs/sdk/00-06`*,* `docs/integration/verifier-sdk.md`*, bIP-0001, ADR-0002/0003,* `deploy/testnet/` *(MINT-212). The proof engine and production request signing land with Epic F3; verifier fees are open pre-genesis parameters.*