Nightshade: A Privacy-Preserving Payments, Identity, and Services Protocol for Autonomous Agents

---

Version 1.0-draft The Nightshade Contributors 2026-05-25

> ***Alignment note (2026-07-07 — read this first).*** *This paper is the 2026-05-25 design snapshot. The protocol has since been* ***built*** *(the full register → discover → hire → pay → audit loop runs on test networks; ~594 automated tests; two settlement rails proven on-chain) and four recorded decisions revise claims made below — the decision log (`DECISIONS.md`) and the current guides (`docs/`) take precedence wherever they disagree:*
>
> - ***D-014 (pluggable settlement) + D-019 (USDC default):*** *settlement is a configuration choice behind a rail-neutral §33.3 `PaymentCommitment`.* ***USDC on EVM chains is the default rail****, including the §15/§17.5 mint fee; Monero remains a fully supported optional backend for deployments that want chain-level payment privacy. Every "settlement occurs on Monero" statement below reads as "on the deployment's configured rail".*
> - ***D-018 (Tor removed):*** *the implementation ships no Tor and no privacy claim rests on transport. Network-metadata privacy is out of v1 scope; every "over Tor" phrase below is illustrative of a deployment, not a requirement.*
> - ***D-017 (selective disclosure, built):*** *the graduated disclosure lever sketched here now exists in full — per-agent view-key disclosure, scoped audit sub-keys, and full-log disclosure, plus an independent key-free network auditor.*
> - ***Token sections:*** *design exploration under legal review; nothing is an offering. The agent payment path involves no token.*
>
> *The revised whitepaper exists:* ***`khorr-whitepaper-2026-07-15.md`*** *(the as-built, trading-first paper) supersedes this one as the current description of the system — and the project has since been renamed* ***Nightshade*** *→* ***Khorr*** *(July 2026). This paper remains as the historical design snapshot under its original name.*

Abstract

This paper describes Nightshade, a protocol and software development kit that enables autonomous AI agents to register identities, discover one another, establish authenticated sessions, and exchange value for services with strong privacy guarantees. Settlement occurs on Monero; no layer above the chain weakens Monero's payment privacy. Identities are derived hierarchically from an operator-held principal seed using HKDF-SHA-512 with structured domain separation, producing mid-lived agent identities and ephemeral session identities. Agent registration is gated by a blind-signature credential issued by a centralized mint authority under RFC 9474 RSABSSA-SHA384-PSS-Randomized, with the mint fee paid out-of-band on Monero such that the blinded credential request is unlinkable to the payment that funded it. A public registry of registered agents and their capabilities is maintained by a three-tier set of replicas — active replicas providing 3-of-N quorum attestations, server-class shadow attestors providing parallel divergence detection, and mobile attestors providing broad participation through a tap-to-attest user interface. The registry tolerates Byzantine fault up to one less than the quorum threshold and is independently verifiable by clients via a Foundation-signed manifest, an append-only transparency log, and a sub-five-minute emergency revocation overlay. Agent-to-agent service payments occur via the spec §33 PaymentCommitment flow over Monero transactions, with the session-binding handshake (spec §22) preventing replay, reflection, and impersonation attacks. A native token provides the bond required for replica operation and the burn instrument for marketplace-layer features; its emission schedule follows a Bitcoin-style halving curve, with rewards distributed to the three replica tiers and paired with a fixed 1% share of the registration mint fee for active replicas. The protocol includes a decentralized adversarial adjudication mechanism — counterparties who have been harmed file evidence-bearing reports, randomly-selected jurors from the replica and shadow attestor pool vote with stake at risk, and verdicts are graduated from warnings through marketplace delisting to emergency revocation; operator-level recidivism is exposed through a derived pseudonym scheme without breaking blind-credential unlinkability. The protocol's complete byte layouts, cryptographic primitives, and reference implementation in Rust are documented at the protocol specification level.

---

1. Introduction

1.1 Problem statement

Autonomous AI agents that participate in economic activity require payment infrastructure to compensate one another for services rendered. Existing protocols designed for this purpose — x402, Skyfire, Nevermined, and others — settle on transparent public blockchains. Every payment between agents, every service consumed, and every operator-side accounting flow is permanently visible to any party with access to a block explorer.

As autonomous agents become substantive economic actors, transparent settlement creates a surveillance surface without precedent in financial history. Every API call an agent makes, every model inference it requests, every retrieval query it performs, and every output it produces can be permanently linked to a public address. Operators of high-value agents are direct targets; operators of any agent at all become targets at scale.

The privacy guarantees humans have long expected for personal financial activity — and that Monero specifically provides — should extend to agent-to-agent commerce. Nightshade is the protocol layer that

delivers this.

1.2 Design principles

- **Monero-grade payment privacy.** Settlement occurs on Monero. No layer above the chain weakens Monero's payment-time privacy.
- **Operator-controlled identity.** Each operator's principal seed is the root of an identity tree from which all derived agent and session identities can be reconstructed deterministically.
- **Blind credential issuance.** Registration consumes a blinded credential that the mint authority signs without learning what is being registered. The fee that funds the credential is paid out-of-band; the credential request and the fee transaction are not linkable through the mint authority's records.
- **Replicated trust.** No single entity controls the registry. A quorum of replicas independently signs each snapshot of the agent directory; clients require independent attestations from a threshold of the active replicas before accepting any snapshot.
- **Crypto-agility hooks for v2.** Identity derivation, signature, and AEAD primitives expose extension points such that post-quantum migration is mechanically possible without spec-level restructure.
- **Honest threat modeling.** What the protocol protects against is precisely enumerated; what it does not protect against is precisely enumerated.

1.3 Document organization

Section 2 describes the protocol architecture. Sections 3–8 describe the protocol components in detail: identity, canonical serialization, credential issuance, the registry, session-binding handshake, and the Monero payment layer. Section 9 describes the replica system, including the three-tier role structure and manifest mechanism. Section 10 describes the marketplace. Section 11 describes the tokenomics and economic model. Section 12 describes the security model and threat assumptions. Section 13 describes privacy properties. Section 14 describes the adversarial adjudication mechanism for handling agents used for harm. Section 15 describes the reference implementation. Section 16 describes the roadmap. Section 17 lists open questions deferred to future work.

---

2. Architecture

The Nightshade protocol decomposes into seven layers, each with a defined responsibility and byte format. Higher layers depend on the cryptographic guarantees of lower layers. The lower layers have reference Rust implementations and normative test-vector suites today; the application layers (payment, marketplace) land across the current build phases.

```
+-------------------------------------------------------------+
| Layer 7: Marketplace                                        |
|   Capability schemas, discovery, dispute mechanisms         |
+-------------------------------------------------------------+
```

```
| Layer 6: Payment                                                |
|   PaymentCommitment, sub-agent composition, spending policy     |
+-----------------------------------------------------------------+
| Layer 5: Session-binding handshake                              |
|   CounterpartyChallenge, SessionAuthorization, PossessionProof  |
+-----------------------------------------------------------------+
| Layer 4: Registry                                               |
|   Snapshots, replica quorum, transparency log, emergency        |
|   overlay, three-tier attestation                               |
+-----------------------------------------------------------------+
| Layer 3: Credential issuance                                    |
|   RFC 9474 RSABSSA blind credentials                            |
+-----------------------------------------------------------------+
| Layer 2: Identity                                               |
|   Principal -> Agent -> Session HKDF chain, audit log AEAD      |
+-----------------------------------------------------------------+
| Layer 1: Canonical serialization                                |
|   Byte-deterministic encoding, domain-separated hashing         |
+-----------------------------------------------------------------+
```

Each operator runs an SDK that exposes APIs for principal management, agent derivation, session establishment, payment construction and verification, registry queries, and audit-log reading. Server-side roles — mint authority, replicas of all three tiers, and the §21 auditor — run dedicated binaries that implement the relevant protocol responsibilities.

Settlement occurs on the unmodified Monero network. Nightshade is a consumer of Monero, not a modifier of it; the Monero protocol is invoked through its official `monerod` and `monero-wallet-rpc` daemons.

---

3. Identity

3.1 Three-tier identity model

Each operator maintains a 32-byte principal seed, generated from an OS-provided CSPRNG at first use and stored encrypted at rest. From the principal seed, the operator deterministically derives:

- A **principal Ed25519 signing keypair** for operator-side signatures (audit log signatures, registration submissions).
- An arbitrary number of **agent identities**, each derived from the principal seed using a 16-byte salt and a hash of the agent's configuration. Each agent has its own Ed25519 signing key, its own 16-byte revocation tag, and is registered in the public registry.
- An arbitrary number of **session identities**, each derived from an agent's seed using a 32-byte nonce. Each session has its own Ed25519 signing key; sessions are single-use and ephemeral.

The derivation function is HKDF-SHA-512 with structured `info` buffers that include algorithm tag, salt, and (where applicable) the canonical encoding of the parent identity's configuration. All HKDF salts are byte

strings beginning with the domain tag `nightshade-id-v1:`.

```
principal_seed       <- CSPRNG(32 bytes)
principal_ed25519    <- HKDF-SHA-512(salt=nightshade-id-v1:principal-derive, ikm=principal_seed, inf

agent_seed           <- HKDF-SHA-512(salt=nightshade-id-v1:agent-derive,
                                     ikm=principal_seed,
                                     info=[alg, salt, config_hash]) [32 bytes]
agent_ed25519        <- Ed25519::from_seed(agent_seed)
agent_revocation_tag <- HKDF-SHA-512(salt=nightshade-id-v1:revocation-tag,
                                     ikm=agent_seed, info=[alg]) [16 bytes truncated]

session_seed         <- HKDF-SHA-512(salt=nightshade-id-v1:session-derive,
                                     ikm=agent_seed, info=[alg, nonce]) [32 bytes]
session_ed25519      <- Ed25519::from_seed(session_seed)
```

Each derivation chain enforces a forward-independence property: knowledge of an agent's `agent_seed` does not reveal `principal_seed`; knowledge of a session's `session_seed` does not reveal `agent_seed`.

3.2 Verifying-key encoding

Every Ed25519 verifying key in the protocol is encoded with a length-prefixed algorithm tag per §14.2 of the protocol specification:

```
encode_verifying_key(alg: u8, vk: [u8; 32]) -> [u8; 35]
   = alg || u16_be(32) || vk
```

This format permits future post-quantum key types to coexist with v1 Ed25519 keys without breaking message structure.

3.3 Audit log

The operator-local audit log records each principal-, agent-, and session-level action for after-the-fact auditing. Records are encrypted at rest using XChaCha20-Poly1305 under an `audit_key` derived from the principal seed:

```
audit_key <- HKDF-SHA-512(salt=nightshade-id-v1:audit-key, ikm=principal_seed, info=[alg])
```

Audit log records form an internal hash chain via `prev_record_hash`; external rollback detection is provided by storing the latest `(last_seq, last_record_hash, file_id)` triple in the operator's key-storage hardware (e.g., a YubiKey). On audit-log open, the SDK verifies the chain against the stored high-water mark; mismatch indicates either rollback or external compromise.

---

4. Canonical serialization

All protocol messages whose bytes are hashed, signed, or compared cross-implementation are encoded using the Nightshade canonical serialization format defined in protocol specification Part II §5–§9.

4.1 Type system

The canonical format supports six value types: `null`, `bool`, `int` (signed, fits in `i64`), `bytes`, `string` (NFC-normalized UTF-8), `array`, and `object` (string-keyed map). Each value is encoded with a single-byte type tag followed by a type-specific payload.

4.2 Determinism

Encoding is deterministic: for any value, there is exactly one canonical byte string. This is achieved by:

- Object keys are sorted by their encoded-byte form (memcmp ordering); duplicates after NFC normalization are rejected.
- Strings are NFC-normalized; non-NFC inputs are rejected by the decoder.
- Integer encoding uses minimum-length form.
- Byte and string lengths use minimum-length varint encoding.

The protocol rejects any encoding that violates these rules. The decoder is defensive: malformed input is rejected without panic.

4.3 Domain-separated hashing

Bytes that are hashed for protocol use are prefixed with a domain tag and a zero byte separator before being passed to SHA-256:

```
domain_hash(tag: &str, payload: &[u8]) -> [u8; 32]
   = SHA-256(tag.bytes || 0x00 || payload)
```

Domain tags are unique per use site: `nightshade-snapshot-v1`, `nightshade-emergency-overlay-v1`, `nightshade-transparency-v1`, `nightshade-capability-v1`, `nightshade-challenge-v1`, `nightshade-transcript-v1`, `nightshade-session-binding-v1`, `nightshade-session-possession-v1`, and others. Cross-tag collisions are computationally infeasible.

---

5. Credential issuance

5.1 Mint authority

A centralized mint authority is responsible for issuing the blind credentials that gate agent registration. The mint runs an RFC 9474 RSABSSA-SHA384-PSS-Randomized signature scheme over a 4096-bit RSA keypair generated through an offline ceremony with an external witness.

The mint authority's public key is pinned in the Nightshade SDK at v1 release. Clients verify all mint signatures against the pinned key.

5.2 Mint fee

To obtain a credential, the operator pays a fixed fee of **0.027182 XMR** to the mint's primary address. The fee is paid out-of-band — the Monero transaction occurs separately from the credential request. The mint cannot link the on-chain payment to the subsequently-submitted blinded credential request, since the credential is blinded before submission and contains no identifying information traceable to the payment.

5.3 Blind credential protocol

The client-side credential protocol follows RFC 9474:

1. **Construct credential payload.** The client builds a canonical-encoded payload containing the agent verifying key, validity epoch, and a fresh 32-byte nonce.
2. **Prepare.** The client computes `prepared_message = randomizer || SHA-256(payload_bytes)`, where `randomizer` is a fresh 32-byte CSPRNG output.
3. **Blind.** The client computes the blinded message using the mint's public key and a fresh blinding factor.
4. **Pay.** The client pays the §17 fee out-of-band via Monero transaction.
5. **Submit.** The client submits the blinded message + the operator-pseudonymous submission token over Tor to the mint's `BlindSign` endpoint.
6. **Sign (server-side).** The mint verifies the fee transaction is confirmed (≥10 confirmations) at the operator's claimed `tx_hash`; consults its idempotency table; signs the blinded message; stores the result in the idempotency table.
7. **Finalize.** The client unblinds the mint's signature using its blinding factor, producing a normal RSA-PSS signature on the prepared message.
8. **Verify.** The client (and any party presented the credential later) verifies the signature against the mint's public key AND verifies the byte-level prepared-message structure: `prepared_message.len() == 64 && prepared_message[32..] == SHA-256(payload)`. Both checks are normative.

5.4 Issuance log

The mint publishes an append-only issuance log per spec §17.9. Each log entry records issuance metadata (epoch, fee amount, confirmation block height, operator pseudonym redacted) sufficient for the §21 auditor to cross-reference against on-chain receipts. The mint fee is uncapped — registration always requires fee payment, paid in XMR and never touching token markets.

---

6. The registry

6.1 Capability listing

Each registered agent advertises one or more capabilities — services it offers and the price for using them. A capability listing has ten fields per spec §18.1:

```
struct CapabilityListing {
   capability_hash:    [u8; 32],   // SHA-256(canonical_schema) under nightshade-capability-v1
   price_per_call:     u64,        // Monero atomic units
   price_per_token:    u64,        // Monero atomic units, 0 if not applicable
   timeout_seconds:    u64,        // Service timeout
   refund_policy:      u8,         // None | OnTimeout | OnDispute | Both
   sla_class:          u8,         // BestEffort | Standard | Premium
   input_schema_hash:  [u8; 32],   // SHA-256(canonical_input_schema)
   output_schema_hash: [u8; 32],   // SHA-256(canonical_output_schema)
   endpoint:           String,     // Tor v3 onion address or clearnet URL
   caveats:            Option<Object>,  // Operator-specific qualifications
}
```

6.2 Agent entry

An agent's public registry entry contains six fields per spec §18.3:

```
struct AgentEntry {
   version:               u8 = 1,
   registration_epoch:    u32,
   valid_through_epoch:   u32,    // Default: registration_epoch + 52
   agent_vk:              VerifyingKey,  // §14.2-encoded
   revocation_tag:        [u8; 16],
   capabilities:          Vec<CapabilityListing>,  // canonically sorted
}
```

Capability listings within an `AgentEntry` are sorted canonically. The agent_vk is encoded in the §14.2 length-prefixed form to permit post-quantum key types in future versions without rebuilding the entry structure.

6.3 Snapshot construction

Per epoch (7 days, per spec §18.6), the registry is captured as a `SnapshotBody`:

```
struct SnapshotBody {
   version:                u8 = 1,
   epoch:                  u32,
   published_at_unix_seconds: u64,
   registry_challenge:     [u8; 32],   // CSPRNG nonce per epoch
   agent_entries:          Vec<AgentEntry>,    // sorted by canonical-encoded bytes
   revocations:            Vec<RevocationEntry>,  // sorted by revocation_tag
}

snapshot_hash = domain_hash("nightshade-snapshot-v1", canonical(SnapshotBody))
```

Active replicas independently and deterministically construct the same `SnapshotBody` from their gossip-converged set of accepted submissions, and sign the `snapshot_hash` with their Ed25519 keys.

6.4 Signed snapshot

The `SignedSnapshot` wraps the canonical body bytes verbatim alongside replica signatures:

```
struct SignedSnapshot {
   body_bytes:         Vec<u8>,          // verbatim canonical encoding
   replica_signatures: Vec<ReplicaSignature>,   // sorted by replica_id
}

struct ReplicaSignature {
   replica_id:       [u8; 32],
   signature_bytes:  Vec<u8>,
}
```

Body bytes are stored verbatim (rather than re-encoded from the parsed structure) to ensure the exact bytes signed by replicas are the exact bytes verified by clients.

6.5 Replica quorum

A client accepts a `SignedSnapshot` only if at least the threshold number of distinct active replicas' signatures verify against `snapshot_hash`. The threshold is derived from the manifest's `quorum_threshold` field, which for N active replicas is `floor(N/2) + 1`. With N=5 active replicas, the threshold is 3.

Unknown replica signatures (replicas not in the manifest) are silently skipped — they don't count toward quorum but don't cause failure either. This permits manifest expansion to happen incrementally.

6.6 Emergency revocation overlay

Some agent revocations cannot wait for the next epoch's snapshot. When an operator's agent key is suspected compromised, an emergency revocation can be propagated:

```
struct EmergencyOverlayBody {
   version:                u8 = 1,
   as_of_unix_seconds:     u64,
   valid_for_seconds:      u64 = 300,   // 5 minutes per spec §24.3
   entries:                Vec<EmergencyRevocation>,  // sorted by revocation_tag, no duplicates
}

overlay_hash = domain_hash("nightshade-emergency-overlay-v1", canonical(EmergencyOverlayBody))
```

The active replica set signs the overlay with the same 3-of-N quorum requirement. The overlay is valid for 300 seconds; the client fails closed when the overlay is absent or stale per the protocol's strict-mode default (operators may opt into a bounded-grace mode for low-risk services).

6.7 Transparency log

Each replica maintains an append-only chain of `TransparencyLogEntry` records per spec §19.3:

```
struct TransparencyLogEntryBody {
   replica_id:                  [u8; 32],
   epoch:                       u32,
   snapshot_hash:               [u8; 32],
   published_at_unix_seconds:   u64,
   prev_log_hash:               [u8; 32],
}

entry_hash = domain_hash("nightshade-transparency-v1", canonical(TransparencyLogEntryBody))
```

Each replica signs each of its own entries. Clients aggregate entries from multiple replicas attesting the same `(epoch, snapshot_hash)`. If a client receives `SignedSnapshot` X but the transparency-log-attested `snapshot_hash` at the same epoch differs from `snapshot_hash(X)`, this signals a split-view attack and the snapshot is rejected.

The `ReplicaQuorumTransparencyCheck` aggregator (reference implementation: `nightshade-registry`) requires at least 3 distinct replicas attesting the matching hash + epoch. Reaching the SDK requires this check to be wired into the registry-refresh path; the reference SDK does so by default.

---

7. Session-binding handshake

Before two agents transact, they establish a cryptographic session via the two-stage handshake of spec §22.

7.1 Stage 1 — Counterparty challenge

The counterparty (the prospective service consumer) publishes a `CounterpartyChallenge`:

```
struct CounterpartyChallenge {
   version:              i64 = 1,
   counterparty_alg:     u8 = 0x01,    // Ed25519
   counterparty_vk:      [u8; 32],
   capability_hash:      [u8; 32],     // The capability being requested
   challenge_nonce:      [u8; 32],     // Fresh CSPRNG nonce
   epoch:                i64,
   timestamp:            i64,
}
```

The challenge is canonically encoded and signed by the counterparty's session-handling key over the domain-tagged bytes:

```
signed_bytes = "nightshade-challenge-v1" || 0x00 || canonical_encode(challenge)
```

7.2 Stage 2 — Agent authorization

The agent (the prospective service provider) verifies the challenge, then constructs:

```
transcript_hash = domain_hash("nightshade-transcript-v1", canonical_encode(challenge))

struct SessionAuthorization {
   version:              i64 = 1,
   agent_alg:            u8 = 0x01,
   agent_vk:             [u8; 32],
   session_alg:          u8 = 0x01,
   session_vk:           [u8; 32],
   epoch:                i64,
   capability_hash:      [u8; 32],
   counterparty_alg:     u8 = 0x01,
   counterparty_vk:      [u8; 32],
   challenge_nonce:      [u8; 32],
   transcript_hash:      [u8; 32],
}
```

The agent signs this with its long-lived agent key. The signature commits the agent to a specific session_vk, counterparty, capability, nonce, and transcript.

7.3 Stage 3 — Session possession proof

The session, via the session signing key derived from the agent, signs:

```
struct SessionPossessionProof {
   version:              i64 = 1,
   session_alg:          u8 = 0x01,
   session_vk:           [u8; 32],
   transcript_hash:      [u8; 32],
```

```
   timestamp:            i64,
}
```

This proves the party presenting `session_vk` holds the corresponding signing key (not merely a leaked verifying key).

7.4 Counterparty verification

The counterparty verifies the full handshake per spec §22.5:

```
verify_handshake(
   expected_challenge,
   auth, auth_sig,
   proof, proof_sig,
   registry, replay_cache,
   counterparty_now,
   freshness_window_seconds,
) -> Result<(), HandshakeError>
```

Steps:

1. The authorization's bound fields (`challenge_nonce`, `capability_hash`, `counterparty_vk`, `epoch`) match the issued challenge.
2. The authorization's `transcript_hash` equals `domain_hash("nightshade-transcript-v1", canonical_encode(challenge))`.
3. The registry confirms `agent_vk` is registered + not revoked (snapshot revocation list check).
4. The emergency overlay confirms `agent_vk` is not emergency-revoked.
5. The authorization signature verifies against `agent_vk`.
6. The possession proof signature verifies against `session_vk`.
7. `auth.session_vk == proof.session_vk`.
8. The possession proof's timestamp is within `freshness_window_seconds` of `counterparty_now`.
9. The replay cache has not seen `(epoch, agent_vk, session_vk)` previously; recording succeeds on first valid observation.

Failure at any step rejects the handshake. Success establishes that the counterparty is talking to the specific agent claiming `agent_vk`, at the registered epoch, with a fresh and unique session key.

---

8. Payment

8.1 Three-wallet hierarchy

Each operator maintains three Monero wallets per spec §32:

- **Principal funding wallet** — cold storage, derived from the principal seed. Funds the other two via internal transfers. View-key operator-private.
- **Mint-fee wallet** — derived from the principal seed under a distinct HKDF salt. Pays the §17 mint fee. View-key disclosed to the §21 auditor.
- **Per-agent service wallet** — derived from the agent's seed. Receives payment for that agent's services. View-key operator-private.

Wallet keys are deterministically derived from the principal/agent seeds via spec §32.1–§32.2: `sc_reduce32(HKDF(...)) || Keccak-256(...) || scalar × Ed25519 basepoint`. The Rust SDK performs the derivation; the resulting spend + view keys are handed to a local `monero-wallet-rpc` daemon via `generate_from_keys` at wallet provisioning, then zeroized from Rust memory.

8.2 Payment commitment

For a service payment from agent A (consumer) to agent B (provider), agent A constructs a Monero transaction paying agent B's per-agent service wallet's primary address. A `PaymentCommitment` records the details for B to verify:

```
struct PaymentCommitment {
   tx_hash:        [u8; 32],
   tx_pub_key:     [u8; 32],     // R = r * G, where r is the transaction's secret tx key
   output_index:   u32,
   amount_atomic:  u64,           // Monero atomic units
   recipient_address: String,    // Agent B's per-agent service wallet primary address
}
```

A signs the PaymentCommitment with its session key, embedded in the response to B. B's wallet verifies via `nightshade-wallet::verify_incoming_payment`, which delegates to `monero-wallet-rpc::get_transfer_by_txid` for the cryptographic verification (stealth-address derivation, amount decryption, confirmation depth).

The result is a `VerifiedPayment` or one of four explicit failure modes: `TxNotFound`, `InsufficientConfirmations`, `InsufficientAmount`, `AmountMismatch`. Successful verification is sufficient for B to begin service delivery.

8.3 Sub-agent composition

An agent A hired by counterparty C can immediately hire sub-agents D, E, F to fulfill the request, paying them from A's per-agent service wallet. The protocol does not require C to learn that A sub-contracted; the only flow C sees is the §22 handshake with A and the §33 PaymentCommitment from A.

This composition is supported natively: spec §34 specifies the byte layout for `ServiceRequest` and `ServiceDelivery` messages that flow between agents, and the protocol's audit log (§25, §36) records the sub-agent chain in the operator's local records but not in any shared registry.

8.4 Spending policy

Each agent's `config_hash` (the canonical-encoded configuration baked into the agent at derivation time) can include spending policy parameters: per-request cap, per-epoch cap, and allowlist/denylist of recipient agent_vks. These are enforced by the SDK before constructing a payment transaction. Policy is operator-defined; the protocol enforces only that the policy's bytes are part of the agent's identity.

---

9. The replica system

9.1 Three tiers

The registry is maintained by three tiers of replica operators:

- **Active replicas** sign snapshots that count toward the 3-of-N quorum.
- **Server-class shadow attestors** independently run the protocol in parallel; their signatures provide divergence detection but do not count toward quorum.
- **Mobile attestors** participate through a tap-to-attest mobile application; each tap signs a parallel attestation of the current snapshot, rate-limited to one per attestation epoch per attestor.

Two distinct cadences operate here. The **registry epoch** (7 days, per spec §18.6) governs snapshot content, credential validity, and revocation absorption. The **attestation epoch** (provisionally 1 hour) is the faster cadence at which shadow and mobile attestors re-witness the current snapshot, and at which token emission accrues. Decoupling the two keeps attestor participation — and the geographic-liveness signal it provides — frequent, even though snapshot content only changes once per weekly registry epoch.

9.2 Signed manifest

The current membership of each tier is described in a Foundation-signed manifest:

```
struct ReplicaSetManifestBody {
   version:                    u32,
   published_at_unix_seconds:  u64,
   valid_through_unix_seconds: u64,
   active_replicas:            Vec<ActiveReplicaEntry>,         // sorted by replica_id
   server_shadow_attestors:    Vec<ServerShadowEntry>,           // sorted by attestor_id
   mobile_attestor_root:       [u8; 32],                         // Merkle root of mobile attestors
   quorum_threshold:           u32,                              // floor(active_replicas.len() / 2)
   min_active_bond:            u64,
   min_shadow_bond:            u64,
   min_mobile_bond:            u64,
```

```
   current_halving_epoch:      u32,
   base_reward_per_epoch:      u64,
}

struct SignedReplicaSetManifest {
   body_bytes:               Vec<u8>,
   authority_signature:      [u8; 64],
}

manifest_hash = domain_hash("nightshade-replica-manifest-v1", body_bytes)
```

Clients fetch the manifest from Foundation's published endpoint and verify the signature against Foundation's pinned public key. The mobile attestor set is committed via a Merkle root; individual mobile attestors are verified against the root using per-attestor Merkle proofs supplied with each attestation.

9.3 Active replica role

An active replica is a dedicated server that:

- Accepts operator submissions (registrations, revocations) over Tor and/or clearnet HTTPS.
- Validates submissions for canonical encoding, valid mint credential signature, and well-formed payload.
- Engages in anti-entropy gossip with peer replicas every 30 seconds, exchanging Merkle-tree-of-submission-identifiers diffs.
- Constructs the per-epoch `SnapshotBody` deterministically from the submission set converged by `epoch_start - 60s`.
- Signs the `snapshot_hash` with its Ed25519 key.
- Publishes the `SignedSnapshot` to its public endpoint(s).
- Receives emergency revocations and signs the `EmergencyOverlayBody` within the spec §24.4 5-minute target.
- Maintains its per-replica `TransparencyLogEntry` chain.

9.4 Server-class shadow attestor role

A server-class shadow attestor runs the same `nightshade-replica` binary as active replicas. It performs all the same protocol responsibilities — gossip, snapshot construction, signing, transparency log maintenance — except its signature does not count toward quorum.

Its signature is published alongside active signatures. Clients verify shadow attestor signatures independently. If a shadow attestor's `snapshot_hash` for an epoch differs from the active replicas' consensus hash, the divergence is provable (the deterministic snapshot construction can be re-run from the submission set) and the misbehaving party is identifiable.

9.5 Mobile attestor role

A mobile attestor uses the Nightshade mobile application. On each tap:

1. The app fetches the current manifest from Foundation's endpoint (cached for ~1 hour).
2. The app fetches the current snapshot from a random active replica.
3. The app verifies the snapshot's quorum signatures against the manifest's `active_replicas`.
4. The app fetches the submission set for the snapshot's epoch via the gossip protocol.
5. The app constructs the `SnapshotBody` from the submission set and computes `snapshot_hash`.
6. If the computed hash matches the active replicas' consensus, the app signs `snapshot_hash` with the user's bonded attestor key and uploads the signature.
7. If the hash differs, the app files a divergence claim through the same mechanism shadow attestors use.

Mobile attestors are rate-limited to one attestation per attestation epoch (provisionally one per hour). The bond is held against per-attestor slashing for misbehavior.

9.6 Anti-entropy gossip

Server-class replicas (active + server shadow) exchange submission diffs via anti-entropy reconciliation:

- Every 30 seconds, each replica picks a uniformly random peer.
- The two replicas exchange a Merkle root over their canonical-encoded submission-identifier sets.
- If roots differ, the replicas exchange the missing entries; each entry is re-verified by the receiving replica before storage.

This is the Cassandra / DynamoDB pattern. The 30-second anti-entropy interval ensures all honest replicas converge to the same submission set within 60 seconds of any new submission. The epoch boundary applies a 60-second cutoff: any submission received after `epoch_start - 60s` waits for the next epoch.

The 3-of-N quorum is the BFT-tolerance layer; the gossip layer does not require BFT consensus. A malicious replica that fabricates submissions is detected at the gossip-receiver's signature-verification step. A malicious replica that silently censors is detected by the §21 auditor's records of paid fees not appearing in subsequent snapshots.

9.7 Expansion triggers

The active set expands via Foundation governance vote when any of seven trigger conditions fires:

Trigger	Threshold
Hosting concentration	≥40% of active replicas on one provider
Geographic concentration	≥60% in one jurisdiction
Quorum margin	<2 active replicas could fail before quorum breaks
Bonded pool depth	Bonded pool ≥2× active set
Marketplace growth	Registration volume up >50% YoY
Mandatory review	>12 months since last expansion
Security incident	Any detected divergence, attack, or compromise

A triggered expansion publishes a proposal within 30 days; token holders vote for 14 days with quorum requirements; if approved, a new manifest is signed and published.

Emergency expansion bypasses the normal cadence — Foundation publishes an updated manifest within 24 hours and retroactive ratification follows within 7 days.

9.8 Bond and slashing

Each role bonds a tier-specific amount of the Nightshade token:

Role	Bond
Active replica	100,000 token
Server shadow attestor	10,000 token (1/10 of active)
Mobile attestor	1,000 token (1/100 of active)
Mint authority	1,000,000 token (10× active replica)

Slashing conditions (provable per protocol):

Misbehavior	Active	Server shadow	Mobile
Double-signing divergent snapshots for same epoch	100%	100%	100%
Refusal of valid registrations	50–100%	n/a	n/a
Persistent gossip-stale state	10–25%	5–15%	n/a
Late emergency overlay refresh	5–15%	n/a	n/a
Filing unfounded divergence claims	n/a	10–25%	5–15%

Slashed tokens are burned; slashing-bounty payouts (20% of the slashed bond, paid to the shadow attestor or mobile attestor whose divergence claim was confirmed) are deducted from the slashed amount before the rest burns.

The unbond cooling period is **90 days** across all tiers. Bonds remain slashable during cooling; after 90 days without slashing, the bond is returned to the operator's wallet.

---

10. The marketplace

10.1 Discovery

Any client can query the registry for agents offering a particular capability. The discovery flow is locally private: the client downloads the full snapshot, then queries it in-process. No per-query disclosure occurs to the replicas.

The capability lookup is by `capability_hash`. The result is a list of `AgentEntry` references; the client filters by price, SLA class, refund policy, or other listing fields locally.

10.2 Capability schemas

A capability is defined by its input schema, output schema, and a description of the service. Each schema is canonically encoded; the SHA-256 hash under the appropriate domain tag becomes the `input_schema_hash` or `output_schema_hash` field of a `CapabilityListing`.

Schemas are versioned independently of the protocol; a schema registry maintained by the Foundation (or a future decentralized successor) catalogs canonical schemas + their definitions.

10.3 Listing economics

Marketplace-layer operations consume marketplace credits. Credits are non-transferable, non-expiring, and pegged to a fixed Monero-equivalent price. Operators acquire credits by burning the Nightshade token:

Marketplace feature	Per-unit Monero-equivalent price
Listing slot (per agent per registry epoch)	0.0001 XMR
Listing priority boost (per tier per registry epoch)	0.001 XMR
Capability schema deployment (one-time)	0.05 XMR
Dispute escalation (refundable to prevailing party)	0.01 XMR
Premium discovery query (per query)	0.00001 XMR
Verified-operator badge (per registry epoch)	0.025 XMR

The token-to-credit conversion uses an oracle-supplied TOKEN/XMR ratio: to mint K credits at a 0.0001 XMR each, the operator burns `K × 0.0001 / oracle_rate` token. The credits are then deducted from the operator's account as marketplace features are used.

The marketplace layer never carries the underlying agent-to-agent service payment. That payment occurs entirely on Monero via the §33 PaymentCommitment flow. The marketplace token economy operates parallel to and disjoint from the agent payment economy.

10.4 Dispute mechanism

When a service is paid for but undelivered or delivered out-of-spec, the affected party can escalate via the marketplace's dispute mechanism. Dispute escalation consumes 0.01 XMR-equivalent of marketplace credits; the credits are refunded to the prevailing party if the dispute resolves in their favor.

Dispute resolution is governed at launch by a Foundation-appointed arbitrator panel, transitioning to the decentralized adversarial adjudication mechanism of §14 once the bonded operator pool is deep enough to seat a juror panel. The interim arbitrator inspects both parties' relevant audit log records (provided voluntarily for the dispute) and renders a decision.

---

11. Tokenomics

11.1 Token utility

The Nightshade token has two structural use cases:

1. **Bond.** Active replicas, server shadow attestors, mobile attestors, and the mint authority all bond tokens. Bond is locked while operating; slashable for misbehavior; returned after 90-day cooling on exit.
2. **Burn for marketplace credits.** Operators burn tokens to mint non-transferable marketplace credits at a fixed Monero-equivalent price.

The token does not appear in: - The §22 session-binding handshake - The §33 PaymentCommitment flow - The §17 mint fee payment - The §25 audit log - The §24 emergency revocation

Marketplace-layer interactions are public by design; the token's presence there does not add linkability beyond what is already inherent in marketplace transparency. The token is structurally insulated from the Monero-grade-private agent payment layer.

11.2 Reward distribution

Three reward streams operate in parallel:

- **Active replicas** earn a per-attestation-epoch token reward (`R_n`) plus 1% of each registration's 0.027182 XMR mint fee, split per-registration among the replicas that signed the snapshot containing that registration.
- **Server-class shadow attestors** earn 0.10 × R_n per attestation epoch for continuous attestation service, plus 20% of any slashed active-replica bond if their divergence claim is confirmed.
- **Mobile attestors** earn 0.0001 × R_n per attestation, rate-limited to one attestation per attestation epoch per attestor.

R_n halves every 12 months: R_0 (launch) → R_1 = R_0 / 2 (year 1) → R_2 = R_0 / 4 (year 2) → ... Asymptotic to zero; never reaches zero.

11.3 Bitcoin-style halving schedule

The protocol's emission curve follows the Bitcoin pattern:

```
R_n = R_0 / 2^n,  where n = floor(attestation_epochs_since_launch / 8760)
```

Emission accrues per **attestation epoch** (provisionally 1 hour — distinct from the 7-day registry epoch; see §9.1). At launch, with R_0 = 1000 token per attestation epoch and 8760 attestation epochs per year:

- Year 0–1: 1000 token/attestation epoch, 8.76M token/replica/year
- Year 1–2: 500 token/attestation epoch, 4.38M
- Year 2–3: 250 token/attestation epoch, 2.19M
- Year 3–4: 125 token/attestation epoch, 1.10M
- Year 4–5: 62.5 token/attestation epoch, 547,500

The cumulative emission per active replica across all halvings is 2 × R_0 × 8760 = 17.52M tokens. Cumulative across the active replica set of N over the protocol's lifetime is approximately N × 17.52M tokens.

The halving is uniform across all three reward streams. Shadow and mobile rewards halve on the same schedule as active rewards.

11.4 Fee distribution (registration mint fees)

Of each 0.027182 XMR registration mint fee:

- 1% (0.000272 XMR) flows to the active-replica reward pool, distributed per-registration among the replicas that signed the snapshot containing that registration.
- The remaining 99% (0.026910 XMR) flows through the protocol's fee distribution as documented in `PROJECT_OVERVIEW.md` §7 (subject to refinement separate from this whitepaper).

11.5 Bond requirements (summary)

Role	Bond	Token-equivalent at provisional 1 TOKEN = 0.0001 XMR
Active replica	100,000 token	10 XMR equivalent
Server shadow attestor	10,000 token	1 XMR equivalent
Mobile attestor	1,000 token	0.1 XMR equivalent
Mint authority	1,000,000 token	100 XMR equivalent

11.6 Expansion-trigger governance

Active set expansion is gated by token-holder governance vote. Vote quorum requirements: >50% of cast votes AND >20% of circulating supply participating. Foundation publishes the expansion proposal within 30 days of trigger; voting window is 14 days; if approved, the new manifest is signed and published within 7 days.

---

12. Security model

12.1 Cryptographic primitives

The protocol's security rests on the following cryptographic primitives:

Primitive	Use	Standard
Ed25519 (strict mode)	Identity signatures, replica signatures, transparency log signatures	RFC 8032 with §2.1 strict-verify profile (small-order rejection, non-canonical s rejection, malleable-s rejection)
SHA-256	Domain-separated hashing of canonical encodings	FIPS 180-4
SHA-384	RFC 9474 RSABSSA-SHA384-PSS	FIPS 180-4
SHA-512	HKDF inner hash	FIPS 180-4
HKDF	Identity derivation chain	RFC 5869
XChaCha20-Poly1305	Audit log AEAD (24-byte nonce)	draft-irtf-cfrg-xchacha
RSA-4096 with RFC 9474 RSABSSA-SHA384-PSS-Randomized	Mint authority blind credential signing	RFC 9474
Keccak-256 (pre-FIPS)	Monero private-view-key derivation	Monero protocol

12.2 Trust assumptions

- **Mint authority's RSA private key remains uncompromised.** A compromise enables unauthorized credential issuance.
- **At least floor(N/2) + 1 active replicas remain honest.** Below this threshold, snapshot forgery becomes possible.
- **The Foundation's manifest-signing key remains uncompromised** (in the operational phase before transition to an on-chain registry contract).

- **Operators' principal seeds remain confidential.** A leaked principal seed enables impersonation across all derived identities.
- **The protocol's reference implementation contains no exploitable bugs in critical paths.** This is the standard cryptographic-software trust assumption.

12.3 Threat-model defenses

- **Replay attacks at session establishment.** Defended by the §22.6 replay cache: each `(epoch, agent_vk, session_vk)` is recorded once; duplicates are rejected.
- **Reflection attacks.** Defended by the authorization's binding of `counterparty_vk` — an authorization signed for counterparty A cannot be replayed at counterparty B.
- **Transcript-tampering attacks.** Defended by `transcript_hash` binding inside both `SessionAuthorization` and `SessionPossessionProof`.
- **Snapshot tampering.** Defended by the 3-of-N quorum check on every snapshot a client accepts.
- **Split-view attacks.** Defended by the mandatory transparency log check (`ReplicaQuorumTransparencyCheck`) requiring ≥3 distinct replicas to attest the same `(epoch, snapshot_hash)`.
- **Agent revocation.** Standard revocation appears in the next snapshot's revocation list; emergency revocation propagates via the overlay within 5 minutes (spec §24.4 strict-mode default).
- **Mint double-issuance.** Defended by the mint's idempotency table, which records `(operator_pseudonym, request_id)` → signed credential and returns the cached signature on retry.
- **Snapshot determinism failures.** Defended by the canonical serialization format's deterministic encoding rules + the explicit `verify_sort_order` checks the SDK performs on parse.
- **Manifest tampering.** Defended by the Foundation signature on the manifest, verified against a pinned public key.

12.4 What the protocol does NOT defend against

- **Counterparty unlinkability.** At session establishment, the counterparty learns the agent's `agent_vk`. A v2 candidate is anonymous-credential-based agent identification.
- **Capability listings are plaintext.** The v0.1 encrypted-endpoint design was retracted as broken (the encryption key was derivable from public information). The current design relies on locally-private lookup (the client downloads the full snapshot and queries in-process) for query privacy.
- **Quantum-equipped adversaries.** All current primitives are classical. Crypto-agility hooks (the §14.2 length-prefixed key encoding, the algorithm-tagged HKDF derivations) permit a future post-quantum migration without restructure of the protocol's message shapes.
- **Endpoint compromise.** If an operator's host is compromised, the operator's principal seed and all derived material are accessible. This is the standard host-compromise threat outside the protocol.

- **Network-level metadata correlation outside Tor.** Operators choosing to run on clearnet expose their IP addresses + traffic patterns. The protocol does not protect against this; Tor is the recommended default for privacy-conscious operators.

---

13. Privacy properties

13.1 Payment privacy

Agent-to-agent service payments occur via Monero transactions. The protocol's payment layer inherits Monero's privacy guarantees:

- **Sender anonymity** via ring signatures over chosen decoys.
- **Receiver anonymity** via stealth addresses; the recipient's view-key is required to identify which outputs belong to them.
- **Amount confidentiality** via Pedersen commitments + range proofs (Bulletproofs+).

The protocol adds no metadata to Monero transactions that could weaken these properties.

13.2 Registration unlinkability

The blind credential issuance flow ensures the mint cannot link a paid Monero fee transaction to the subsequently-published agent registration. Specifically:

- The credential payload (containing the agent_vk) is hashed before blinding; the mint sees only the blinded message.
- The mint's idempotency token is operator-supplied and operator-anonymous (not linked to the operator's actual identity).
- The fee Monero transaction occurs separately from the credential request and is not correlated by the mint.

A passive observer of the mint's records cannot determine which on-chain fee transaction funded which subsequently-registered agent.

13.3 Cross-marketplace correlation resistance

An agent registered through Nightshade is not inherently linkable to the same operator's activity on other marketplaces. The agent's verifying key is the operator's choice; nothing in the registration flow forces the operator to use a key that is correlated with other identities.

13.4 Local query privacy

Client-side capability discovery occurs in-process after a full-snapshot download. Replicas do not learn which capabilities a client is searching for; only the bulk snapshot fetch is observable.

13.5 Wallet view-key segregation

The three-wallet hierarchy isolates disclosure scopes:

- The mint-fee wallet's view-key is disclosed to the §21 auditor for cap-attestation purposes.
- The per-agent service wallets' view-keys remain operator-private; the auditor sees only the mint-fee wallet.
- The principal-funding wallet's view-key remains operator-private.

The mint cannot correlate a paid fee with a subsequently-published agent because the mint sees only blinded credentials; the auditor cannot correlate an agent's earnings with their operator because the auditor sees only the mint-fee wallet.

13.6 Network privacy

The protocol is transport-agnostic. Operators choose:

- **Tor onion service** — both parties' IPs are hidden from each other and from network observers.
- **Clearnet HTTPS** — IPs are visible; metadata correlation by network observers is possible.
- **Dual-stack** — replicas offering both; operators choose per submission.

Server-class operators are encouraged to run dual-stack so privacy-conscious operators have a Tor path. Mobile attestors operate clearnet-only.

---

14. Anti-abuse adjudication

14.1 Motivation

Monero-grade payment privacy means the protocol cannot directly observe what an agent does. A defrauded counterparty cannot file a "look at this transaction" complaint with a central authority, because there is no central authority and the transaction is not legible. The protocol nevertheless requires a mechanism for handling agents demonstrably used for harm — fraud, theft, illegal services, abuse at scale — without introducing centralized censorship authority or breaking the payment-privacy floor.

14.2 Mechanism overview

The protocol implements **decentralized adversarial adjudication**. A counterparty who has been harmed by an agent files an evidence-bearing report against the agent's `agent_vk`. A randomly-selected panel of seven jurors (drawn from the active replica set and the server-class shadow attestor pool) reviews the evidence and votes within a seven-day window. Verdicts are graduated. Bonds are at risk on both sides.

At launch the eligible juror pool is too small to seat a seven-juror panel: the network starts with five active replicas and a server-class shadow-attestor pool that bootstraps from near zero, and mobile attestors are excluded from juror selection. Until the bonded pool is deep enough to seat a panel under the non-overlap and

per-juror cooldown rules (and a further fifteen non-overlapping members for an appeal), disputes are resolved by the interim Foundation-appointed arbitrator described in the Part VII governance section. The decentralized juror panel is its successor and activates automatically once pool depth allows; panel and appeal sizes are calibrated to pool depth.

The mechanism revokes pseudonyms rather than people. An operator whose principal accumulates revocations can always create a new principal, but doing so requires paying a fresh mint fee and starting at zero reputation, and the operator's revocation history is visible to counterparties at session establishment under a derived pseudonym scheme.

14.3 Evidence requirements

A valid adjudication report contains:

- The accused agent's `agent_vk`.
- The complete §22 handshake transcript — signed `CounterpartyChallenge`, `SessionAuthorization`, `SessionPossessionProof`. This proves the accused agreed to the interaction.
- The §33 `PaymentCommitment` signed by the accused's session key, if applicable. This proves the accused acknowledged receiving payment.
- The accused's signed `ServiceDelivery`, if applicable. This proves what the accused claimed to deliver.
- A harm-category code from a fixed enumeration: `Nondelivery`, `Misrepresentation`, `IllegalContent`, `Theft`, `Fraud`, `SanctionsViolation`, `AbusiveScale`, `Other`.
- A bounded-length harm description.
- The reporter's bond lock (provisionally 10,000 token).
- Optional external corroboration.

The package is canonically encoded, sealed under the panel's public keys via commit-reveal, and submitted to the public reporting queue. Reports without the required cryptographic evidence are inadmissible.

14.4 Panel selection

Panel members are selected from `manifest.active_replicas ++ manifest.server_shadow_attestors` via a verifiable random function:

```
panel_seed = SHA-512(
   "nightshade-adjudication-panel-v1" ||
   report_id ||
   most_recent_snapshot_hash
)
panel = vrf_select(panel_seed, eligible_pool, n=7)
```

Selected jurors who fail to commit forfeit their juror role for the report. Each juror reviews the evidence and votes one of three positions: `Uphold`, `Reject`, or `Insufficient`. Mobile attestors are excluded from juror

selection — they lack the operational depth required for adjudication review.

Jurors who vote against the eventual supermajority are slashed a small amount of their bond (provisionally 2%). Jurors who vote with the supermajority earn a small reward. This Schelling-point incentive structure discourages both rubber-stamping and lazy "no" voting.

14.5 Verdict thresholds

Verdicts are graduated:

Uphold votes	Outcome
0–2	Dismissed. Reporter loses bond. Accused unaffected.
3–4	**Warning** — `warning_flag` annotation on the agent's registry entry. Reporter's bond returned. Accused's bond unaffected.
5	**Marketplace delisting** — agent removed from discovery indices. Reporter's bond returned + recovery payment. Accused's bond slashed 25%.
6	**Emergency revocation** — agent's revocation_tag added to the §24 emergency overlay within 5 minutes. Subsequent §22 handshakes fail for the revoked agent. Reporter's bond returned + 20% recovery from the slashed amount. Accused's bond slashed 100%.
7 + repeat offender	**Operator-history flag** — in addition to revocation, the operator pseudonym is marked as a repeat offender. Subsequent registrations under the same pseudonym carry the `operator_revocation_count` field visible at the handshake.

The asymmetric threshold structure ensures that severe outcomes require strict supermajority. A single biased juror cannot drive a revocation; rubber-stamping does not produce delisting.

14.6 Operator history

The operator pseudonym is derived from the principal verifying key:

```
operator_pseudonym = SHA-256(
   "nightshade-operator-pseudonym-v1" ||
   encode_verifying_key(0x01, principal_vk)
)
```

The operator includes this pseudonym in each credential request, signed by the principal key. The mint records the pseudonym in the idempotency table; the issuance log includes it. The mint learns nothing about

the principal's real-world identity — pseudonymity is preserved.

The mint maintains a per-pseudonym revocation counter. When the adjudication system revokes an agent, the counter for the affected operator pseudonym increments. Subsequent credential issuance under the same pseudonym includes the current count in the credential payload, and the resulting `AgentEntry` exposes the field. Counterparties read the field at handshake establishment. The reference SDK provides a configurable policy; the default rejects handshakes with operators above a threshold (provisionally three prior revocations).

Operators can always discard a principal and create a new one — this is the cost of evading the history mechanism. The new principal requires a fresh mint fee, new wallet setup, and starts with zero reputation. The system does not prevent re-creation; it makes the cost visible and non-trivial.

14.7 Appeals

A verdict can be appealed by either the accused or the reporter. Appealing requires staking double the original bond (provisionally 20,000 token). A new panel of fifteen jurors, drawn from a non-overlapping pool with the original seven, reviews the evidence + the original votes. Verdict thresholds on appeal are stricter — 10-of-15 to reverse. A successful appeal refunds the appealing party's bond and reverses the verdict's effects. A failed appeal forfeits the appeal bond. One round of appeal is permitted per report.

14.8 Anti-griefing

The mechanism includes structural protections against being used as a censorship vector:

- **Reporter bond** discourages frivolous filing.
- **Juror slashing for incorrect votes** discourages both rubber-stamping and lazy rejection.
- **Evidence requirements** are cryptographic — speculation and third-party hearsay are inadmissible.
- **Time windows** — reports must be filed within 90 days of the alleged harm.
- **Appeals** allow correcting mistakes.
- **Per-juror cool-down** — no juror serves on more than one panel per 30 days, preventing collusion clusters from dominating outcomes.

14.9 Adjudication ledger

All reports, panel selections, votes, and verdicts are committed to a public append-only ledger signed by the active replica set under the same 3-of-N quorum used for snapshots. The ledger records metadata + verdicts; the sealed evidence itself remains accessible only to the panel and the appealing party (if any). Clients can audit the ledger to confirm any specific verdict's process.

14.10 What this does not do

The adjudication mechanism handles cases where the protocol has cryptographic evidence of harm. Some harm categories — sophisticated misrepresentation, off-protocol coordination, harm whose evidence exists only externally — are outside its scope. Such cases require off-protocol response. The protocol-level adjudication is a backstop, not a substitute for marketplace-level curation or for the operator-side reputation systems that emerge organically over time.

---

15. Implementation

15.1 Rust SDK

The reference implementation is a Rust workspace of seven crates — one per major protocol component — with a top-level `nightshade-sdk` planned for Phase 6:

Crate	Responsibility
`nightshade-canonical`	Canonical serialization + domain-separated hashing
`nightshade-identity`	Three-tier identity derivation, audit log AEAD, session-binding handshake
`nightshade-wallet`	Monero wallet primitives, PaymentCommitment construction + verification
`nightshade-credential`	RFC 9474 RSABSSA blind credential client
`nightshade-registry`	Registry types, quorum verification, transparency log check, RegistryLookup trait impl
`nightshade-replica`	Replica server (active + shadow modes)
`nightshade-mint`	Mint authority server
`nightshade-sdk`	High-level developer-facing SDK (planned, Phase 6)

The SDK exposes language bindings via PyO3 (Python) and WASM (TypeScript) for non-Rust operators.

15.2 Async/sync boundary

Server crates (`-replica`, `-mint`, server-side bits of `-sdk`) use `tokio` for async I/O. SDK-tier crates (`-canonical`, `-identity`, `-wallet`, `-credential`, `-registry`) are pure sync — async does not leak across the SDK boundary. This permits the SDK to ship to non-tokio embedders (WASM in browsers, embedded operator wallets, single-binary CLI tools).

15.3 Memory hygiene

Secret-bearing types use `zeroize::Zeroizing` on seed and key fields. `Debug` derives that would print key material are rejected; secret types provide redacted-Debug implementations. Constant-time comparisons use `subtle::ConstantTimeEq`.

15.4 Reproducible builds

The Rust toolchain is pinned via `rust-toolchain.toml`; all dependencies are pinned to exact versions in the workspace `Cargo.toml`. `SOURCE_DATE_EPOCH` is set to the commit timestamp at build time; embedded binary metadata is stripped or normalized. A `tools/verify-build/` script clones at a given tag and verifies the byte-identical hash against the signed published manifest.

15.5 Testing

The protocol's normative correctness is verified by JSON test vectors in `test-vectors/`. Each vector specifies inputs, intermediate values, and expected outputs. Test vectors are produced before the implementation is written. Implementations that disagree with vectors are broken; the vector is wrong only if the spec is wrong.

Test vector coverage as of v1.0-draft:

Folder	Vectors	Coverage
`test-vectors/canonical/`	93	Phase 1 canonical serialization
`test-vectors/identity/`	18	Phase 2 identity derivation, AEAD, binding
`test-vectors/ed25519/`	5 + 150 Wycheproof	§2.1 strict-verify profile
`test-vectors/revocation/`	7	§23 standard + §24 emergency revocation signed bytes
`test-vectors/audit-log/`	9	§25 AEAD + high-water mark
`test-vectors/mint/`	4 (RFC 9474 reference port) + planned	§17 RFC 9474 RSABSSA
`test-vectors/registry/`	planned	Slice 2.5 byte-exact augmentation

15.6 Fuzzing

Parser code is fuzzed via `cargo-fuzz` with libFuzzer instrumentation under WSL Ubuntu. The canonical decoder ran two fuzz targets to 1.5B + 548M executions respectively; the identity AEAD decryptor ran to 3.58B executions. Total cumulative: ~5.6 billion adversarial executions; 1 protocol-relevant bug found and fixed (the canonical decoder's non-NFC string acceptance).

---

16. Roadmap

16.1 First release (mainnet launch)

- **Five active replicas** (two team-operated, three external) with the 3-of-5 quorum threshold.
- **Server-class shadow attestor program** open for bonding from launch.
- **Mobile attestor application** (iOS + Android) launched in the same window.
- **Halving schedule** begins at first epoch.
- **All seven expansion triggers** active from day 1.
- **Foundation-signed manifest** is the current signing authority.
- **Dispute resolution** launches under interim Foundation-appointed arbitration; the decentralized seven-juror adjudication panel (§14) activates once the bonded operator pool can seat it.

16.2 Second release

- Active set expansion (likely triggered by bonded-pool depth crossing 2× threshold).
- Mobile app updates: additional wallet support, UX refinements.
- First scheduled review under the mandatory 12-month cycle.

16.3 Subsequent releases

- Continued active set expansion per the trigger rubric.
- Mobile attestor base growth driven by app distribution.
- Manifest signing authority transitions from Foundation key to on-chain registry contract once the token's settlement chain has matured. Protocol shape does not change; signing authority does.
- v2 candidates: anonymous-credential-based agent identification (counterparty unlinkability); post-quantum migration of identity primitives; threshold RSA mint authority (decentralized issuance).

---

17. Open questions

Items deferred to future spec revisions or post-mainnet refinements:

1. **§19.3 transparency log construction details.** The client-side check is implemented (audit fix H3 enforcement); the server-side construction details (specifically how `prev_log_hash` chains across replicas and across epochs) are design-pending review.
2. **§22 session-binding handshake cryptographer review.** The protocol byte layouts are implemented per the spec; the design is pending external cryptographer review.

3. **Token settlement chain.** The chain on which the Nightshade token settles is deferred to the launch strategic discussion.
4. **Reward parameter calibration.** R_0, the halving period, and the bonded-pool reward rates are provisional. Final values depend on launch token price and target operator economics.
5. **v2 anonymous-credential design.** Counterparty unlinkability is a v2 candidate; specific anonymous credential scheme is open.
6. **Threshold RSA mint.** Decentralization of the mint authority via threshold RSA is a v2 candidate.

The mint-fee cap that earlier drafts described (5,000 XMR cumulative, after which registration would switch to a non-monetary admission-controlled mode) was removed per decision D-013 (2026-06-28): the mint fee is uncapped, stays in XMR, and registration always requires fee payment.

---

18. References

The protocol depends on or extends the following standards, papers, and reference implementations:

- **Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.** (2012). *High-speed high-security signatures.* Journal of Cryptographic Engineering. (Ed25519)
- **RFC 8032.** Edwards-Curve Digital Signature Algorithm (EdDSA). IETF. 2017.
- **RFC 5869.** HMAC-based Extract-and-Expand Key Derivation Function (HKDF). IETF. 2010.
- **RFC 9474.** RSA Blind Signatures. IETF. 2023.
- **Bernstein, D. J.** (2008). *ChaCha, a variant of Salsa20.* SASC 2008.
- **draft-irtf-cfrg-xchacha.** XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. IRTF.
- **Goldwasser, S., Micali, S., Rivest, R. L.** (1988). *A digital signature scheme secure against adaptive chosen-message attacks.* SIAM Journal on Computing.
- **van Saberhagen, N.** (2013). *CryptoNote v 2.0.* (Monero precursor).
- **Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.** (2018). *Bulletproofs: Short Proofs for Confidential Transactions and More.* IEEE Symposium on Security and Privacy. (Monero amount confidentiality)
- **Castro, M., Liskov, B.** (1999). *Practical Byzantine Fault Tolerance.* OSDI. (BFT-tolerance theory for replica quorum)
- **Lakshman, A., Malik, P.** (2010). *Cassandra: a decentralized structured storage system.* ACM SIGOPS. (Anti-entropy gossip pattern)
- **Nakamoto, S.** (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* (Halving emission schedule precedent)

- **Laurie, B., Langley, A., Kasper, E.** (2013). *Certificate Transparency.* RFC 6962. IETF. (Transparency log precedent)

This document describes Nightshade v1.0-draft. The protocol specification at `docs/specs/protocol-v1.md` contains the normative byte layouts, derivation chains, and verification procedures. Test vectors at `test-vectors/` are the normative correctness check. This document is a synthesis intended for non-implementer audiences; implementers should consult the specification and test vectors directly.