



MINTID · ROLE MANUAL

MintID Verifier & SDK Integrator Manual

How a relying party becomes a registered MintID verifier and accepts privacy-preserving presentations through one verification pipeline and a public service contract.

PUBLISHED BY

MintID Foundation

Netherlands Stichting, Amsterdam · related parties disclosed

VERSION

0.2 · October 2026

AUDIENCE

Relying parties and integrators

CONTRACT

Verifier Service 1.5.0

VERIFICATION

Free by design

Contents

01	The verifier role at a glance	3
02	Becoming a registered verifier	4
03	One pipeline, thin clients	5
04	The presentation flow	6
05	The public service contract	8
06	Reason codes	9
07	Privacy, retention and operations	11
08	Agent payments: x402 and Web Bot Auth	11
09	Conformance, versioning and status	13
10	Getting access and the public testnet	14
A	Go-live checklist	15
B	Core requirements that shape a verifier	16

Status and scope. MintID runs in production on a single-node chain, which is never a place to test; the distributed mainnet has not launched. Third parties test on **the public testnet**, with its KYC sandbox, where nothing issued carries weight; whether it is live is stated in one place, the developer documentation’s page *The public testnet and its KYC sandbox* (chapter 10). MintID ships no consumer wallet (holders run the holder library in their own infrastructure, and a wallet built on the same transports can present too). The verifier service, its public contract and the proof core are implemented and in production. This manual explains what a verifier does and why; for the wire format, the published service contract governs. The cost of verifier registration and any verifier bond are open pre-genesis parameters. This manual is not legal advice, and nothing is for sale.

01

The verifier role at a glance

A verifier learns that a claim is true, not who made it, and keeps nothing but a minimal record of its decision.

A **verifier** is a relying party (a bank, a marketplace, a platform, a payment facilitator) that accepts privacy-preserving identity presentations instead of collecting documents. The holder proves *predicates* such as *over 18*, *assurance grade ≥ A3* or *credential active* without revealing attributes, documents or identifiers. Verifiers are held to account too: every MintID verifier is KYC-approved, registered on-chain, bound to exact web origins and approved by the compliance council.

10 s
lifetime of every challenge and its presentation, fixed and not configurable

45 s
maximum age of an issuer status root before verification fails closed

0
presentations, proofs or decision records written on-chain

Free
basic verification, by design: no usage tax exists or is planned

ASPECT	WHAT IT MEANS FOR A VERIFIER
May	Ask only for the claims it needs; check the proof against current finalised chain state; make sure each challenge is used once and before it expires
Must never	Ask for claims it is not approved to request; use wildcard origins; replay a proof; write proof activity to the chain; handle passports, biometrics, names or KYC case files
Receives on acceptance	The accepted issuer, the status root the proof was checked against and the predicates proven; never the exact grade, the expiry date, the credential identifier or any personal attribute
Keeps	One minimal decision record per decision, whether it accepts or rejects, and nothing else
Pays	Nothing per verification. Registration cost and any verifier bond are set before genesis; no figure is published

One pipeline. A single implementation of the acceptance logic; nothing you deploy can weaken it.

Ten seconds, always. Every challenge expires ten seconds after issuance, in every language.

Fail closed. Stale, unproven or unavailable chain state always means rejection.

Keep the record, not the proof. Presentations are never stored, forwarded, logged or put on-chain.

Your service is your verifier. Clients talk only to a service instance that you operate.

02

Becoming a registered verifier

Installing an SDK does not make you a verifier: registration happens on-chain, is backed by KYC and needs an admission decision. Today that decision is a governance proposal voted by the chain's single validator; the compliance council takes it over once seated.

The pipeline refuses presentations for any identity that is not registered. It looks up your origin in authenticated registry state on every decision. Registration follows five steps:

STEP 1 Verifier KYC Complete KYC with an accepted issuer. Only a commitment to the attestation reaches the chain. off-chain	STEP 2 Keys and origins Register your request-signing key and each exact HTTPS origin you will present from. on-chain	STEP 3 Domain control Prove control of every origin through a web and/or DNS proof. The chain keeps only a digest. digest only	STEP 4 Admission decision Your policy and domain-control evidence are checked and your record is activated by governance proposal: today voted by the chain's single validator, by the compliance council once it is seated. governance	STEP 5 Active Presentations can now be accepted. Keep keys, origins and domain control current. revalidate
--	--	---	--	---

Until the compliance council is seated, admissions go through on-chain token governance. The hand-over to the council is explicit, so this interim arrangement cannot become permanent.

What the chain records about you

Your record is small and organisational. It holds your verifier identifier and public display name, your authority state, the accepted issuer that performed your KYC and **a commitment to that attestation, never the attestation itself**, your request-signing keys and their rotation history, your registered origins, a digest of each domain-control proof and when revalidation is due. Your authority state is one of a closed set: *pending*, *active*, *under enhanced monitoring*, *suspended* and *revoked*, which is final. Presentations are accepted only while you are active. No holder data, presentation or decision record ever enters your record.

Exact origins

- An origin is the HTTPS scheme, the host and, if it is not the default, the port. Paths, wildcards and IP addresses are not allowed.



- Register each origin in its canonical form, the same form your challenges will carry. A non-canonical spelling is rejected, never silently rewritten.
- An origin belongs to one verifier only. A lookalike domain cannot borrow your standing, because trust is never granted by name.
- Domain control is revalidated periodically. If a proof lapses, presentations for that origin are rejected (`revalidation_expired`) until you renew it. The revalidation cadence is an open pre-genesis parameter.

Request-signing keys

- Every challenge is signed with a key registered for the origin it names. Only an active key verifies, and keys are Ed25519 at launch.
- You rotate keys yourself from your registered operator account. Keep signing keys server-side, in hardware-backed signers; they never reach a browser.
- Rotating a key affects holders, as the next section explains.

Holder-side trust and re-consent

Each holder's software keeps a **local** list of trusted verifiers, keyed to your verifier identifier, your exact origin and the fingerprint of your request key. Nothing is trusted by name. A holder must consent again after any key rotation, any origin change, any change in your KYC status and while you are under enhanced monitoring. These approvals never go on-chain, and neither you nor any validator can see them. **In practice:** plan rotations and origin changes ahead, and expect returning users to see a consent prompt after one.

03

One pipeline, thin clients

The acceptance logic is implemented once, and every way of using it runs that same code.

Identity systems have often gone wrong when each programming language got its own copy of the acceptance logic. MintID therefore keeps **one verification core**, which you can use in three ways, plus an integration for agent tooling. None of them has a “lite” path.

MODE	WHAT YOU RUN	SUITS
Embedded	The verifier SDK inside your own backend (the Python package <code>mintid-verifier-sdk</code>)	Python backends; the lowest latency
Service	The same SDK behind a thin first-party HTTP service that you deploy yourself and that exposes the public contract (chapter 5)	Backends in any language; operational isolation
Client	A transport-only client pointed at your own service instance: <code>@mintid-net/verifier-client</code> for Node.js and TypeScript backends	TypeScript today; other languages as demand appears
Agent tooling	<code>@mintid-net/verifier-mcp</code> : the contract's three operations exposed as Model Context Protocol tools, working against your own service	Agent frameworks running inside your own infrastructure

Service mode is embedded mode with an HTTP layer added. It adds no authority and no options, and it runs exactly the same pipeline. Clients contain no cryptography, no single-use store and no stored presentations, and they have

no setting that could weaken an acceptance condition. Challenges and request keys stay on the server in every deployment, and no client is meant to run in a browser.

Your service is your verifier. Point clients only at a service instance that you operate as the registered verifier. You *can* rely on another verifier’s decision, as an x402 seller relies on its facilitator (chapter 8), but then that party is the verifier: the presentation binds to its origin and the accountability is its own. Selling verification as a service is allowed, but that revenue is kept separate from consensus rewards, and basic verification stays free.

You supply the infrastructure, never the policy

ELEMENT	YOUR CHOICE	FIXED BY THE PIPELINE
Chain access	Which nodes you query	Every answer is checked against a light-client-verified header; nodes are always treated as untrusted
Trust anchor	Which header source you rely on (ideally your own node, never an arbitrary public endpoint)	State that cannot be proven, including a claim that a record does not exist, leads to rejection
Single-use store	The storage backend	Each challenge must be consumed exactly once, atomically, in a way that survives a crash
Proof engine	None: it is pinned inside the SDK artefact	Which engine runs and how it verifies
Decision log	Where the records go	What a record may contain
Clock	Your time source	It must be kept accurate; clock skew can only make you stricter, never more accepting
Staleness window	How far a challenge’s chain height may fall behind, set when you deploy	It is never a per-request input

Distribution. The packages are pre-release: their source is published from the main repository under the Apache License 2.0, and none is on a public package index yet. They will be published on a public package index with the proof engine compiled in at a pinned version, so installing them needs no extra toolchain. Other client languages will follow when there is demand, and any future in-process verification outside the embedded SDK will use controlled bindings to the same audited core, never a rewrite.

04

The presentation flow

One signed, single-use, ten-second challenge; one zero-knowledge proof; one decision.

STEP 1 Challenge Your backend asks your service for a challenge that states the claims it needs. 10 s starts	STEP 2 Forward You pass the signed challenge to the holder's software over OpenID4VP. OpenID4VP	STEP 3 Prove The holder software checks its trust pins and returns one encrypted proof covering every predicate. holder-side	STEP 4 Validate Your service checks the acceptance conditions against finalised chain state. fail-closed	STEP 5 Decide Your business logic gets one decision, and one minimal record is kept. nothing on-chain
--	---	--	--	---

What a challenge contains

The service builds the challenge, validates its shape, signs its canonical form with your registered request key and returns it. Its fields are exactly the ones in the public contract:

FIELD	SET BY	MEANING
<code>session_id</code>	service	A 128-bit session identifier, generated on the server
<code>nonce</code>	service	A 256-bit single-use random value, generated on the server
<code>verifier_id</code>	deployment	Your on-chain verifier identifier
<code>audience</code>	deployment	The exact HTTPS origin the presentation binds to
<code>request_key_id</code>	deployment	The registered key that signs the challenge
<code>expires_at</code>	service	Always the issuance time plus 10 seconds
<code>finalized_chain_height</code>	service	The finalised chain height, read from authenticated state when the challenge is issued
<code>issuer_policy</code>	you	An optional list of accepted issuers, plus the minimum issuer state (<i>active</i> by default)
<code>claim_policy</code>	you	One or more predicates, each a claim, a value and an operator: <code>>=</code> , <code><=</code> or <code>==</code>
<code>same_credential_required</code>	you	Whether all predicates must be proven from a single credential (default: yes)

The identity fields come from deployment configuration linked to your on-chain registrations. HTTP callers can never set them, so a compromised web tier cannot get your service to sign for another origin or another key.

The acceptance conditions

The pipeline accepts a presentation only when all nine conditions hold. It checks them in order and rejects at the first failure:

1. your verifier record and the exact origin are active in authenticated chain state;
2. the challenge's request key is registered and active for that origin;
3. the nonce has never been used;
4. the response arrived before the ten-second deadline;
5. the proof is bound to the origin, verifier identifier, request key, nonce, policy and chain height;
6. every requested predicate is proven;
7. the issuer is active and allowed by your issuer policy, and its status root is current;
8. chain state is re-checked immediately before the decision;
9. the nonce is consumed permanently in your own durable store, and any concurrent attempt with the same nonce fails.

Binding is total. The proof commits to the whole signed challenge. If any part of it is changed, whether the policy, the origin or the chain height, the proof no longer verifies.

Freshness is enforced. Issuers publish a status root every 30 seconds. A presentation is rejected if the root it was built against is older than the maximum root age (45 seconds at the reference values; a deployment can govern a longer one), was replaced longer ago than your height-lag window allows or by an emergency root, or if the issuer is suspended. Every decision reads and proves the current chain state; if the chain cannot be read, the decision is a refusal, never a decision on a root kept from an earlier read. When the decision is made, the challenge's chain height must still be inside your staleness window. Stale state always leads to rejection.

What you get back

The decision has three parts: `accepted`, a `reason_code` (chapter 6) and, only when the presentation is accepted, `verified`. `verified` holds the `issuer_id`, the `status_root` and the `proven_predicates`. That is everything the verifier learns. You never receive the exact grade, the expiry date, the credential identifier or any attribute.

05

The public service contract

The HTTP interface of service mode is a versioned, published OpenAPI document, and that contract is the product.

The **MintID Verifier Service** contract is an OpenAPI 3.1 document, currently at version 1.5.0 and frozen at major version 1. It is published at mintid.net/openapi/verifier-service.v1.yaml. Its operations, schemas and reason codes are binding for integrators; this chapter is a guide to it and does not replace it.

OPERATION	METHOD	PATH	RESPONSES
<code>getBuildInfo</code>	GET	<code>/v1/build-info</code>	200: the release pins of the running artefact (never secrets)
<code>createChallenge</code>	POST	<code>/v1/challenges</code>	201: a signed challenge · 400: malformed body · 503: authenticated state unavailable
<code>validatePresentation</code>	POST	<code>/v1/presentations</code>	200: a decision, whether accept or reject · 400: malformed body
<code>checkStatusRoot</code>	POST	<code>/v1/status-root-checks</code>	200: whether a status root is still the issuer's current, unexpired one, from authenticated state · 400 · 503

A challenge for an agent names the agent profile; a policy the service will not issue for an agent is answered 422 (`policy_refused`) with the reason.

Creating a challenge

The request body carries policy only, never identity. A minimal example:

```
{
  "claim_policy": [{ "claim": "age_over", "value": 18, "operator": ">=" }],
  "issuer_policy": { "issuer_state_minimum": "ACTIVE" },
  "same_credential_required": true
}
```

The claim name above is only an example; the list of claims is published with the credential profile. The response contains `request` (the full signed challenge body described in chapter 4) and `signature_hex` (the signature over its canonical form). Forward both to the holder unchanged, and keep `request`, because you will send it back when you validate.

Validating a presentation

The request body contains the `request` you issued and the holder's `envelope`. The envelope carries the opaque proof bytes (`proof_hex`), the `issuer_id`, the `status_root` and the `claimed_predicates`, and never any holder attribute. The proof is checked by the pinned proof engine; a rejection looks like this:

```
{ "accepted": false, "reason_code": "proof_invalid", "verified": null }
```

Decisions are data, not errors

- Any validation request that can be parsed returns **HTTP 200 with a decision**, whether accept or reject, and exactly one decision record is written on the server.
- HTTP 400** (`malformed_body`) is returned only when the body cannot be parsed into the schema. In that case no session exists, so no record can be written.
- HTTP 503** (`state_unavailable`) means that the service could not read authenticated chain state when issuing a challenge, so it issued none. Retry later, and never work around it.
- Clients raise errors only for transport failures. Base your logic on `accepted` and `reason_code`, not on exceptions.

Deliberately absent. The contract has no lifetime setting, because ten seconds is a protocol constant. It has no flag to skip or relax a condition, and no per-request staleness window. As a result, a compromised web tier cannot widen what your verifier accepts.

Build information. GET `/v1/build-info` returns `sdk_version`, `engine_pin`, `state_vectors_version`, `state_vectors_sha256`, `spec_revision` and `conformance_suite_version`. Together they identify the SDK, the pinned proof engine, the chain-state test vectors and the specification revision the artefact was built against, and the conformance suite it passes. Check `engine_pin` in your deployment pipeline: it names the exact proof engine compiled into the artefact, and two deployments that disagree on it are not interchangeable.

06

Reason codes

The list of reason codes is closed and identical across the SDK, the service contract and every client.

CODE	COND.	MEANING	TYPICAL HANDLING
<code>accepted</code>	none	All nine conditions passed, and <code>verified</code> is present	Proceed
<code>request_malformed</code>	before 1	The challenge body does not have the required shape (for example, an empty claim policy)	Fix how the challenge is built
<code>verifier_not_active</code>	1	Your registration is not active in authenticated state	Check your registry state

CODE	COND.	MEANING	TYPICAL HANDLING
<code>origin_not_registered</code>	1	The origin does not resolve to your verifier	Register or renew the exact origin
<code>revalidation_expired</code>	1	The domain-control proof for the origin has lapsed	Renew domain control
<code>request_key_not_active</code>	2	The signing key is not registered or has been rotated out	Fix key registration or rotation
<code>nonce_replayed</code>	3	The challenge's nonce has already been used	Issue a fresh challenge and investigate
<code>deadline_exceeded</code>	4	The response arrived at or after the ten-second deadline	Issue a fresh challenge
<code>proof_invalid</code>	5	The proof failed, or it is bound to a different context	Reject; the request may have been tampered with
<code>predicate_missing</code>	6	A requested predicate is not among those proven	Reject
<code>issuer_not_accepted</code>	7	The issuer is outside your accepted-issuer policy	This follows from your own policy
<code>issuer_not_active</code>	7	The issuer is not operational, or is below your minimum state	Reject; a later retry is fine
<code>status_root_stale</code>	7	The proof was built on a root that has been replaced or has expired	The holder retries with a fresh proof
<code>height_stale</code>	8	The challenge's chain height fell outside your staleness window	Issue a fresh challenge
<code>state_unproven</code>	1 or 8	A chain read could not be proven, including a claim that a record does not exist	Treat as an operational alert; never bypass
<code>nonce_race_lost</code>	9	A concurrent presentation used the nonce first	Reject; only one attempt can succeed

Reserved codes. `origin_invalid` and `domain_validation_failed` are part of the list, but the current pipeline never returns them. If you receive one, treat it like any other rejection.

The one policy choice in the list. Under the default issuer minimum (`ACTIVE`), a presentation from an issuer under enhanced monitoring is rejected with `issuer_not_active`. If you set the minimum to `ENHANCED_MONITORING`, such presentations are accepted. Any change to the list of codes counts as a major contract version.

07

Privacy, retention and operations

What you keep is fixed by design. How you run the service is your responsibility as an accountable verifier.

The decision record

The service keeps **exactly one record per decision**, whether it accepts or rejects. The record holds five things: the session identifier, the outcome, the reason code, a 32-byte digest of the context the proof was bound to, and the time of the decision. Proof bytes, claim values and personal data have no place in the record. Presentations are never stored, forwarded, logged, broadcast or written to the chain. Extending the record needs a privacy review, not a configuration change. Keeping more than the record would require an explicitly approved legal retention policy.

Running your service

- **Deploy inside your own perimeter.** Your service instance is part of your verifier. Put your own web tier in front of it, and point your clients at it and at nothing else.
- **Protect your keys.** Request-signing keys are your responsibility: keep them in hardware-backed signers, on the server only.
- **Make storage durable.** The service holds no state of its own. It relies on your single-use store and your decision log, and both must survive a crash.
- **Alert on unproven state.** Treat HTTP 503 and `state_unproven` as operational alerts about a node, the trust anchor or the network. They are not integration bugs to work around.
- **Watch your registration.** Track when revalidation is due, the state of your keys and your authority state, and keep your clock accurate.
- **Design for retries.** Verification fails closed on purpose. If an issuer stops publishing, its credentials stop verifying within 45 seconds, so your user flows should let people retry calmly.

Accountability. Abuse by a verifier leads to suspension of its registration and, if a verifier bond is adopted, to adjudication of that bond. Verifiers are covered by a quarterly privacy and domain-control review, and the public transparency report lists verifier states and council actions, never holder data.

08

Agent payments: x402 and Web Bot Auth

The same verifier machinery serves agent commerce through a transport profile, with no change to the protocol.

In x402, a seller that charges per request already relies on a **facilitator** to verify and settle payments. That facilitator is the natural place to check identity too:

- The seller's payment-required response can state an identity requirement. The buying agent gets a short-lived challenge from the facilitator, **which acts as the registered verifier**, and sends its presentation along with its payment.

- The presentation binds to the facilitator’s origin, and any seller-specific requirements go in the claim policy. The seller relies on the facilitator’s decision in the same way it already relies on the facilitator for payment.
- Presentations expire, so a permanent “verified” badge cannot exist. An explorer that wants to show one registers as a verifier, re-verifies on a published schedule and publishes **its own signed verdict**.

LEVEL	WHAT THE COUNTERPARTY LEARNS
Anonymous	Payment only
Web Bot Auth	The software is identified through signed HTTP requests
Web Bot Auth + MintID	Identified software with an accountable, KYC-verified human behind it, without learning who that human is; the bot’s signing key can be bound into the agent credential

The classical seller variant. In the first version, a seller’s accountability is certified with conventional credentials signed by an issuer, and read back from authenticated chain state as a signed standing verdict. **Nothing in this exchange is a presentation, and it makes no zero-knowledge claim.** It will be replaced, not extended, by the zero-knowledge version. Agent credentials on the buyer side are zero-knowledge presentations under the agent profile, and there is no interim mode without zero knowledge. The profile is a draft; it will be frozen at the identity-only launch, and its dispute annex is reserved for later.

ZadQ, the anchor product. ZadQ is an x402 seller-verification service and the first product built on MintID. It is operated by the CONECTIA group and disclosed as a related-party position. It runs on the **single-node production chain**, operated by the CONECTIA group. That chain is not yet the distributed network, and nothing measured on it counts towards the traction criterion. Explorers and buying agents use its signal for free, and it has no protocol privilege.

Bounded agents and policy presets. A *bounded agent* (ADR-0023) is an agent credential minted from a named bundle of its issuer’s catalogue, which bounds every axis of the delegation (actions, categories, regions, sellers, value per transaction, lease, key custody, backing grade) and is always private-only, so the agent discloses nothing about who stands behind it. Instead of writing the agent policy by hand, you build it from the bundle with a **policy preset** in the SDKs: it covers every axis, includes the mandatory pseudonym and expiry entries, and refuses to ask for anything the bundle cannot grant. A preset is a library helper, not a contract change: the service validates its output like any other policy. Each issuer’s catalogue is its operator’s own (MintID ships a reference catalogue as an example): obtain the catalogue file and its digest from the operator, and the SDK builds presets from it after checking the content against that digest.

Disclosure releases: built, switched off. Specification SPEC-3 lets identity detail reach a relying party in two bounded ways, both decided by the agent’s issuer and never by the agent: a **consented disclosure** under the principal’s policy or live approval, and a **mandated identification**, where a counterparty that the law obliges to record its client (accommodation, vehicle rental, passenger transport) receives exactly the fields a governed **obligation class** names, under a mandate the principal gave, capped, expiring and revocable. You never choose the fields. The issuer delivers a signed release straight to an inbox on your own service, never through the agent; the inbox reads it only against an offer you sent and still pending, and treats anything more or less than was asked as no disclosure. What you receive is your own record, under your own controllership. Both ends ship **switched off and fail-closed**: the issuer side and your inbox answer as if they did not exist until enabled, activation waits for counsel per jurisdiction, and one proof the rail needs is still to be added to the proof core. The inbox is described by its own additive document beside the frozen contract, which it does not change.



Premium services. Paid verification services, mirrors and archives earn direct fees, which are kept separate from consensus rewards. Basic verification remains free.

09

Conformance, versioning and status

Every SDK release must pass the same conformance tests, and the status of each component is stated plainly.

Versioning and conformance

- **Semantic versioning applies to the contract.** A change to decision behaviour or to the list of reason codes is a major version. New additions are a minor version: 1.1.0 added the conformance-suite version to the build information; 1.2.0 to 1.5.0 added the agent profile, its pairwise pseudonym (mandatory since 1.4.0), the mandatory `unexpired` entry and the status-root check, leaving identity challenges byte-for-byte unchanged. Everything else is a patch.
- **One shared conformance suite gates every release.** It has one set of test cases (challenge, envelope, expected decision, reason code) for all languages. It covers an accepted case, a rejection for each acceptance condition and every reason code the pipeline can return.
- **Equivalent builds decide identically.** Two artefacts with the same contract major version and the same conformance-suite version make the same decisions, whatever their language.

What exists today

COMPONENT	STATUS	WHAT IT MEANS FOR YOU
Verifier service and contract	Implemented	The full pipeline, single-use challenges, authenticated chain reads, decision records and the versioned public contract
Proof engine	In production	The pinned proof core checks every presentation; a failing proof returns <code>proof_invalid</code> . You can also build and test an integration against test doubles
Client SDKs	First client available	A TypeScript client, agent clients (Python and TypeScript) and an agent-tooling integration that exposes the contract's three operations and nothing else; bounded-agent policy presets in Python and TypeScript; other languages as demand appears
Consented commercial disclosure rail	Built, switched off	The release inbox and the issuer side ship disabled and fail-closed; activation waits for counsel
Standing read	Implemented	The classical read of a seller's standing used by ZadQ's first version; it is not a presentation
Holder side	Headless holder library	A holder library issues, presents and refreshes witnesses over the proof core; there is no consumer wallet, and none is planned
Networks	Single-node production	The single-node production chain, in production since 20 September 2026 and operated by the CONECTIA group, carrying ZadQ; plus the public testnet, with its KYC sandbox, where third parties test and nothing issued carries weight (chapter 10). The distributed mainnet has not launched: there is no token genesis and no public validator set

Getting access and the public testnet

Test on the public testnet, with its KYC sandbox; the single-node production chain is never a place to test. A public adversarial testnet with independent participants will run before any mainnet.

The public testnet and its KYC sandbox

The public testnet is the only MintID deployment a third party may use to test. It runs the same software and the same chain parameters as the single-node production chain, on its own chain, with its own keys and hostnames. Its issuer verifies nobody: when you open a KYC session you choose its outcome, so every path of issuance and every error a verifier can meet is reachable on purpose: an approval at the grade you choose, a refusal, an expiry, a manual review, an unavailable provider, a reuse window that has closed. Its sandbox verifier answers challenges and validates presentations, and the testnet runs holder self-revocation and the privacy relay as production does.

Nothing issued on the testnet carries weight: a credential, a grade, an agent or a decision obtained there says nothing about any person and must never be accepted anywhere else. It has one validator, run by the operator of the production chain; it runs on its own test token, free to anyone from an open faucet, which has no value and converts into nothing; and it is reset when the software changes shape, with everything issued before gone.

Its status, its addresses, its limits and the full table of sandbox outcomes are on one page of the developer documentation, *The public testnet and its KYC sandbox* ([docs/sdk/17-sandbox-and-testnet.md](#) in the main repository). That page is the only place that states whether the testnet is live.

What you can do there: run the whole journey with public access only — obtain a sandbox principal, issue an identity credential and agents, present to the sandbox verifier, exercise every rejection path, revoke by every path, an agent's kill switch included, and read the decision evidence. The testnet offers no registration path for a verifier of your own, so its decisions are always the sandbox verifier's. To run your own verifier end to end, the local stack of the main repository runs every service on your own machine, where you hold every key.

The single-node production chain

The single-node production chain answers three public read endpoints — the chain's status, the issuer's build information and the verifier's status and build information — so anyone can check that blocks are produced, which proof engine the verifier is pinned to and whether the issuer's root is fresh. They are evidence, not a place to test. Your clients should always point at your own service instance. Accepting real presentations there requires a registered verifier (Appendix A), admitted by governance proposal until self-service registration reaches that chain; it starts as a conversation through the contact page on [mintid.net](#) or at info@mintid.net. Tell us what you are building and which mode you plan to use.

The public adversarial testnet

Before any mainnet, MintID will run a **public adversarial testnet** with independent validators, issuers and verifiers; it is a different network from the public testnet above, which has one validator and a sandbox issuer. It is one of the committed milestones before a public network, alongside independent audits published with their findings, a funded bug bounty and an incident-response exercise. For verifiers it will offer:

- the real registration path: KYC with an accepted issuer, registered keys and origins, domain-control proofs and approval;

- presentations from real holder clients;
- adversarial testing of the presentation path, together with other independent participants;
- test tokens, which are worthless by design and never convert into anything.

The path to production. MintID launches in stages, each opening only when its published conditions are met. After today's single-node production stage, the identity-only mainnet (genesis targeted for the second half of 2027 or 2028) brings issuance, zero-knowledge presentations, status roots, free verification, the verifier registry and a working council. All dates are targets, not commitments.

APPENDIX A

Go-live checklist

Everything that must be true before your first real presentation.

AREA	CHECK
Registration	Verifier KYC completed with an accepted issuer; admission granted by governance proposal (the council's once seated); authority state <i>active</i>
Origins	Every origin you present from registered exactly, in canonical form; domain-control proofs current; revalidation dates monitored
Keys	Request-signing key registered and active for each origin; keys held in hardware-backed signers on the server; rotation planned, with holder re-consent in mind
Deployment	Your own service instance (or the embedded SDK) inside your perimeter; identity set in deployment configuration; clients pointed only at your instance
Chain access	Nodes chosen, a trust anchor you control, and a staleness window set at deployment
Storage	A single-use store that is durable, atomic and survives crashes; a decision log that holds the minimal record and nothing else
Operations	Accurate clock; alerts on HTTP 503 and <code>state_unproven</code> ; <code>engine_pin</code> and <code>conformance_suite_version</code> checked in your deployment pipeline
Product	Claim policy limited to the claims you are approved to request; logic based on <code>accepted</code> and <code>reason_code</code> ; user flows that allow a retry
Privacy	No presentation stored, forwarded or logged; any extension of the record sent for privacy review

APPENDIX B

Core requirements that shape a verifier

The requirements from the MintID core specification that a verifier meets most directly.

ID	IN ONE LINE	ID	IN ONE LINE
R2	No raw KYC, credential serial or presentation log on-chain, ever	R18	Presentations stay off-chain; you keep only the minimal decision record
R6	Grades are private: you receive <i>grade</i> ≥ A3, never the exact grade	R21	Selective disclosure: you receive predicate results, not attributes
R9	Issuers publish status every 30 seconds; stale status is rejected	R22	One ten-second session can prove several predicates
R13	Verifiers are registered, KYC-approved and prove control of every origin and key	R23	Holders may pin your identifier, origin and key fingerprint; changes need their re-consent
R14	Presentations live ten seconds, fully bound to your challenge	R24	Exact origins only, no wildcards, periodic domain-control revalidation
R15	Verification runs online against the latest finalised state	R26	Paid verification services are allowed, kept apart from consensus rewards; basic verification stays free

MintID Verifier & SDK Manual v0.2, September 2026 · prepared by the MintID Foundation, a Netherlands Stichting · non-normative; the specification corpus and the published service contract govern · not legal advice · not a MiCA crypto-asset white paper and not an offer of any crypto-asset.