

NamID Protocol — specification

Status: v1.5.1 draft · 2026-10-20 (v1.5: 2026-10-19 · v1.4: 2026-10-19 · v1.3: 2026-10-12 · v1.2: 2026-10-11 · v1.1: 2026-10-10 · v1.0: 2026-07-19; what changed: §20)

Scope: this document specifies the NamID Protocol and nothing else: the name primitive, the allocation of roots, on-chain registrars, sealed sets, records, resolution, and the protocol-level mechanisms for identity, agents, payment and addressing. Applications, services and deployments built on the Protocol are out of scope.

Rule: every mechanic cites the configuration constant, module or puzzle that implements it, so that the specification and the code cannot silently diverge. Where policy has not been decided, §19 says so; nothing is invented here.

Audit status: no puzzle described in this document has been independently audited. Every mechanic runs on Chia testnet11; none is used on mainnet before an independent audit.

1. Abstract

The Protocol allocates, binds, resolves and records human-readable names on the Chia blockchain. It has **no protocol token**: bids, prices, renewals, fees and settlements are paid in XCH.

A name is a Chia **singleton coin**, its *deed*. The deed's puzzle carries the name string, the owner's key, the lock policy and `DATA_ROOT`, the root hash of the name's records. The name *is* the on-chain object: anyone with a Chia node can walk the deed's lineage from its launcher to the unspent tip and learn, with consensus-grade certainty, who controls the name and what it commits to.

Top-level names (**roots**, also called TLDs) are released on a public, hash-determined schedule and allocated by sealed-bid second-price auctions. Under each root an **on-chain registrar** — a sorted linked list of coins — records which second-level names exist, which deed or key controls each, and the terms its root owner set. Uniqueness is enforced by consensus, not by a server. Records live off chain and are proved against `DATA_ROOT`. A conforming client resolves a name from a Chia full node alone and refuses any answer it cannot verify.

On this primitive the Protocol specifies a DID method (`did:namid`), agent cards and spending budgets, payment to a name, payment channels between names, and the publication of keys used to address mail and calls to a name — each strictly as an on-chain or cryptographic mechanism.

Three properties define the design:

- **Readable.** Labels are plain lowercase letters and digits (§3.1); a second-level name reads like the web (`katarina.x`).
- **Decentralized.** Roots are allocated by sealed-bid auction paid in XCH (§5) on a schedule anyone can compute (§4); second-level names are written by registrar puzzles that every Chia node enforces (§8). No emission, no token, no exchange dependency.
- **Verifiable.** Every answer a resolver gives is backed by an on-chain proof or refused (§11). Key loss is survivable through delayed, cancellable rotation (§3.5); a root is lost only by letting its lease lapse (§6).

2. Conventions and terminology

2.1 Requirement words

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT** and **MAY** are to be interpreted as described in RFC 2119 and RFC 8174 when, and only when, they appear in bold capitals. §18 collects the conformance requirements.

2.2 Terms

Term	Meaning
root (TLD)	a top-level name such as <code>x</code> ; one deed coin; allocated under §4–§7
name (SLD)	a second-level name such as <code>katarina.x</code> , issued under a root through that root's registrar (§8)
label	one dot-free component of a name (§3.1)
deed	the singleton coin that is a name's identity (§3.2)
launcher id	the coin id of a singleton's launcher; the deed's permanent identifier and its DID id
root owner	the holder of the key that controls a root's deed
registrar	the on-chain sorted list of a root's names, with the terms that govern them (§8)
generation	one registrar instance; its <code>TERMS</code> are fixed for its life (§8.7)
TERMS	the constants curried into every node of one registrar generation (§8.4)
DATA_ROOT	the 32-byte commitment to a name's records, carried in its deed (§3.4)
sealed set	an immutable on-chain commitment to a set of names that predate a root's registrar (§9)
client	software that resolves, verifies, builds or signs Protocol data on a user's behalf
root profile	a root owner's published rules for names under the root (§16.4)

2.3 Networks and clocks

The Protocol runs on Chia mainnet and on testnet11. Values marked *testnet11* are rehearsal values. Durations enforced by puzzles are counted in **blocks**, not seconds, wherever a puzzle can express them: a height is a fact, a timestamp is farmer-influenced. Chia targets 4608 blocks per day (`BLOCKS_PER_DAY` , `name_terms.py` and `key_roles.py`). Auction windows and leases are measured in days by the auction engine (§5, §6). Rehearsal modes compress clocks (`network_profile.py`) but never change a bucketing function, a hash or a puzzle.

2.4 Citations

Configuration constants are in `apps/namid/config.py` unless another module is named. Puzzle paths are relative to `primitives/all-kinds-of-nft/puzzles/` . Registry modules are in `apps/NalandaX/registry-Backend-artefacts/` . The chain-only reference client is `packages/namid-open/` (`namid_open.py` , `namid_sealed.py` , `namid-open.mjs`).

3. The name primitive

3.1 Label grammar

The label grammar is defined once, in `standards/namid-label-grammar.json` (version 1):

- a label matches `^[a-z0-9]+$`, 1 to 63 characters;
- input is lowercased and surrounding whitespace is trimmed before checking; the result is the canonical form;
- the separator is `.`; a name has at most 4 labels;
- hyphens, punycode (`xn--`), non-ASCII characters and symbols are refused.

The restriction removes whole classes of failure: there is no second spelling of a label to normalise, and no confusable script. Every implementation **MUST** reproduce the vectors in that file exactly, including the stated reason for each refusal. Widening the grammar is a governance decision (§16), never a drift.

3.2 The deed

A name's identity is a singleton coin (singleton top layer v1.1) whose inner puzzle is Chia's standard DID layer (`did_innerpuz`), and whose DID inner puzzle is the NamID covenant `did-deed/namid_deed_did.clsp`:

```
singleton_top_layer_v1_1( (SINGLETON_MOD_HASH . (LAUNCHER_ID . LAUNCHER_PH)),  
  did_innerpuz( COVENANT, RECOVERY_DID_LIST_HASH, N, SINGLETON_STRUCT, METADATA ) )
```

Every deed is therefore a standard Chia DID whose DID id is its launcher id, readable by any Chia DID tool. The covenant carries nine fields into **every generation** of the coin:

Curried field	Meaning
<code>MOD_HASH</code>	tree hash of the uncurried covenant (self-reference)
<code>NAME</code>	the full name, e.g. <code>katarina.x</code> ; immutable
<code>OWNER_PUBKEY</code>	the 48-byte BLS key that signs every owner action; rotatable (§3.5)
<code>LOCK_FOREVER</code>	1 = soulbound: the deed never changes holder (§3.3)
<code>UNLOCK_HEIGHT</code>	no transfer before this height (0 = none)
<code>DATA_ROOT</code>	the commitment to the name's records (§3.4)
<code>PENDING_KEY</code>	<code>()</code> or the announced next key of a rotation in flight
<code>READY_HEIGHT</code>	the height from which a pending rotation may be finished (0 when none)
<code>ROTATION_DELAY</code>	blocks a rotation must wait; 0 = rotation off; immutable

The DID layer runs its inner puzzle with the solution only, so the covenant cannot read the DID layer's curried values directly. The solution carries them and the covenant proves them by asserting `ASSERT_MY_PUZZLEHASH` against the singleton it would form. The covenant also asserts `ASSERT_MY_AMOUNT`, so a deed can never be melted.

Actions. Every action except `finalize` is signed by `OWNER_PUBKEY` with `AGG_SIG_ME` over `sha256tree(NAME action args)`.

Action	Arguments	Effect
<code>update</code>	new <code>DATA_ROOT</code>	commits a new records root
<code>transfer</code>	new 48-byte owner key	refused when <code>LOCK_FOREVER = 1</code> ; gated by <code>ASSERT_HEIGHT_ABSOLUTE UNLOCK_HEIGHT</code> ; clears a pending rotation and turns DID recovery off
<code>lock</code>	(<code>new_lock_forever</code> , <code>new_unlock_height</code>)	only tightens: soulbound is terminal and a time lock only moves forward; turns DID recovery off
<code>rotate</code>	(<code>new_key</code> , <code>ready</code>)	\$3.5
<code>cancel</code>	—	discards a pending rotation
<code>finalize</code>	—	installs the pending key at <code>READY_HEIGHT</code> ; no signature
<code>recovery</code>	(<code>recovery_list_hash</code> , <code>n</code>)	<code>n = 0</code> is always allowed; <code>n > 0</code> is refused when soulbound and waits for <code>UNLOCK_HEIGHT</code>
<code>metadata</code>	the DID metadata list	its first pair MUST be (<code>"aka"</code> , <code>NAME</code>) , so a deed for one name can never read as another in a plain DID wallet
<code>announcement</code>	a 32-byte message	<code>CREATE_PUZZLE_ANNOUNCEMENT</code> from the deed (used when an NFT is assigned to the DID); 32 bytes only

A DID recovery performed by software that does not know the covenant leaves the deed **frozen**: its inner puzzle is no longer the covenant, so no registrar node accepts it (§8.5) until the holder repairs it back into the covenant.

3.3 Lock policies

Exactly one lock mode applies to a coin:

Mode	Curry	Use
<code>transferable</code>	<code>LOCK_FOREVER = 0</code> , <code>UNLOCK_HEIGHT = 0</code>	may change holder at any time
<code>timelock(H)</code>	<code>LOCK_FOREVER = 0</code> , <code>UNLOCK_HEIGHT = H</code>	may change holder from height H
<code>soulbound</code>	<code>LOCK_FOREVER = 1</code>	never changes holder; roots won at auction mint this way (§5.4)

Rotation is not transfer. Soulbound forbids a change of *holder*, not a change of *key*: a soulbound deed still rotates (§3.5). The chain cannot tell "I rotated to my own new key" from "I handed my key to a buyer"; both are

the same edit and no puzzle reads intent. What `LOCK_FOREVER` enforces is narrower and true: `NAME` is re-curred into every generation and never changes, `transfer` is refused outright, and every key change is public for the rotation delay and cancellable throughout. A stolen key therefore cannot *quietly* become the owner.

Locked roots. A root whose deed is locked soulbound issues only soulbound names: the issuance gate refuses `timelock` and `transferable` under it (`subtld_register.owner_default` and `root_vault_locked` ; the answer a client reads is the asset-status record, format `namid-asset-status/1` , `standards/NAMID-ASSET-STATUS.md` , computed by `asset_status.py` , the same logic the gate calls). Unknown values are reported as `null` , never guessed; a coin that disagrees with its root's rule is reported as the coin says, with `consistent: false` .

3.4 DATA_ROOT

`DATA_ROOT` is one 32-byte value that binds a name to its records (§10). It takes one of two forms:

- 1. **A DataLayer root.** The root hash of the name's DataLayer store (binding modes `root` and `store` , `standards/dig-integration.md` ; `DataRootBinding` in `standards/dig-transport.ts`). Leaves and internal nodes hash as Chia DataLayer does (§10.3).
- 2. **A records fingerprint.** The sha256 of the name's canonical JSON records document. A client verifies a document obtained from anywhere by hashing it (`namid_open.py verify-records`).

`DATA_ROOT` changes only through the deed's `update` action, signed by the owner key. Each change creates a new deed generation, so the deed's lineage is the complete, ordered history of its record commitments.

3.5 Key rotation and the clawback window

Every change of a name's controlling key is **announced, delayed and cancellable**. A single spend can never move control.

DID deeds (`namid_deed_did.clsp`):

Step	Who	Consensus condition	Effect
<code>rotate</code>	the current key and the new key sign	<code>ROTATION_DELAY > 0</code> ; the new key is 48 bytes and differs from the current one; the spend lands only while <code>now ≤ ready - ROTATION_DELAY</code>	parks the new key in <code>PENDING_KEY</code> with <code>READY_HEIGHT = ready</code> ; replaces any pending rotation
<code>cancel</code>	the current key	a rotation is pending; no time bound	clears <code>PENDING_KEY</code>
<code>finalize</code>	anyone	<code>ASSERT_HEIGHT_ABSOLUTE READY_HEIGHT</code>	the pending key becomes <code>OWNER_PUBKEY</code>

`ROTATION_DELAY` is curried at mint (`DID_ROTATION_DELAY` , `records_chain.py`): **9216 blocks (two days) on mainnet**, 96 blocks on testnet11. The new key signs the announcement so that a name can never be rotated to a key nobody holds; `finalize` needs no signature so that a rotation can never be stranded by a key that has gone quiet; `cancel` has no time bound so that an owner can act the moment they notice.

Root deeds of the earlier form (`namid_tld` , with the covenant `ownership-lock/namid_tld_clawback.clsp`) follow the same three-phase rule with a relative clock: `rotate` or `transfer` parks the incoming key in `PENDING_OWNER` and leaves `OWNER_PUBKEY` unchanged; `finalize` is

permissionless after `ASSERT_HEIGHT_RELATIVE CLAWBACK_BLOCKS` ; `cancel` by the current key has no time bound. `CLAWBACK_BLOCKS = 9216` , measured from the pending coin's own confirmation height; `tld_clawback.py` and `src/lib/clawback.ts` mirror the constant, each with a drift check that reads the `.clsp` . A second change while one is pending is refused, so it is never ambiguous which change the window covers.

Why. A stolen key must not be a silent, instant, complete takeover. Under this rule a thief holding the key must announce the change publicly and wait; the owner cancels with the key they still hold. A full node rejects a `finalize` that arrives early: the protection is consensus, not a registrar's restraint.

Limitation. If a key is genuinely compromised, both parties hold "the current key", and each can cancel the other's rotation: a stalemate in which nobody rotates while it lasts. The window guarantees that no takeover is silent or fast; settling a contested key needs a separate recovery key (§19).

3.6 Identifiers: did:namid

Every deed is addressable as a DID under the `did:namid` method (`standards/DID-NAMID-METHOD.md` , reference implementation `did_namid.py`):

```
did-namid    = "did:namid:" [ network ":" ] id
network      = 1*16( %x61-7A / DIGIT )      ; omitted on mainnet
id           = launcher-id / name
launcher-id  = "1" 1*bech32-char             ; bech32m, checksum over the hrp "namid:"
```

- The **launcher form**, `did:namid:1...` (mainnet) or `did:namid:testnet11:1...` , is the canonical DID of a minted name; it never changes for the life of the coin lineage. Its bech32m payload is the 32-byte launcher id, so `did:namid:1...` and the protocol URI `namid:1...` share one checksum.
- The **name form**, `did:namid:katarina.x` , resolves to the same document and is an `equivalentId` of the launcher form once the coin exists.
- Identifiers are lowercase; uppercase is invalid, not normalised. `did:namid:mainnet:...` is invalid. An id that matches a Chia address (`^t?xch1...`) is invalid by grammar.
- The DID document lists `did:chia:1...` and `namid:1...` for the same 32 bytes under `alsoKnownAs` .

3.7 Deed forms and puzzle spines

Form	Puzzle	Status
DID deed	did-deed/namid_deed_did.clsp	every new name, and every root won at auction since 2026-10-17
boot eve	did-deed/namid_deed_did_boot.clsp	the first coin of an auction-minted deed; proves its own launcher id and becomes the DID deed (§5.4)
root deed, earlier form	namid_tld and its _tiers / _recoverable variants	roots bid before 2026-10-17; still valid; followed by registrars from v4 on (§8.5)
passkey deed	did-deed/namid_deed_did_phone.clsp	a DID deed whose owner is a 33-byte secp256r1 passkey key, verified inside the puzzle; recovery waits a public window the passkey can cancel; accepted only by registrar v7, a draft (§8.8)

Verifiers recognise exactly two puzzle spines: **v1 classic** (sha256tree as a Chialisp function; live on testnet11) and **v2 opcode** (native sha256tree, CHIP-0049; staged, fork-gated, build flag off). **One spine per coin, forever**: migration is a new mint, never a mutation. Pinned mod hashes: standards/SPINE-VERSIONS.md.

4. Roots: the release schedule

New roots enter circulation over a deterministic window of `RELEASE_WINDOW_WEEKS = 260` weeks (five years) from `RELEASE_EPOCH_ISO` (2026-03-03T00:00:00Z ; environment `NAMID_RELEASE_EPOCH`). Every root's release week is a public hash: computable by anyone, gameable by no one.

The bucketing function has a single implementation (`schedule.py` , `release_week_v2`):

```
week(name) = int.from_bytes( sha256( SEED || 0x00 || utf8(name) )[0:8], big-endian ) mod 260
SEED       = RELEASE_SEED_V2    (21 ASCII bytes, hex
68616e647368616b65642d72656c656173652d7632)
```

`RELEASE_SEED_V2` is **frozen schedule identity**: changing one byte reshuffles every root's week, so it changes only by a deliberate, signed governance decision (§16). Clients **MUST** use the seed bytes exactly as given.

Migration (`GRANDFATHER_V1_LIVE = true`). Roots already auction-live under the earlier 156-week v1 schedule (FNV-1a, seed `tld-rollout-v1`) remain released; the v2 schedule governs every root not yet released.

From week to auction.

1. When a root's release week arrives it becomes **queued**.
2. Interested parties join its **interest queue**, first come first served.
3. Each day the next `DAILY_CAP_N` auctions open from the queue. The cap is derived from the release window, not set independently: `_WEEKLY_SLOTS = 140` per week gives 20 per day (`DAILY_CAP_N_STATUS = "DERIVED-FROM-5Y-WINDOW"` ; environment `NAMID_WEEKLY_SLOTS` , `NAMID_DAILY_CAP_N`).

4. A root not yet at its week can be opened now by EARLY-OPEN (\$5.5).
5. A root whose lease has lapsed re-enters this pipeline immediately (\$6.3): its week is long past, so it is queueable and biddable on the same terms as a root never held.

5. Roots: sealed-bid auctions

Roots are allocated by **sealed-bid second-price (Vickrey) auction**, paid in XCH. The Protocol has one allocation mechanism; a re-auction after a lapse (\$6.3) is the same auction.

5.1 Phases

Phase	Window	What happens
open	—	the auction starts from a queue slot (\$4) or by early-open (\$5.5)
bid	AUCTION_BID_DAYS (10 days on the mainnet profile)	sealed bids are placed
reveal	AUCTION_REVEAL_DAYS (5 days)	bidders reveal on chain
settle	after the reveal window closes	the winner pays the second price; the root is minted (\$5.4)

The 2:1 bid-to-reveal ratio is preserved in every rehearsal mode, so a compressed clock rehearses the same mechanism.

5.2 Sealed bids

A bid is a **real** amount plus an optional **blind**. The deposit (real + blind) is a sealed-bid coin, `tldbbase_bid_v5c.clsp`, whose puzzle commits to `sha256tree(TLD, real, salt)`; one puzzle serves every amount. The commitment is made **in the bidder's own wallet**: the salt is derived from the wallet's signature, the auction service receives only the commitment and the deposit total (intent `namid-auction-bid-v2`), and the real amount is sealed to a key that only the bidder's wallet can re-derive. Nobody, the auction service included, can read a real amount before its bidder reveals it.

- The real amount **MAY** be 0: a blind alone makes a valid, non-zero deposit that can never win below the reserve.
- The minimum applies to the deposit (`NAMID_MIN_BASE_XCH`, `price_floor.py`).
- Every bid is a separate bid. One wallet may place several on the same root (`<wallet>`, `<wallet>~2 ... ~99`), each with its own coin, reveal and refund, all signed by the key of the wallet's first bid.
- A bid deposit carries **no memo** (\$13.3): nothing on chain says that a coin is a bid, for which root, or for how much, before the reveal.
- Deposits made before 2026-10-11 keep `tldbbase_bid_v5.clsp` or `tldbbase_bid_v5b.clsp`.

5.3 Reveal, refund and burn

A bidder reveals on chain during the reveal window by spending its own deposit through the reveal path. The real amount moves into a per-bid **escrow covenant** (`tldbbase_escrow_v4`) that can only pay the treasury or refund that exact bidder; the blind returns in the same spend; a revealed 0 creates no escrow coin.

An unrevealed bid is no bid. Nobody can know its real amount, so it neither wins, nor sets the price, nor is refunded. From close, anyone may spend the deposit to a burn address, and the coin itself requires the burn: at most 0.001 XCH of it may be left to the farmer as a fee. It never returns to the bidder and the auction service can never take it.

5.4 Settlement and the minted deed

The highest revealed real bid wins and pays the **second price**: the second-highest revealed bid, whoever placed it (a bidder's own second bid can set its price). A single revealed bid pays the reserve, `RESERVE_PRICE_XCH`, whose floor is **10 USD per root, paid in XCH** (`NAMID_BASE_PRICE_USD = 10`, a governance dial, converted at the XCH/USD rate by `price_floor.py`; a feed outage converts at a deliberately low fallback rate so that it can only over-charge). The difference refunds to the winner and revealed losers refund in full, in **one transaction with the mint**.

A win mints a DID-wrapped, soulbound deed, by protocol. At bid time every escrow carries `MINT_INNER_HASH`, the inner puzzle the winner's singleton must have. For every new bid it is the **boot eve** `namid_deed_did_boot.clsp`, which cannot know its launcher id at bid time and therefore proves its own launcher id when spent and becomes the deed. At settlement the escrow asserts that its launcher child is the singleton of exactly that boot eve, and the same bundle spends the boot eve, with no signature, into

```
singleton(launcher, did_innerpuz(namid_deed_did(NAME, OWNER_PUBKEY, LOCK_FOREVER = 1, ...),
no_recovery, ("aka" . NAME)))
```

— a standard Chia DID whose id is the launcher id, soulbound, with key rotation behind `ROTATION_DELAY`. Escrow, launcher and boot eve are spent in one block. There is no switch: `NEW_BID_DEED = "did"` (`auction_atomic.py`) is a protocol constant, and no environment variable or configuration selects another deed for a new bid. A bid placed before 2026-10-17 committed to the earlier `namid_tld` deed and settles exactly as it was funded (`deed_of()` reads a record without a `deed` field as `tld`).

A reader verifies the mint on chain (`auction_roots.verify`): the record coin is the boot eve for this root, owner and lock; the launcher's parent is this root's escrow, spent in the same block; and the escrow committed to that boot eve.

5.5 EARLY-OPEN

Any party may open a root's auction before its release week by paying, in XCH, in place of a regular open:

```
early_open_xch = EARLY_OPEN_MULTIPLIER × reserve price      (EARLY_OPEN_MULTIPLIER = 10)
```

- The payment goes to the treasury, like any sale (\$16.1).
- Early-open bypasses the interest queue (`EARLY_OPEN_BYPASSES_QUEUE = true`) and does not consume a daily slot (`EARLY_OPEN_CONSUMES_DAILY_SLOT = false`): the payment is the throttle.
- The opener gains **no bidding advantage**: the payment only starts the clock, and the opener bids like anyone else.

5.6 Unsigned drafts

Every chain-touching response from an auction or registry service is an **unsigned draft** with a human-readable summary. Only the holder's wallet signs, after approval; a service never holds a user's keys, and nothing in a draft path can transmit a transaction.

6. Roots: lease, renewal and lapse

6.1 The lease

A root is held by a **services lease** of `LEASE_YEARS = 3` (`LEASE_DAYS = 1095` on mainnet) that opens at settlement. While it runs, the binding of the root to its deed is absolute: no registrar, governance body or operator can take it.

The boundary is a covenant, not a promise. The lease is its own singleton, `namid_tld_lease.clsp`, and its two actions are gated by consensus time bounds:

Action	Time gate	Authorization
<code>renew</code>	<code>ASSERT_BEFORE_SECONDS_ABSOLUTE(TERM_END + GRACE)</code>	none : permissionless; proof of payment is an <code>ASSERT_PUZZLE_ANNOUNCEMENT</code> from the root's renewal escrow (§6.2)
<code>close</code>	<code>ASSERT_SECONDS_ABSOLUTE(TERM_END)</code>	<code>REGISTRAR_PK</code> , signed over <code>(TLD, DID_ID, "close", TERM_END)</code>

Consequences (verified against the mempool, `test_namid_tld_lease_mempool.py`):

1. **No one can take a live root.** A `close` before `TERM_END` is consensus-invalid, not merely refused by policy.
2. **No one can refuse a live renewal.** `renew` carries no registrar signature.
3. **GRACE is the redemption window, honestly expressed.** At 0 the two bounds coincide: there is no gap in which a root is neither renewable nor re-allocatable. Above 0 they overlap, and the singleton spends once, so whichever transaction lands first wins.

The identity constants (`TLD`, `DID_ID`, `ROOT_LAUNCHER`, price, escrow, registrar key) are re-curried into every generation, so a lease can never be re-pointed at another root, another identity or a lower price. A `close` melts the lease singleton. Any service's store of leases is a mirror of this truth; where the two disagree, the chain is right.

6.2 Permissionless renewal

Every owned root has one deterministic **renewal address**: the escrow covenant curried with the root, every spend path bound to the treasury. Anyone may renew any root: paying `N × RENEWAL_PRICE_XCH` extends the lease by N years when the coin confirms. Credits are idempotent per coin; a coin below the price credits nothing. `RENEWAL_PRICE_XCH` is a governance dial (placeholder 0.01 while `PRICING_TBD`, \$19).

6.3 Lapse and re-allocation

When `lease_expiry` passes, the root **enters the auction pool** at once (`LAPSE_GRACE_DAYS = 0`).

- **Redemption by the owner only.** Until its first re-auction opens, a lapsed root may still be redeemed, but only by its current owner (`LAPSE_REDEEMABLE_UNTIL_AUCTION_OPENS = true` , `LAPSE_REDEEM_OWNER_ONLY = true`): by the owner's own deed spend, or by the owner's key signing a claim for a renewal payment made after the lapse. A payment *confirmed* before the lapse counts like any renewal.
- **Renewal stops at the auction bell.** Once a re-auction opens, renewal is closed for everyone, the owner included; an unclaimed payment is recorded as late at the renewal address, and the auction decides the root.
- **Fresh mint, fresh identity.** The winner of a re-auction receives a new singleton with a new launcher id: generation N+1, with nothing carried over — not the records, not the DID. The lapsed coin, being soulbound, can never transfer; it persists on chain with no standing and is kept in the root's history.

The *lease* is the property right and the *coin* is the identity: neglect returns the namespace to the commons, and never hands an identity to a stranger.

7. Reserved roots and allocation classes

Not every root is publicly auctioned. Each label belongs to at most one class below; the classes are disjoint, and every overlap with the ICANN or sanctions lists is written down with its winner (`roots.PINNED_OVERLAPS` , enforced by `apps/namid/tests/test_one_reservation_per_label.py`).

Class	Members	Rule
(a) Governance-held roots	the single characters <code>a – z</code> and <code>0 – 9</code> not in class (d), and the utility roots <code>.tld</code> , <code>.did</code> , <code>.dao</code> , <code>.admin</code> , <code>.root</code> , <code>.null</code> , <code>.strict</code> , <code>.us</code> , <code>.india</code> , <code>.bharat</code> , <code>.usa</code>	held as governance treasury assets; allocation only by governance proposal (<code>SINGLE_LETTER_DAO_GATED = true</code>); no operator can allocate them unilaterally
(b) Registrable-domain gifts	the first 10,000 eligible labels of <code>standards/reserved-popular-sites.tsv</code> (a dated DNS-request ranking), less labels another class holds (declared in the snapshot header)	held for the operators of the corresponding registrable domains. A claimant publishes a one-time challenge in a fully DNSSEC-validated apex TXT RRset; the proof binds the root and the recipient puzzle hash and rejects CNAME and wildcard substitution. A valid proof only makes a governance allocation proposal eligible; it does not mint or bypass quorum
(c) ICANN root-zone strings	the live IANA root zone, snapshot <code>standards/reserved-icann.txt</code> , loaded fail-closed by <code>config.reserved_icann()</code>	reserved; never publicly auctioned, never gifted. The Protocol never squats the legacy DNS root
(d) Root-owner service and brand roots	roots a root owner operates for typed services and never auctions (<code>roots.SERVICE_ROOTS</code> ; for example <code>.x</code> , <code>.nm</code> , <code>.ids</code> , <code>.mojo</code> , <code>.chia</code>), and roots held for brand protection under which no names are issued	service roots issue names under their root profile (§16.4); a brand root refuses every issuance (<code>brand-root</code>)
(e) Sanctions exact-name screen	<code>standards/blocked-ofac.tsv</code> , generated from the official SDN and consolidated lists (primary names and strong aliases)	an exact normalisation into the label alphabet is refused before allocation. This is a conservative namespace screen: not fuzzy matching, not sanctions compliance, not KYC. It fails closed when the pinned file is unavailable
(f) Participant gift reservations	<code>roots.CHIA_ECOSYSTEM_GIFTS</code>	held by the treasury for possible good-faith gifts. A reservation proves no identity, creates no entitlement and promises no allocation; the treasury verifies the intended recipient before any transfer

Precedence when a label appears in several sources: grammar, then the sanctions screen, then service-root policy, then the governance hold, then the ICANN lock. Overlap counts are published with the reserve lists.

8. Registrars

Every root that issues names has an **on-chain registrar**: the list of its names, kept by a Chialisp puzzle that every Chia node enforces. A name is registered **only** when it is on chain in its own coin; a list, commitment or reservation held off chain reserves nothing, and an unminted label is free for anyone.

8.1 Structure

A registrar is one coin per name, kept in a **sorted linked list** that starts at the root's genesis head. Each node carries:

```
MOD_HASH REGISTRY_ID TERMS VALUE OWNER STORE EXPIRY NEXT_VALUE NEXT_ID [ROTATION] [RANGE]
genesis head:  VALUE = ""  OWNER = 0  STORE = 0  EXPIRY = 0  NEXT_VALUE = 0xff...ff (32
bytes)  NEXT_ID = 0
```

Field	Meaning
REGISTRY_ID	the root's deed launcher id; every node coin is hinted with it, so one hint query lists the whole registrar
VALUE	the label bytes
OWNER	a 48-byte BLS key, or a 32-byte deed launcher id (§8.5)
STORE	the DataLayer store id of the name's records (0 = none)
EXPIRY	height at which the name lapses (0 = never)
NEXT_VALUE , NEXT_ID	the successor's label and coin
ROTATION	() or (new_key . ready_height) , a key change in flight (from v2)
RANGE	() , or an interval (§8.6) (from v6)

8.2 Uniqueness and recognition

An insert is accepted only inside its predecessor's gap: `VALUE <s new_value <s NEXT_VALUE` in byte order (**strict-gap insert**). Because every insert must fall strictly inside one gap, a label has at most one node: **uniqueness is the list itself**. A node is *recognised* only if it descends from the genuine genesis head through spends of the registrar puzzle; a coin anyone creates with the same puzzle hash and invented curried values has another parent and is an orphan that clients ignore (§11.1). The list never needs a removal spend.

8.3 Modes

Solution: `(mode my_coin_id amount args)`. Modes of the current registrar, v6
(`register/namid_slot_node_v6.clsp`):

Mode	Arguments	Rule
1 insert	(new_value new_owner new_store new_expiry)	strict-gap check; creates the new node and re-creates the predecessor pointing at it; pays PRICE (proved by a settlement announcement); new_expiry at most one term from now (0 when TERM_BLOCKS = 0)
2 renew	(new_expiry)	anyone may pay; new_expiry ≤ max(EXPIRY, now) + TERM_BLOCKS and ≤ now + MAX_AHEAD ; refused when TERM_BLOCKS = 0
3 update	(new_owner new_store)	OWNER signs (AGG_SIG_ME); a new owner only when TRANSFERABLE = 1 ; BIND : a key-owned node may move to the 32-byte deed holding that same key, whatever TRANSFERABLE says
4 takeover	(new_owner new_store new_expiry)	only after ASSERT_HEIGHT_ABSOLUTE EXPIRY ; paid like an insert; the node keeps its place
5 announce key	(new_key ready)	current and new key sign; ready ≥ now + ROTATION_DELAY
6 cancel key	—	current key; the node returns exactly to what it was
7 finish key	—	anyone, after ready (ASSERT_HEIGHT_ABSOLUTE)
8 reserve	(lo hi commitment_root auth count new_expiry)	creates an interval head (§8.6)

Every node spend asserts ASSERT_MY_AMOUNT NODE_AMOUNT , so no spend can leave a zero-value node. Modes 5–7 are refused on a deed-bound node: its key changes on the deed (§3.5) and the node follows with no spend of its own.

8.4 TERMS

TERMS is one tree curried into every node of a generation. A v6 node carries fourteen terms:

#	Term	Meaning
1	TREASURY_PH	where PRICE is paid on insert, renew and takeover
2	PRICE	mojos per term (0 = free)
3	TERM_BLOCKS	blocks per term; 0 = names never expire (no renewal, no takeover)
4	TRANSFERABLE	1 = a name may change holder; 0 = never
5	SETTLEMENT_PH	the standard settlement_payments puzzle hash; payment is proved by its announcement, so collecting a fee needs no registrar key
6	ROTATION_DELAY	blocks a key change waits; 0 = key changes off
7	NODE_AMOUNT	every node's amount (1)
8-9	SINGLETON_MOD_HASH , LAUNCHER_PUZZLE_HASH	to rebuild a deed's singleton puzzle hash
10	MAX_AHEAD	how far ahead a name can be paid (at least one term; ten terms on the reference registrars, name_terms.MAX_AHEAD_TERMS)
11-12	DID_INNER_MOD_HASH , DEED_MOD_HASH	to rebuild a DID deed (§8.5)
13	ROOT_NAME	the root's name; a deed must be this name's deed
14	RANGE_KEY	the key that co-signs interval reservations; () = reservations off

8.5 Deed-bound owners: one key per name

A node's OWNER may be the name's deed: its 32-byte launcher id. An owner action on such a node carries the deed's current values; the node rebuilds

```

cov      = curry(DEED_MOD_HASH, DEED_MOD_HASH NAME OWNER_PUBKEY LOCK_FOREVER UNLOCK_HEIGHT
DATA_ROOT PENDING_KEY READY_HEIGHT ROTATION_DELAY)
did      = curry(DID_INNER_MOD_HASH, cov RECOVERY_DID_LIST_HASH N STRUCT METADATA)
coin id = sha256(deed_parent, singleton(launcher = OWNER, did), deed_amount)

```

and asserts that this coin is spent alongside (ASSERT_CONCURRENT_SPEND) and that the covenant's OWNER_PUBKEY signs (AGG_SIG_ME). The deed's key is therefore the only key that controls the name. The node also checks that the deed is *this* name's deed (NAME = VALUE " ." ROOT_NAME) and, under a root whose names never change hands, that the deed is soulbound. A frozen deed (§3.2) has no coin with that id and authorises nothing until it is repaired.

8.6 Intervals, owner-only and open registrars

A v6 node may stand for a label interval [lo, hi) : RANGE = (HI AUTH COUNT) with COUNT > 0 is an **interval head** (VALUE = lo , OWNER = AUTH , STORE = a commitment root). An insert whose predecessor carries a RANGE and whose label is below HI lands **inside** the interval: only AUTH may make it (AGG_SIG_ME AUTH over ("inside" label owner store expiry)), no PRICE is paid again, and the new node inherits

(HI AUTH 0) , so the gap after it stays guarded. An interval is created by mode 8 only from a predecessor whose gap holds all of [lo, hi) , never inside another interval; RANGE_KEY and auth both sign, and PRICE × count is paid. An interval head renews at PRICE × COUNT , never changes owner and never rotates; a lapsed head may be taken over and becomes a plain name.

Two registrar modes follow (mode in the client manifest, registrars.<network>.json):

- **Open registrar.** No interval covers the label space: anyone may insert a free label by paying PRICE .
- **Owner-only registrar.** One interval covers every label, with AUTH = the root owner's key: only the root owner can add names (open_registrar.py issue). A root won at auction issues names only through its owner until the owner opens it for registration in the rule book (§16.4).

8.7 Generations and conformity

TERMS are fixed for a registrar's life. A registrar that disagrees with its root's rules is **retired, not edited**: a new generation is issued with the right terms, its names are written into it, and the retired generation's nodes are kept and shown as history. namespace_rules.registrar_terms(root) states what a root's registrar must carry and conformity() names every difference; a writer **MUST NOT** insert into a non-conforming generation. Which generation of a root is current is not a fact the chain can state: the client manifest names it (§11.6).

8.8 Registrar versions

Version	Change
v1	sorted list, modes 1–4, five terms
v2	key change separate from transfer (modes 5–7, ROTATION_DELAY)
v3	fixed node amount (NODE_AMOUNT , ASSERT_MY_AMOUNT)
v4	deed-bound owners: one key per name
v5	renewal adds a term on top of the current expiry (MAX_AHEAD)
v6 (current)	the deed is a Chia DID (DID_INNER_MOD_HASH , DEED_MOD_HASH , ROOT_NAME); intervals and owner-only registrars (RANGE , RANGE_KEY , mode 8)
v7 (draft, not audited)	two switchable terms on top of v6: a passkey-deed term (PHONE_DEED_MOD_HASH ; () = off) under which a deed whose owner key is 33 bytes authorises a node change by announcing the node's message, ASSERT_PUZZLE_ANNOUNCEMENT sha256(deed puzzle hash, sha256tree(("node" msg))) ; and a sealed-set term (SEALED ; () = off) that enforces \$9 in the puzzle

The two v7 branches (register/namid_slot_node_v7.clsp and register/namid_slot_node_v7_sealed.clsp) are being merged into one TERMS layout before audit. Until an audited v7 exists, **v6 remains the current registrar**; v7 runs only in a testnet11 test generation.

8.9 Registration in one bundle

A name is registered by one spend bundle that launches its DID deed and inserts its node, paid and signed by the buyer's own wallet (namid_open.py register). The owner pays every fee of the transaction (§16.5).

9. Sealed sets

A **sealed set** makes names that exist on chain as deeds, but predate their root's registrar, provable from the chain alone, with one immutable anchor per root. The feature is specified here as it runs on testnet11; it **MAY** be removed in a later version of the Protocol (§19). Anchors already on chain are unspendable and stay valid whatever is decided.

9.1 Leaves

A leaf is the CLVM list `(label launcher_id owner_pk data_root)`; the leaf hash is its `sha256tree`. These are exactly the fields that, with the launcher id, determine the name's **eve deed coin** `(deed_eve_id(name, launcher, owner, data_root))`: the coin id of the soulbound DID deed minted with those values, amount 1). If that coin exists on chain, that launcher minted that name with that key: one coin lookup makes a leaf *true*, not merely committed. `data_root` is the `DATA_ROOT` at mint; later record updates live in the deed's lineage.

9.2 Tree and proofs

Leaves are sorted strictly increasing by label bytes (the registrar's own order) and padded with `()` to 2^{depth} . An internal node is `sha256(0x02 || left || right)`, so the Merkle root is the `sha256tree` of a CLVM tree of cons cells, and a Chialisp puzzle checks a path with `(sha256 2 l r)`.

- **Membership:** the leaf, its index and `depth` siblings.
- **Non-membership:** the two *adjacent* leaves (indices i and $i + 1$) with `left < label < right`; or the first leaf alone when `label < first`; or the last alone when `label > last`. An empty set proves every label absent. This is the registrar's strict-gap rule applied to the set.

9.3 Anchor

The anchor is a 1-mojo coin whose puzzle is `(x) (mod hash 044ebff3ad6d351856fd37d38c599ad0e416aa8a22621d8c13c27e24561f7f77)` curried with

```
("namid-sealed-set/1" root registry_id merkle_root count leaf_file_sha256)
```

`(x)` raises for every solution, so the coin can never be spent, moved, melted or changed, by anyone. Its puzzle hash commits to all six fields; a client recomputes it and reads the coin. The anchor is found by the hint `sha256("namid-sealed-set/1:" || registry_id)` and **authenticated by the client manifest**, exactly like a registrar's genesis head. The leaf file is content-addressed by the sha256 the anchor commits to: any copy that hashes correctly is the file, wherever it was fetched from (`namid_sealed.fetch_leaf_file`).

9.4 Enforcement

- **Registrar v6: resolver-enforced.** A label in a root's sealed set belongs to the sealed name. A free label is one absent from the registrar list (gap) **and** from the sealed set (non-membership proof). If a registrar node for a sealed label has the same deed, both agree; if it has another owner, resolvers follow the sealed entry, committed first, and report a `collision` (§11.3). The v6 puzzle cannot see the set, so it cannot refuse such an insert itself.
- **Registrar v7 (draft): puzzle-enforced** when its `SEALED` term is set to `(SEALED_ROOT SEALED_COUNT SEALED_DEPTH SEALED_ANCHOR_ID)`. Every insert carries a non-membership proof for its label (inside an

interval too); a reservation proves that no sealed label lies in `[lo, hi)` ; and a new mode 9, **claim**, moves a sealed name into the list: the leaf is proved a member, the new node's owner is its deed, the deed's current coin is spent alongside and its key signs `("claim" label new_store new_expiry)` . A claim pays no `PRICE` : the claim price is 0 under the current terms. A label in the set has no non-membership proof, so it can never be inserted by anyone else.

10. Records

10.1 Model

A name answers questions through typed key/value records in its store, committed by `DATA_ROOT` (§3.4): one hash on chain, the data off chain, every answer provable against the hash. The record keys (`standards/NAME-RECORDS-SCHEMA.md` , published by `datalayer_view.py`):

Key	Value	Meaning
<code>_namid/name</code>	<code>katarina.x</code>	the name, ASCII
<code>_namid/did</code>	<code>did:namid:...</code>	the name's DID (launcher form once minted)
<code>address</code>	a Chia address	where payments to the name go (§13)
<code>dns/<TYPE>/<n></code>	JSON <code>{type, name, data, ttl}</code>	one DNS resource record; <code>TYPE</code> in <code>dns_records.SUPPORTED</code> (A, AAAA, CNAME, TXT, MX, NS, DS, SRV, CAA, TLSA, SSHFP, SVCB/HTTPS ...)
<code>socials/<kind>, profile/<k></code>	text	handles and display profile (<code>metadata.SOCIAL_KEYS</code>)
<code>agent/card</code>	the canonical JSON of an agent card	§12.3
service record	endpoint, routing keys, X25519 key-agreement key	the keys used to address mail and calls to the name (§15)

In a root's shared store every key is prefixed `<label>/` . Richer documents (credentials, files) belong in their own store, referenced from a record, never inlined.

10.2 Typed and versioned

Every record write is signed by the key that controls the name (`record_guard`) and becomes effective only when the deed's `update` commits the new `DATA_ROOT` , so each version of a name's records is a generation of its deed. Every structured answer the Protocol defines carries a versioned schema tag (`namid-asset-status/1` , `namid-pay-to/1` , `namid-sealed-set/1` , `namid-channel/1` , `namid-invoice/1`), and a reader that meets an unknown version **MUST** say so rather than guess.

10.3 Merkle anchoring

Record proofs use Chia DataLayer's hashing: leaf = tree hash of (key . value) , internal node = sha256(0x02 || left || right) , so a proof verifies with sha256 alone. Chia DataLayer is the storage and mirroring layer; any peer that can serve (record, proof) is a valid source, because verification (§11) never trusts the server.

A registry that serves many names **MAY** additionally anchor the Merkle root of its index on chain: a 1-mojo coin whose puzzle hash is the root, created from a pinned anchor address (slot_anchor.py , at most every ANCHOR_EVERY_S = 600 seconds). Such an anchor is a public, timestamped statement by its operator, not proof of the operator's honesty; a name issued after the last anchor shows as *pending anchor*. Registrars (§8) and sealed sets (§9) are what remove that trust.

10.4 DNS, certificates and service discovery

- **DNS.** Records are resource records, so a resolver can synthesise DNS answers from them. service records {service, protocol, priority, weight, port, target, ttl} map to SRV; identity records map to _namid TXT. A bridge validates DNS labels, port ranges, TTL bounds and TLSA parameters only after the record's proof succeeds, and **MUST NOT** present an unsigned RRset as DNSSEC: an authoritative frontend must create its own RRSIG and DNSKEY data and publish a DS through its parent zone.
- **Certificates (DANE).** tlsa records use the standard tuple usage selector matching-type association-data ; DANE-EE 3 1 1 is the recommended profile for holder-controlled services. The trust order is: current deed → DATA_ROOT → Merkle proof → TLSA record → certificate or SPKI match. PKIX validation remains required for usages 0–2. A mismatch, or an unavailable current generation, is a hard failure, never a silent fallback to WebPKI.

10.5 Privacy

- **No personal data in any record, ever.** Private facts are stored as salted digests sha256(salt || claim) ; the holder selectively discloses (salt, claim) with an inclusion proof.
- A name's owner **MAY** be issued a signed credential file (W3C Verifiable Credential shape, <name>.namid-id.json): name, DID and launcher are verified; linked handles come from the name's records; personal details are self-attested and labelled so. The issuer keeps no copy and nothing about it goes on chain.

11. Resolution

Resolution is **fail-closed**: an unverifiable answer is never returned as success.

11.1 Client algorithm

resolve(name, type, key) :

1. **Parse** the name and canonicalise every label (§3.1); refuse on a grammar failure.
2. **Load the root's registrar** from the client manifest (registrars.<network>.json): its current generation, TERMS , genesis head coin and any sealed-set anchor (§11.6).
3. **Sync the list from the chain.** Fetch the coins hinted with REGISTRY_ID , group them by parent, decode each parent's spend (puzzle reveal and solution), recompute the children the registrar puzzle must create, and require them to equal the coins on chain, puzzle hash for puzzle hash. Only coins descended from the genesis head are nodes. Stop CONFIRMATIONS blocks below the peak.

4. **Find the label.** If a node holds it, its curried fields give the owner (key or deed), `STORE` and `EXPIRY` . If the label falls in a gap and the root has a sealed set, obtain a membership or non-membership proof and check it against the anchor coin read from the chain (§9).
5. **Read the deed.** For a deed-owned or sealed name, walk the deed's lineage from its launcher to the unspent tip, rebuild its covenant from the last revealed spend, check that the current coin matches the state that spend creates, and read `OWNER_PUBKEY` , `LOCK_FOREVER` and `DATA_ROOT` . A sealed name's leaf is confirmed by looking up its eve deed coin.
6. **Fetch and verify the record** with its proof from any source, against `DATA_ROOT` .
7. **Return** a resolution-proof object (§11.2) or a typed outcome (§11.3).

11.2 The resolution-proof object

A successful resolution returns `{name, network, height, registrar: {root, generation, node coin id}, deed: {launcher id, coin id, owner, lock}, data_root, record, proof, sealed?, verified: true}` : everything a second client needs to re-check the answer offline against the chain. A client **MUST** ignore any `verified` flag set by a server and recompute it.

11.3 Typed outcomes and failures

Outcome	Meaning
<code>registered</code>	a node, or a sealed leaf with a proof, holds the label
<code>unregistered</code>	the label is in a registrar gap and, where the root has a sealed set, proved absent from it
<code>registered: null</code>	unknown: the root has a sealed set and no proof source answered; never reported as free
<code>lapsed</code>	the node's <code>EXPIRY</code> has passed (only under <code>TERM_BLOCKS > 0</code>)
<code>collision</code>	a registrar node for a sealed label has another owner; the sealed entry wins (§9.4)
<code>frozen</code>	the deed's inner puzzle is not the covenant (§3.2)

Failure reasons (`FailureReason` , `standards/dig-transport.ts`): `data_root_mismatch` (a hint does not recompute to the curried `DATA_ROOT`), `store_not_found` , `stale_root` (a proof made against a superseded root), `proof_invalid` , `record_missing` (the key is absent under a verified root), and `transport_unreachable` (every source exhausted: an availability failure, never an integrity one).

11.4 Chain-only operation

A conforming resolver needs exactly one external dependency: a Chia full node, or any RPC compatible with the coin-set API (`get_coin_records_by_hint` , `get_puzzle_and_solution` , `get_block_spends` , `push_tx`). It **MUST NOT** depend on any operator's server to decide ownership, uniqueness or record validity. The reference client (`namid_open.py` , and `namid-open.mjs` for JavaScript, both without dependencies beyond sha256) refuses operator hosts outright (`BLOCKED_HOSTS`), and its end-to-end test blocks them at the socket.

11.5 Multiple peers and caches

A resolver **MAY** ask several independent peers for the deed generation, record and proof. Only identical verified answers at the highest observed height count toward a quorum; duplicate peer identities count once; conflicting

deed tips at one height fail closed; invalid proofs are recorded against the peer. Caches are bound to (deed coin id, DATA_ROOT, height) : a record cached for an older coin or root is unusable after a rotation or update even if its TTL has not elapsed, so a DNS-style TTL can never extend compromised key material past the on-chain change.

11.6 What the chain alone does not prove

- **Which generation of a root is current.** A retired generation's list still verifies; it is simply not the one new names go into. The manifest names the current generation, the genesis head coins and the sealed-set anchors; a client **SHOULD** obtain its manifest from more than one source and pin it.
- **Deep reorganisations** below the client's confirmation depth.

12. Identity and agents

12.1 Names are identities

A name is simultaneously a readable label and a persistent Chia identity:

1. the launcher coin fixes the identity lineage, and its id is the DID id (§3.6);
2. the unspent tip of the deed fixes the current owner key and DATA_ROOT ;
3. key rotation advances the deed without changing the launcher identity (§3.5);
4. a lapsed and re-auctioned root is a new generation with a new identity (§6.3); a name under a root with TERM_BLOCKS = 0 never lapses.

12.2 Authentication

An application authenticates possession of a name with a fresh, domain-separated challenge signed by the key that controls the name's deed (or, for a key-owned registrar node, by its OWNER key). A displayed name without a verified lineage and a key-possession proof is a label, not an authentication. A relying party **SHOULD** read LOCK_FOREVER before treating a name as a persistent personal identity: a transferable deed can change holder.

12.3 The agent card

Software that acts and pays needs a name others can find, check and pay. The Protocol gives it one without giving it the key. The **agent card** is a record on a name (standards/AGENT-CARD.md): what the agent does, where to reach it (url , MCP servers, OpenAPI, skill.md), how to pay it (pay_to , accepts) and how it spends (spending : a person approves each payment, within a limit, or not at all). Its shape follows the open A2A agent card, so tools that read agent cards read this one.

- The card is written only by the key that controls the name, in the same signed edit as every record.
- Its canonical JSON hashes to card_hash and is published as the record agent/card , committed by the name's DATA_ROOT .
- A client that fetched a card from anywhere checks it against the chain; a card that does not match is not the owner's.
- An agent **MAY** propose a name or a card. A proposal pays nothing, signs nothing and reserves nothing; it takes effect only when the person who will own the name signs the registration and the card in their own wallet. The agent never holds the name's key.

- An agent card is a record, never a separate NFT: an NFT can change hands apart from the name it describes, while a record cannot.

12.4 The spending budget

An owner can give the agent behind a name money of its own: a coin,

`agent/namid_agent_allowance_v3.clsp`, that the **agent's own key** spends from within limits the puzzle enforces.

```
CURRIED: MOD_HASH ALLOWANCE_ID OWNER_PK OWNER_PH AGENT_PK LIMIT PERIOD EXPIRY MAX_PAY
RETURN_FEE WINDOW_START SPENT
```

Mode	Who	Constraint
1 pay	the agent signs (payee_ph amount fee now)	payments and network fees together never exceed <code>LIMIT</code> mojos in any window of <code>PERIOD</code> blocks; no single payment exceeds <code>MAX_PAY</code> (0 = no per-payment cap); refused at or after <code>EXPIRY</code> (<code>ASSERT_BEFORE_HEIGHT_ABSOLUTE</code> ; 0 = no expiry)
2 owner	the owner signs <code>sha256tree(cond itions)</code>	anything: refill, empty, close
3 top up	anyone	the coin re-creates itself with the same state and more mojos
4 expired	anyone, at or after <code>EXPIRY</code>	the remainder returns to <code>OWNER_PH</code> ; at most <code>RETURN_FEE</code> mojos, fixed by the owner at open and taken from this coin, may go to the farmer (<code>RESERVE_FEE</code>), so the return pays its own fee

The budget is opened through the standard singleton launcher, which creates exactly one child; its coin id is the budget's id, and every spend proves that its parent was the launcher or the budget's previous coin. Mojors sent to the budget's puzzle hash any other way cannot be spent by the agent, so a second coin can never become a second limit. A stolen agent key costs at most one window's limit, and the owner can empty the coin the moment they notice. The card carries `allowance = {id, owner_pk, agent_pk, limit_mojos, period_blocks}` with `spending: "within-limit"` ; anyone can find the current coin (hinted by the id) and check its puzzle. A variant whose agent key is a secp256r1 passkey is `agent/namid_agent_allowance_pk.clsp`.

13. Payments to a name

13.1 The payment address

A payment to a name goes to the address in the name's `address` record, the one address its owner, proved by signature when the record was written, says to pay. A root is not payable through this record: a wallet resolves a root's deed directly.

13.2 Address proof levels

A wallet that turns a name into an address **MUST** state how the address is known (`pay_to.py` , schema `namid-pay-to/1`):

Level	How	Meaning
chain / deed-key	the address is the standard wallet puzzle (<code>P2_MOD_HASH = e9aaa49f45bad5c889b86ee3341550c155cfd10c3a6757de618d20612fffd52</code>) of the key carried into the name's deed on chain now	nobody but the key that controls the name can have that address
chain / anchored-record	the deed's <code>DATA_ROOT</code> on chain is the sha256 of the name's records document, and that document names this address's puzzle hash	the chain commits to the address
registry	anything else	the owner's signed record, not something the chain states; MUST NOT be shown as verified

Address changes. A changed payment address is what a payer most needs to hear about, because a stolen key repoints the address first. A wallet **SHOULD** warn when the current address took over within the last `RECENT_DAYS = 7` days, and **MUST NOT** describe a name first seen after tracking began as "never changed".

13.3 The nm1 memo standard

Transactions built by Protocol software say what they are in one short memo (`standards/NAMID-MEMOS.md` ; `namid_memos.py` , with a generated TypeScript twin and a drift test).

- **Format** `nm1:<kind>[:<ref>][#w3u]` : ASCII, at most 64 bytes, **never exactly 32 bytes** (padded with `~`), so it can never be mistaken for a hint.
- **Never displace a hint.** In (`51 puzzle_hash amount memos`) , a 32-byte first memo is the hint wallets use to find a coin; the `nm1` memo goes after it.
- **kind** is from a closed list, lowercase `[a-z-]` , at most 12 characters. Among them: `reg` , `reg-many` , `renew` , `commit` , `sealed-set` , `key` , `deed` , `records` , `price` , `pay` , `pay-many` , `payout` , `budget-open` , `budget-add` , `budget-out` , `budget-pay` , `budget-back` , `chan-open` , `chan-close` , `reveal` , `bid-back` , `settle` , `anchor` , `fee` .
- **ref** is optional and public: a name, `<from>><to>` names, a root and a count (`<root>*<count>`), or the **first 8 hex** of a public id. It uses lowercase letters, digits and the characters `.` `@` `>` `_` `-` and the asterisk, at most 52 characters, with **no run of 17 or more hex characters**.
- **#w3u** marks a transaction in which the service operator is a *party* (it pays, is paid, or its key signs); it is not set merely because operator software built a user's transaction.
- **A payment to a name** carries `nm1:pay:<from>><to>` (or `nm1:pay:<to>`).
- **A sealed bid carries no memo at all** (§5.2). The kind `bid` is never written.
- A decoder that meets `nm2:` reports a newer memo version and never guesses. Where a puzzle writes the coin (registrar nodes, budget payments, auction settlement), the memo is whatever the puzzle writes; changing a puzzle to add a memo would change its address. A memo costs about `(bytes + 3) × 12,000` cost; a large batch carries one memo, not one per coin.

14. Payment channels

Names pay each other off chain through two-party channels that settle on chain. This section specifies the channel puzzle and its rules only.

14.1 The channel coin

A channel coin (`payment-channel/namid_channel.clsp`) carries the two parties' keys, a dispute window `DISPUTE` (blocks), a channel tag `TAG` , and `NONCE` and `STATE_HASH` (both 0 while nothing is posted). A **state** is the whole balance sheet:

```
state = (nonce bal_a bal_b ph_a ph_b htcls)      htcls = ((to_b amount payment_hash
expiry) ...)
TAG    = sha256("namid-channel/1", genesis challenge, key_a, key_b, salt)
```

Both parties sign `sha256(TAG, sha256tree(state))` , so a state never replays in another channel or on another network.

Mode	Who	What
1 cooperative close	both sign (<code>AGG_SIG_ME</code> , bound to this coin)	pays both balances now; allowed at any stage, also during a dispute
2 update (post or override)	anyone holding a co-signed state with a higher nonce than the coin's	the coin re-creates itself at that nonce (same value, hinted with <code>TAG</code>); the dispute window (<code>ASSERT_HEIGHT_RELATIVE DISPUTE</code>) starts again
3 settle	anyone, no signature	only with the posted state, only <code>DISPUTE</code> blocks after it was posted; pays both balances and one hash-lock coin per lock

14.2 Hash-locked transfers

Each lock settles into a `payment-channel/namid_htlc.clsp` coin: the receiver claims with the preimage **before** height `EXPIRY` (`ASSERT_BEFORE_HEIGHT_ABSOLUTE`); the sender takes it back **at or after** `EXPIRY` . The two windows do not overlap, so a forwarding party always knows until when the next hop can still claim. A payee holds a secret `r` ; the payer locks to `h = sha256(r)` , taken from a signed invoice (`namid-invoice/1`) or from hashes the payee published in advance. Because each hop's lock releases only with the same preimage, **an intermediary can never keep a payment it forwards**.

14.3 Settle-ability

A state is checked for settle-ability when it is posted: 32-byte payout puzzle hashes, no negative amounts, at most 40 hash locks, 32-byte payment hashes, expiries below 2^{32} , and totals equal to the coin. No co-signed state can lock money for good. Equal payouts to one puzzle hash become one coin, and every hash-lock coin is carried with its index, so no two outputs collide.

14.4 Disputes

The newest co-signed state always wins: a post with a higher nonce overrides a stale one and restarts the window. Nothing but the latest state needs to be stored, and there are no revocation secrets. The dispute window is curried per channel; channels opened since 2026-10-18 use **64 blocks** (`DISPUTE_NEW` , `tools/namid-agent/namid_channels.py`). Anyone may run a watcher for channel disputes: holding only the latest co-signed state, it can answer a stale post, but it cannot start a close or move money. A cheating post costs its author only the fee.

14.5 Off-chain updates

Every update is a list of operations applied to the latest co-signed state: `add` (lock an amount to a payment hash), `settle` (the receiver claims with the preimage), `fail` (give the lock back) and `push` (a plain payment to the other side). Both sides recompute the next state from the operations, and each side signs only operations that cannot take its own money (`channel_core.acceptable`): the other side may never lock or move my balance, and I never give back a payment I can claim. A variant whose party A is a secp256r1 passkey is `payment-channel/namid_channel_pk.clsp` .

15. Mail and call addressing

This section specifies how mail and calls are addressed to a name; transport, storage and bridging to internet mail are out of scope.

- **Key publication.** A name publishes a service record holding an endpoint, routing keys and an X25519 key-agreement key; the record's hash (`service_hash`) is committed into the name's `DATA_ROOT` by an ordinary records update (`names2names_api.py`). A key that is not committed is not the name's key.
- **Addressing.** Mail is addressed `<local>@<name>` ; a call is placed to the name itself.
- **Confidentiality.** A mail body, and every call signalling envelope, is sealed to the recipient's published X25519 key (anonymous encryption) before it leaves the sender. A relay or mailbox host sees only routing metadata (who, to whom, when, how large), and the Protocol claims nothing more.
- **Sender authentication.** A message or call is accepted as *from* a name only with a signature, by the key that controls that name, over a fresh challenge bound to the sending name and the action (`mail` , `call`). A From line or caller id that is not proved this way is not shown as the name.
- **Reading is proved too.** Draining a mailbox or a call inbox requires the same ownership proof: call signalling contains network addresses, and an open read endpoint would disclose where a name is.
- **Bounded relays.** Call signalling relays are ephemeral and bounded (`ENVELOPE_TTL = 90` seconds, `call_relay.py`); media flows peer to peer. Mail stores are bounded per mailbox and per message and keep messages for a limited retention window (`tld_mailbox.py`).

16. Governance and policy

16.1 No token; one treasury

The Protocol has no coin of its own; every value flow is XCH. Every sale — auction settlement, treasury allocation, early-open payment, root renewal — pays **100% to the protocol treasury** (`PROCEEDS_TO_TREASURY = 1.0` ; the retained names `PROCEEDS_SPLIT_WEB3UTXOS = 1.0` and `PROCEEDS_SPLIT_ECOSYSTEM = 0.0` state the same policy). The split is enforced on chain, not by configuration: the bid escrows curry the treasury puzzle hash into both proceeds slots (`tldbase_chain.escrow_puzzle_v2_treasury_only`), so the chain pays one address —

txch19sj7wkz24ehd4zp7v4qwl7yg2qhh06unv0vr9v28h2u6krccrlqn0hhqe on testnet11 and xch1qlw3ffqa4yq78g2p6rrz35r37vk0dl2gfjtmkjmm40kafh97u9syhs58e on mainnet. Bidder refunds remain key-bound covenant outputs and are never revenue. A registrar's TREASURY_PH is set by its root owner (§8.4).

16.2 Governance

Testnet governance is a signed threshold log. Members sign domain-separated BLS proposal and vote messages; the log stores public keys and signatures, never private keys. Quorum authorises an execution draft; a proposal is recorded as executed only when a real transaction id is recorded and later confirmed by an independent chain read — approval alone is never presented as a chain spend. A chain-anchored governance singleton is a precondition for mainnet (§19).

16.3 Parameters

Parameter	Constant	Value	Notes
Release window	RELEASE_WINDOW_WEEKS	260	\$4; the seed changes only by signed decision
Daily auction cap	DAILY_CAP_N	20 (derived from WEEKLY_SLOTS = 140)	\$4; not yet ratified (\$19)
Bid window	AUCTION_BID_DAYS	10 days	\$5.1
Reveal window	AUCTION_REVEAL_DAYS	5 days	\$5.1
Reserve floor	NAMID_BASE_PRICE_USD	10 USD in XCH	\$5.4; the schedule above the floor is open (\$19)
Reserve price	RESERVE_PRICE_XCH	the floor while PRICING_TBD	\$5.4
Early-open multiplier	EARLY_OPEN_MULTIPLIER	10	\$5.5
Early-open bypasses queue	EARLY_OPEN_BYPASSES_QUEUE	true	\$5.5
Early-open consumes a daily slot	EARLY_OPEN_CONSUMES_DAILY_SLOT	false	\$5.5
Lease term	LEASE_YEARS	3	\$6.1
Renewal price per year	RENEWAL_PRICE_XCH	0.01 placeholder	\$6.2 (\$19)
Lapse grace	LAPSE_GRACE_DAYS	0	\$6.3
Redeemable until a re-auction opens	LAPSE_REDEEMABLE_UNTIL_AUCTION_OPENS	true	\$6.3
Redemption by the owner only	LAPSE_REDEEM_OWNER_ONLY	true	\$6.3
Treasury proceeds	PROCEEDS_TO_TREASURY	1.0	\$16.1
Root clawback window	CLAWBACK_BLOCKS	9216 blocks	\$3.5
Deed rotation delay	DID_ROTATION_DELAY	9216 blocks (96 on testnet11)	\$3.5
New-bid deed	NEW_BID_DEED	"did"	\$5.4; a protocol constant, not a dial
Channel dispute window	DISPUTE_NEW	64 blocks	\$14.4

16.4 The root owner's rules come first

A root's owner decides the rules of its namespace: which labels are reserved or blocked, how a name is priced and launched, how long a name lasts and whether it can change hands. They are written in the root's **rule book**

(`ns-policy` , `standards/tld-records.md` §5; `namespace_rules.py`), hashed in canonical form and committed when the root is locked.

Precedence. (1) The owner's rule book — or, where the owner delegates in it (`authority.by = "registrar"` , a named registrar and a named scope), that registrar within that scope. Until the owner writes a rule book, the owner's published root profile (`standards/root-profiles.json`) is the rule book. (2) A registrar's own settings, only where the rule book is silent or delegates. (3) Protocol defaults. Every implementation that issues, displays or prices names under a root reads the rule book and applies it.

The registrar carries the rules (§8.4, §8.7). There are three kinds of name term (`name_terms.KINDS`):

Kind	Term	On chain
<code>termed</code>	3 years, renewable	<code>TERM_BLOCKS</code> = 3 years of blocks; <code>TRANSFERABLE</code> per the rule book
<code>soulbound</code>	111 years	a soulbound deed; the term is the registrar's <code>TERM_BLOCKS</code>
<code>permanent</code>	never expires; "Permanent — yours forever"	<code>TERM_BLOCKS</code> = 0 and <code>TRANSFERABLE</code> = 0 : an insert carries expiry 0, renewal and takeover are refused by the puzzle, an update cannot change the owner, and a deed-bound name needs a soulbound deed

Permanent, soulbound service roots. A root owner **MAY** fix the `permanent` kind for every name under its roots. The roots the Protocol's initial operator, Web3UTXO LLC, runs for typed services (`roots.SERVICE_ROOTS`) use this profile (`root-profiles.json` , `web3utxos_sld_rule` — a code identifier that keeps the earlier spelling; read by `root_profiles.sld_rule()` , `name_terms.kind_for_root()` , `namespace_rules.registrar_terms()` and `asset_status.issuance()`), so the identity behind such a name — including an agent's — never lapses and never changes holder; its owner can still change its key behind the rotation delay and point it at new records. Exceptions are declared per root (`exceptions.roots` , the root keeps its own profile; `.chia` is one) or per label pattern under a root (`exceptions.labels`); because a registrar's term and transferability are fixed for every name it holds, a label exception needs a registrar generation of its own, and until one exists a matching label is refused (`web3utxos-exception`).

16.5 The owner pays

The owner pays every fee of the owner's own transactions: inserts, renewals, key changes, sweeps and returns. A service never pays a user's network fee from its own wallet; the only acceptable exception is a fee taken from an on-chain coin the owner funded for that purpose, such as a budget's `RETURN_FEE` (§12.4).

17. Security considerations

- **Not audited.** No puzzle in this document has been independently audited; nothing runs on mainnet before an independent audit. Registrar v7 is a draft (§8.8).
- **Key compromise.** Every key change is public and delayed (§3.5); a compromised key can be stopped but, without a separate recovery key, not settled. A stolen agent key costs at most one window of its budget (§12.4).
- **Frozen deeds.** A DID recovery by software that does not know the covenant freezes the deed (§3.2). Registrars refuse frozen deeds; the holder must repair the deed before acting.

- **Manifest trust.** The client manifest names the current generation of each root, its genesis head and its sealed-set anchor (§11.6). A forged manifest can point a client at the wrong list; clients **SHOULD** pin manifests and compare them across independent sources. An anchored index root is an operator's statement, not proof of honesty (§10.3).
- **Sealed sets under v6.** Enforcement is by resolvers only; a resolver that reads only the registrar list could show a squatter's node (§9.4).
- **Payment redirection.** A stolen key repoints the `address` record first; wallets warn on recent changes and state the proof level (§13.2).
- **Memo privacy.** Bid deposits carry no memo; a memo `ref` never carries a run of 17 or more hex characters, so a full coin id or key cannot be published by accident (§13.3).
- **Channels.** A party that stays offline longer than the dispute window can lose to a stale post; anyone may answer one on its behalf (§14.4). At most 40 locks per state bound the cost of a settlement (§14.3).
- **Homographs.** The label grammar admits only `a-z0-9`, removing confusable scripts and second spellings (§3.1).
- **Certificates.** A TLSA mismatch is a hard failure, never a WebPKI downgrade (§10.4).
- **Network confusion.** A client **MUST** check the network (testnet11 or mainnet) before building or signing, and **MUST** sign only the exact spend it displayed. Every chain-touching response from a service is an unsigned draft (§5.6).
- **Reorganisations.** A resolver reads below a confirmation depth (§11.1); deeper reorganisations are outside its guarantees.
- **Metadata.** Encryption protects mail bodies and call signalling, not who communicates with whom, when and how much (§15).

18. Conformance

An implementation conforms to this specification in one or more roles. In each role it **MUST** satisfy every requirement listed for that role.

Resolver.

1. **MUST** canonicalise and validate labels exactly as `standards/namid-label-grammar.json` specifies, including its vectors (§3.1).
2. **MUST** recognise registrar nodes only by lineage from the genesis head named in its manifest, recomputing every child it accepts (§8.2, §11.1).
3. **MUST** verify every record against the `DATA_ROOT` of the deed's current unspent coin, and **MUST** return a typed outcome or failure instead of an unverified answer (§11.3).
4. **MUST** apply a root's sealed set when the root has one, and **MUST NOT** report a label as free without a non-membership proof (§9.4).
5. **MUST NOT** treat a server's `verified` claim as verification, and **MUST NOT** require an operator's server for ownership, uniqueness or records (§11.4).
6. **SHOULD** bind caches to `(deed coin id, DATA_ROOT, height)` (§11.5).

Writer (software that builds registrations, record updates or key changes).

1. **MUST** insert only into the current, conforming generation of a root's registrar (§8.7), and only inside the correct gap (§8.2).
2. **MUST** carry the root's terms (expiry 0 under `TERM_BLOCKS = 0`; no change of owner under `TRANSFERABLE = 0`) (§16.4).

3. **MUST** have the owner pay the fees of the owner's transactions (§16.5).
4. **SHOULD** attach an `nm1` memo after any hint, never displacing it, and **MUST NOT** attach any memo to a bid deposit (§13.3).

Wallet (software that pays a name or signs for a name).

1. **MUST** state the proof level of an address and **MUST NOT** show a `registry`-level address as verified (§13.2).
2. **SHOULD** warn when a name's address changed within `RECENT_DAYS` (§13.2).
3. **MUST** display, then sign exactly, every spend it signs, after checking the network (§17).

Channel party.

1. **MUST** sign only states whose operations cannot take its own balance (§14.5) and that pass the settle-ability checks (§14.3).
2. **MUST** keep the latest co-signed state until the channel is settled, and **SHOULD** watch the chain (itself or through a watcher) for stale posts within the dispute window (§14.4).

Mail or call endpoint.

1. **MUST** encrypt to the recipient's key committed in its `DATA_ROOT`, and **MUST NOT** use a key that is not committed (§15).
2. **MUST** require a signature by the key controlling the sending name before showing a message or call as from that name (§15).

19. Open questions

These are undecided. This document does not decide them.

1. **Reserve-price schedule above the floor.** The floor is decided (10 USD in XCH, §5.4); whether the schedule above it is tiered by length, by phase or otherwise is open. The engine reads the reserve at process start, because the escrow covenant carries it.
2. **Renewal price.** `RENEWAL_PRICE_XCH` is a placeholder while `PRICING_TBD`.
3. **Daily cap.** `DAILY_CAP_N` is derived from the release window (20 per day) and awaits ratification.
4. **Registrable-domain claims.** Whether and how the operators of class (b) registrable domains claim their reserved roots is a governance decision (§7).
5. **Mainnet cutover.** Mainnet timing, ratification of the v2 release seed, the enforcement cutover from the v1 grandfather window, and a chain-anchored governance singleton are gates before mainnet.
6. **Registrar v7.** Merging the passkey-deed and sealed-set branches into one `TERMS` layout, and its audit, before any v7 generation serves a root (§8.8).
7. **Sealed sets.** Whether the sealed-set mechanism stays in the Protocol or is removed in a later version (§9).
8. **Contested keys.** A recovery path (a separate recovery key, m-of-n) that settles a contested key rather than only delaying it (§3.5).
9. **Generation selection.** A way for a client to learn the current generation of a root from the chain alone, rather than from a manifest (§11.6).
10. **Other assets.** Payment to a name, and agent budgets, in a CAT (such as a stablecoin on Chia) rather than XCH.

20. Changelog

v1.5.1 (2026-10-20). Company name: the Protocol's initial operator is Web3UTXO LLC (New Mexico, USA). Code identifiers cited verbatim (`PROCEEDS_SPLIT_WEB3UTXOS` , `web3utxos_sld_rule` , `web3utxos-exception`) keep their names. No change to any mechanic.

v1.5 (2026-10-19). The document is now a protocol specification only. Application, service and deployment material is removed, along with comparisons with other naming systems, project history and operating statistics. New sections: Conventions and terminology (§2), Registrars (§8), Sealed sets (§9), Payments to a name (§13), Payment channels (§14), Mail and call addressing (§15), Security considerations (§17) and Conformance (§18). Corrections to match the code: a lapsed root is redeemable until its re-auction opens, by its owner only (`LAPSE_REDEEMABLE_UNTIL_AUCTION_OPENS = true` , `LAPSE_REDEEM_OWNER_ONLY = true` ; v1.4 said redemption closed at the lapse); the daily cap is 20, derived from the release window (v1.4 said 10); the deed rotation delay is stated (`DID_ROTATION_DELAY`); the release seed is given as bytes. Registrar v7 is described as a draft with two switchable terms; v6 stays current.

v1.4 (2026-10-19). A name is registered only when it is on chain in its own coin; every root that issues names has an on-chain registrar, open or owner-only; chain-only resolution and registration; payment to a name with address proof levels and the `nm1` memo; payment channels with hash locks and a 64-block dispute window; passkey-owned deeds (registrar v7, simulator only).

v1.3 (2026-10-12). The root owner's rules come first; agent cards and agent spending budgets; registrars v2 (key changes separate from transfers), v3 (fixed node amount) and v4 (one key per name: deed-bound owners).

v1.2 (2026-10-11). Sealed bids committed in the bidder's own wallet (`tldbase_bid_v5c.clsp`).

v1.1 (2026-10-10). The specification published as the whitepaper.

v1.0 (2026-07-19). First specification: the singleton name, the release schedule, sealed-bid auctions, early-open, reserved classes, records and fail-closed resolution.

Companion standards: `standards/namid-label-grammar.json` (label grammar), `standards/DID-NAMID-METHOD.md` (`did:namid`), `standards/NAME-RECORDS-SCHEMA.md` and `standards/tld-records.md` (records), `standards/NAMID-ASSET-STATUS.md` (asset status), `standards/AGENT-CARD.md` (agent cards), `standards/NAMID-MEMOS.md` (memos), `standards/SPINE-VERSIONS.md` (puzzle spines), `standards/root-profiles.json` (root profiles), `standards/reserved-icann.txt` , `standards/reserved-popular-sites.tsv` and `standards/blocked-ofac.tsv` (reserved classes).