X-ID Whitepaper

Privacy-first identity, credentials, and data rights for the AI era

X-ID Project | April 2026 | Early Access Beta

Version 2.0

Contents

---

1. Executive Summary

X-ID is a privacy-first decentralized identity platform. It gives individuals, organizations, and developers a single cryptographic identity that they own outright, a credential wallet for the claims made about them, and zero-knowledge tooling to prove facts without revealing the underlying data. The goal is simple: put people back in control of their identity and their personal data, with verifiable receipts that hold up in court, in clinic, and at the door of any modern online service.

As of April 2026, X-ID is in **Early Access Beta** on Flare Network's Coston2 testnet. Six smart contracts are deployed and verified on Blockscout, four production Groth16 zero-knowledge circuits are live, and the full platform — DID resolution, verifiable credential issuance and verification, FHIR-grade medical records, consent anchoring, multi-chain portfolio view, SIWE authentication with passkeys — runs on a six-container Docker stack behind `app.x-id.xyz`. The public API exposes 77 routes across identity, credentials, proofs, consent, payments, and governance.

X-ID is built for four overlapping audiences. **Individuals** get a single sign-in, a wallet of verifiable credentials, and selective disclosure via ZK proofs. **Organizations** issue employment, membership, and access credentials with cryptographic revocation. **Healthcare providers** use a FHIR-native credential pipeline

that can prove clinical facts without handing over the whole record. **Developers** get a REST API, SIWE-based auth, and an OID4VP verification SDK — all against open W3C standards rather than a proprietary walled garden.

This paper describes what the platform does today, what is on the immediate roadmap (Flare mainnet launch, mobile wallet, organization admin console), and what remains deliberately out of scope until a formal security audit is complete. It is written to be verified: every contract address resolves on Blockscout, every technical claim maps to running code, and every roadmap item is phased honestly. X-ID is a beta today. It is designed to become the identity layer for a more private, more portable internet.

---

2. Legal Disclaimer & Risk Notices

This document is for informational purposes only. It is not a prospectus, offer to sell, or solicitation to purchase any security, token, or financial instrument. X-ID is currently deployed on a public test network (Flare Coston2, chainId 114). Testnet tokens have no monetary value and must not be treated as such.

The platform is in Early Access Beta. Features described as "live" are operational on Coston2 but have not undergone third-party security audit. A formal audit of the smart-contract suite and of the backend cryptographic code is planned prior to Flare mainnet launch.

Nothing in this document should be construed as legal, financial, medical, or regulatory advice. Identity regulation varies by jurisdiction; operators and relying parties are responsible for their own compliance (KYC/AML, data protection, medical record handling, accessibility).

Forward-looking statements — including roadmap items, planned integrations, and anticipated capabilities — reflect current intent and are subject to change. The authors and contributors make no warranty, express or implied, regarding the platform's availability, correctness, or fitness for any particular purpose. Use of the software is governed by the repository license (see Section 13).

---

3. Problem: Identity is broken

3.1 Fragmented — 200+ logins per user

The average connected adult now juggles well over two hundred separate online accounts. Each one is its own silo: its own username, its own password recovery path, its own privacy policy, its own data breach history. None of them talk to each other. None of them have a reliable way to prove "this is the same person" without handing over a phone number, an email, or a government-issued document.

The result is a credential sprawl problem that every user solves privately, badly, at their own risk. Password managers help but they do not change the underlying architecture: identity is still owned by whichever service you happen to be logging into.

3.2 Surveilled — data mining at every platform

Because every account is siloed, every service has an incentive to collect and retain as much data about its users as it can get away with. That data is then sold, rented, breached, or fed into training pipelines that the user never consented to. "Free" services are free because the user is the product.

Regulation (GDPR, CCPA, HIPAA, and their analogues) has put some guardrails on the worst behavior, but enforcement is slow and the default posture is still collect-first, justify-later. Users have almost no practical recourse when their data is misused — they rarely even find out.

3.3 Un-portable — your verified identity doesn't move with you

If a doctor's office verifies your identity, a bank verifies your address, and an employer verifies your right to work, none of those verifications are portable. The next service you use will perform its own verification from scratch, often using the same documents, exposing them again, and creating another copy of sensitive data in another database.

This is wasteful, invasive, and structurally impossible to fix without changing who issues and who stores the proof. Verification belongs in the user's pocket, not in the issuer's CRM.

3.4 AI-era: no opt-in/out for training data

Generative AI has sharpened the problem. Personal data, medical notes, creative work, and biometric signals are increasingly ingested into model training sets with no meaningful user consent. Opt-out mechanisms, where they exist at all, are piecemeal, service-specific, and generally post-hoc.

What is missing is a primitive: a user-controlled, cryptographically verifiable statement of consent (or refusal) that a data processor can check before using personal data, and that can be audited after the fact. Today there is no standard way to say "yes, you may train on this" or "no, you may not" in a form that a machine and a court can both understand.

---

4. The X-ID approach

X-ID is built around five design commitments. None of them are novel in isolation; the platform's contribution is implementing them together, using open standards, in code that people can actually run.

4.1 One cryptographic identity (DID)

Every X-ID user has a **Decentralized Identifier** (DID) under the W3C DID specification. The DID is a cryptographic key pair the user controls. It is not assigned by X-ID. It does not expire when X-ID stops paying its bills. It resolves to a DID Document that describes the user's public keys, service endpoints, and proof methods.

X-ID supports four DID methods at launch: `did:ethr` (Ethereum-family chains, including Flare), `did:xrpl` (XRP Ledger), `did:web` (domain-anchored, good for organizations), and `did:key` (self-contained, good for throwaway or offline use). The resolver is pluggable: additional methods can be added without touching the rest of the platform.

4.2 Credentials you issue, hold, revoke

Statements about a user — age, employment, insurance, medical history, academic qualifications — are modeled as **Verifiable Credentials** under the W3C VC Data Model 2.0. A credential is issued by a trusted party (an employer, a university, a clinic, a government), signed cryptographically, and stored in the user's wallet. The user, not the issuer, decides where and when to present it.

X-ID issues and verifies JWT-VC credentials using `did-jwt@8` and `ethr-did-resolver@11`. Issuance and presentation use the OpenID for Verifiable Credentials specifications — OID4VCI for issuance and OID4VP for presentation — so credentials can flow between X-ID and any compliant wallet or verifier, not just X-ID's own UI.

4.3 Zero-knowledge disclosure

Most identity checks do not need the underlying data. A bar checking for adulthood does not need your date of birth; it needs "yes/no, over 21". A pharmacy confirming insurance does not need the policy; it needs "yes/no, active". X-ID ships four production Groth16 circuits, compiled and deployed with a `pot14` powers-of-tau ceremony, to answer exactly these questions:

- `ageProof` — prove "over N years old" without revealing date of birth
- `insuranceProof` — prove "active insurance" without revealing policy number, carrier, or plan
- `vaccinationProof` — prove "received N doses" without revealing dates or provider
- `conditionProof` — prove "this condition is NOT in my records" without revealing the records

Proofs are generated client-side with `snarkjs` and WebAssembly witness generation; verification happens on-chain against deployed verifier contracts or off-chain via the verification SDK. The user's private data never leaves their device.

4.4 Consent as a first-class primitive

Every time a user shares data — a credential, a record, a proof — X-ID can anchor the consent event as a hash on the **ConsentAnchor** contract. The on-chain record is immutable, timestamped, and auditable. The data itself stays off-chain; only the receipt is public.

This matters in three places. First, it gives users a tamper-evident log of what they have shared and with whom. Second, it gives relying parties (and their auditors) cryptographic proof that consent was obtained at a specific moment. Third, it provides a machine-readable substrate for AI-era consent: a model trainer can check, cheaply, whether a dataset's contributors have granted permission — and a contributor can revoke it with a subsequent on-chain event.

4.5 Self-custody by default

Private keys are generated and held on the user's device. They never touch X-ID's servers. Recovery is via user-chosen mechanisms — passkeys bound to hardware, TOTP, social recovery in future releases — not via an X-ID password reset. If X-ID disappears tomorrow, users' identities and credentials continue to function against any compliant resolver.

The trade-off is deliberate. Self-custody is harder to support than custodial identity; it requires better UX, better recovery flows, and more user education. The alternative — X-ID holding the keys — would replicate the exact problem the platform is built to solve.

4.7 Compared to alternatives

It is worth being explicit about where X-ID sits in the landscape. Identity is an old problem with a lot of prior art. The table below is an honest, subjective comparison of X-ID against the three categories of systems that solve adjacent problems today.

Capability	Traditional IAM	Big Tech ID	Existing SSI	**X-ID**
User owns identity	No	No	Yes	**Yes**
Portable between services	No	No	Yes	**Yes**
Verifiable credentials	Partial (SAML)	No	Yes	**Yes**
Selective disclosure	No	No	Partial	**Yes**
Zero-knowledge proofs (live)	No	No	No	**Yes**
Consent ledger on-chain	No	No	No	**Yes**
FHIR-native health credentials	No	No	No	**Yes**
Works without crypto knowledge	Yes	Yes	No	**Yes**
Open source	Partial	No	Yes	**Yes**
Production-ready today	Yes	Yes	Partial	Early Access Beta

The honest reading of this matrix is that Okta, Auth0, and the big-tech sign-in products are more production-proven today than X-ID, full stop. They run the bulk of the world's enterprise and consumer authentication. Their weakness is not reliability; it is their architecture. Identity in those systems is issued, revoked, and monetised by a provider that is not the user. That is fine for convenience, but it is the exact architecture that produces the fragmentation, surveillance, and non-portability problems described in Section 3. Existing SSI platforms — uPort's successors, Veramo's agent framework, TBD's Web5 stack — share X-ID's philosophical commitment to user-held keys and verifiable credentials, and in some cases pre-date us. Where they tend to fall short is the integrated product layer: ZK proofs as first-class primitives, a FHIR-aware credential pipeline, an on-chain consent ledger, and a developer-grade REST API wrapped around all of it. Assembling those pieces ad hoc is possible but expensive.

X-ID's distinct contribution is the combination. SSI primitives (DID plus VC plus ZK) are necessary but not sufficient; the user experience, the clinical integration, the on-chain audit layer, and the developer APIs are what turn the primitives into a usable platform. The SSI community has been laying the groundwork for a decade. X-ID's bet is that 2026 is the year those primitives finally ship together, behind a single wallet, against open standards, on a chain with meaningful oracle infrastructure. The comparison table is not a claim of superiority in every row — it is a claim that the intersection of rows marked `Yes` under X-ID does not currently exist elsewhere in shippable form.

---

5. Architecture

5.1 System overview

X-ID runs as a six-container Docker compose stack behind a Traefik reverse proxy. The components are separated by concern: identity resolution, credential issuance, proof generation, medical records, payments/governance on-chain, and the user-facing web application.

```
flowchart TB
    subgraph User["User device"]
        Browser[Browser / Wallet]
        Keys[Private keys + WebAuthn]
    end

    subgraph Edge["Edge / Traefik"]
```

```
    Traefik[Traefik reverse proxy]
end

subgraph Platform["X-ID platform (Docker compose)"]
    Frontend["Next.js 16 frontend<br/>React 19 + TanStack Query"]
    API["Node.js 22 ESM backend<br/>77 REST routes"]
    FHIR["HAPI FHIR sidecar<br/>medical records"]
    IPFS["Kubo IPFS 0.40.1<br/>content addressing"]
    Postgres["Postgres 15<br/>users, credentials, audit"]
    Redis["Redis 7<br/>sessions + cache"]
end

subgraph Chain["Flare Coston2 (chainId 114)"]
    Contracts["6 contracts<br/>Token/Consent/Payment/<br/>Staking/Escrow/Governor"]
    Verifier["Groth16 verifier<br/>contracts"]
    FTSOv2["FTSO v2<br/>price oracles"]
    FDC["FDC<br/>data connector"]
end

Browser -->|HTTPS| Traefik
Traefik --> Frontend
Frontend -->|REST + SIWE| API
API --> Postgres
API --> Redis
API --> FHIR
API --> IPFS
API -->|RPC| Contracts
API -->|RPC| FTSOv2
API -->|RPC| FDC
Keys -.->|signs SIWE,<br/>generates ZK| Browser
Browser -->|proof submission| Verifier
```

The platform is stateless at the API layer: Postgres holds durable state (accounts, credential metadata, audit log), Redis handles session and rate-limit state, IPFS stores content-addressed blobs (avatar images, credential attachments, user-owned web content), and the FHIR sidecar handles clinical records. On-chain state — consent anchors, payment escrow, governance votes — lives on Coston2.

5.2 Identity layer (DIDs, resolvers, methods supported)

Every account is keyed to a DID. On signup, the backend provisions a `did:ethr` tied to the user's Flare-compatible address; additional methods can be registered against the same account. The resolver layer is modular and currently handles:

Method	Use case
`did:ethr`	EVM-chain anchored (Ethereum, Flare, L2s)
`did:xrpl`	XRP Ledger anchored, useful for payment-bound IDs
`did:web`	Domain-anchored, ideal for organizations

Method	Use case
`did:key`	Self-contained, no network resolution required

The issuer DID for credentials signed by the X-ID platform itself is `did:ethr:114:0xC03BbD4C6349B34b3a14afB657DAE869ed` Third-party issuers (employers, clinics, schools) use their own DIDs and sign credentials with their own keys; X-ID never countersigns on their behalf.

5.3 Credentials layer (W3C VC, OID4VCI/VP, JWT-VC)

Credentials follow the **W3C Verifiable Credentials Data Model 2.0**. The concrete serialization is **JWT-VC**: a JSON Web Token signed by the issuer's DID-resolvable key. X-ID uses `did-jwt@8` for signing and verification and `ethr-did-resolver@11` to resolve issuer keys from on-chain DID documents.

Issuance flows follow **OID4VCI** (OpenID for Verifiable Credential Issuance). A user requests a credential from an issuer (e.g. their employer), completes the issuer's authentication step (e.g. SSO), and receives a JWT-VC bound to their DID. The credential is stored in the user's wallet — locally for self-custody, optionally backed up to user-chosen encrypted storage.

Presentation flows follow **OID4VP** (OpenID for Verifiable Presentations). A verifier (e.g. a service the user is signing up for) requests specific claims; the wallet constructs a Verifiable Presentation, optionally with selective disclosure, and signs it with the user's DID key. The verifier checks the signature and the issuer's reputation — no round-trip to X-ID required.

Revocation uses status list credentials: issuers publish a revocation bitmap, and verifiers check the relevant bit at presentation time. Revocation is the issuer's prerogative; X-ID does not revoke third-party credentials.

Web3 email, via `mailchain@0.30`, gives each DID an optional deliverable email address tied to the user's keys rather than to a provider account.

5.4 Proof layer (Groth16 circuits, on-chain verifier)

Zero-knowledge is implemented as **Groth16** SNARKs. Circuits are compiled with `circom`, the trusted setup uses `pot14` (powers-of-tau sized for up to 2^14 constraints, multi-party ceremony) contributions, and proof generation and verification run via `snarkjs`. Four circuits are deployed today:

Circuit	Predicate
`ageProof`	`dob_year + N <= current_year`
`insuranceProof`	`policy.status == "active" && policy.expiry > today`
`vaccinationProof`	`count(doses where type == X) >= N`
`conditionProof`	`condition X not in record set`

Witness generation runs in WebAssembly in the user's browser. Proofs are compact (a few hundred bytes) and verification is cheap, so verification can be performed on-chain (via a deployed Solidity verifier) or off-chain (via the JavaScript SDK).

A concrete example: a healthcare provider verifies a patient is over 18 via an `ageProof` ZK proof. The patient's date of birth never leaves the patient's device. The provider sees a boolean `true` and a cryptographic proof that can be audited later. No DOB is transmitted, stored, or logged.

Future circuits on the roadmap include residency proofs (for jurisdiction-restricted services), income-band proofs (for means-tested access), and multi-credential composition proofs. The architecture is a module, not a fixed set.

5.5 Consent layer (ConsentAnchor contract)

The **ConsentAnchor** contract (`0x4c384d28245012b6463340D02C28e19c5723058b`) is a minimal, append-only registry for consent events. Each entry is a tuple of `(user DID hash, verifier DID hash, scope hash, timestamp, revocation flag)`. The payload — what data, under what terms, for how long — lives off-chain in a user-signed JSON document; only the hash is anchored.

This design does three things. It lets users prove, later, that they did or did not consent to a specific action. It lets verifiers prove to an auditor that consent was on record before the action occurred. And it lets regulators sample the public record without needing access to underlying personal data.

Revocation is a second transaction: the user (or an authorized delegate) submits a new event with the revocation flag set, and any downstream processor is expected to check for revocations before relying on the original consent. There is no retroactive deletion — the point is an immutable audit trail.

5.6 Data layer (FHIR, IPFS, encryption)

Three storage systems sit behind the API.

Postgres 15 holds relational state: user accounts, credential metadata (never credential bodies), public DID documents, on-chain transaction receipts, rate-limit counters. Sensitive columns (recovery hints, email, FHIR resource references) are encrypted at rest via pgcrypto-style column encryption; secrets are rotated and held in 1Password rather than the repository.

Redis 7 holds ephemeral state: SIWE session tokens, challenge-response nonces, short-lived cache entries. No durable data lives in Redis.

IPFS via Kubo v0.40.1 holds content-addressed blobs: user-owned website content, credential attachments (e.g. a diploma PDF the issuer signed), avatar and profile images. IPFS is run as a real daemon, not a mock; content is pinned against user-owned retention policies. Large clinical attachments can be encrypted client-side before pinning, so IPFS never sees the plaintext.

HAPI FHIR runs as a sidecar and stores clinical resources — `Patient`, `Observation`, `Immunization`, `Condition`, `Encounter`, `DiagnosticReport`, `MedicationStatement` — in a FHIR-native schema. The FHIR sidecar is the source of truth for medical records; ZK circuits query it to derive proofs, and credentials derived from FHIR resources (e.g. a vaccination credential) carry a pointer back to the resource for provenance.

All server-side data is encrypted at rest. Backups are encrypted in transit and at rest. No plaintext secrets are stored in the repository — the git history has been scrubbed of historical leaks and active dependabot alerts stand at zero as of 2026-04-21.

5.6.4 Multichain portfolio via Routescan A design goal of X-ID is that the wallet surface shows the user's full on-chain footprint, not just their activity on Flare. In practice that means reading balances, token holdings, NFTs, and transaction history across dozens of EVM chains. Running a node or a full indexer per chain is the obvious way to do this, and it is the wrong way: operating 39 separate RPC connections, tracking 39 reorgs, and paying for 39 archive nodes produces an ops burden that would consume the entire team.

X-ID uses **Routescan's** aggregated explorer API as the read-only backbone for multichain portfolio queries. Routescan exposes a single HTTP surface that spans mainnet, testnet, L2, and app-chain EVM networks, returning consistent shapes for native balances, ERC-20 holdings, ERC-721 and ERC-1155 collections, and paginated transaction history. This lets `walletController.getPortfolio` answer a cross-chain portfolio request in one fan-out, cached at the API layer, without touching a chain RPC for the read path.

Write operations still go to per-chain RPCs, because Routescan is an explorer and not a transaction broadcaster. For Flare specifically we use the FDC and FTSO v2 endpoints directly for verifiable data and price feeds; for cross-chain writes the user's wallet is authoritative and the backend is a passthrough. The design principle is: read-only reach goes through the aggregator; state-changing calls go through the authoritative chain.

The supported chain families as of this writing:

- **Flare family:** Flare mainnet (14), Coston2 testnet (114), Songbird, Coston
- **Ethereum family:** Ethereum mainnet, Base, Arbitrum One, Optimism, Polygon PoS, BNB Chain, Avalanche C-Chain, Fantom
- **L2s and rollups:** zkSync Era, Linea, Scroll, Mantle, Blast, Mode, Zora
- **App-chains:** Polygon zkEVM, Arbitrum Nova, Moonbeam, Moonriver
- **Testnets:** Sepolia, Holesky, Goerli (legacy), Base Sepolia, Optimism Sepolia, Arbitrum Sepolia

The total is 39+ chains at any given time and grows as Routescan adds integrations; the list above is representative, not exhaustive. The `_source` field on every portfolio response identifies where the data came from, so downstream clients can surface "data from Routescan, last refreshed Xs ago" rather than present stale figures as authoritative.

An example of the shape returned by `GET /api/wallet/portfolio/:address`:


```
{
  "_source": "routescan",
  "_fetchedAt": "2026-04-21T14:32:11.224Z",
  "_cacheAgeMs": 4821,
  "address": "0xd1190ea34051835d8743E057C1F7BF7838BE895A",
  "chains": [
    {
      "id": 114,
      "name": "Flare Coston2",
      "nativeSymbol": "C2FLR",
      "nativeBalance": "92.5",
      "nativeBalanceUsd": "0.00",
      "txCount": 1284
    },
    {
      "id": 1,
      "name": "Ethereum Mainnet",
      "nativeSymbol": "ETH",
      "nativeBalance": "0.12",
      "nativeBalanceUsd": "347.28",
      "txCount": 56
    },
```

```
    {
      "id": 8453,
      "name": "Base",
      "nativeSymbol": "ETH",
      "nativeBalance": "0.004",
      "nativeBalanceUsd": "11.57",
      "txCount": 12
    }
  ],
  "tokens": [
    {
      "chainId": 114,
      "symbol": "XID",
      "address": "0x996605d357a5F71fB4da6FCF0C71457327dBE068",
      "balance": "10000",
      "decimals": 18,
      "valueUsd": "0.00"
    }
  ],
  "nfts": [],
  "totalUsd": "358.85"
}
```


The cache layer (Redis) fronts Routescan with a short TTL — 30 seconds for balance endpoints, 5 minutes for transaction-history pagination — so a user refreshing their wallet hits a cached response rather than the upstream explorer on every keystroke. The portfolio endpoint is safe to poll; Routescan itself is not the rate-limit bottleneck at realistic load.

A fallback matters. If Routescan is unreachable (an outage, a regional block, a rate-limit at the upstream), the endpoint degrades gracefully: `walletController.getPortfolio` falls back to per-chain reads for the Flare family (which we index directly) and returns a partial response with an `_partial: true` flag and a list of chains that could not be read. Clients surface this as a soft error so the user sees their Flare balance even when the broader multichain view is unavailable. The fallback is deliberate: we would rather show accurate partial data than pretend the other chains have zero balance. The same principle applies if Routescan changes its response shape — the parser is tolerant of unknown fields but strict about the fields we do use, so an upstream format change degrades to "this chain's data is temporarily unavailable" instead of a crash.

5.7 Auth (SIWE, Passkeys, MFA)

Primary authentication is **Sign-In With Ethereum** (SIWE, EIP-4361). The user signs a human-readable message with the private key of their DID-bound address; the backend verifies the signature and issues a session token. No passwords are involved on the happy path.

Second-factor authentication and phishing-resistant primary auth both use **WebAuthn / Passkeys**. Passkeys are bound to hardware (TPM, secure enclave, security key) and cannot be phished by a cloned site. For users whose devices do not yet support passkeys, TOTP (RFC 6238) is available as a fallback second factor.

Account recovery is user-chosen: a passkey on a second device, a hardware security key, or a TOTP seed held offline. Social recovery (M-of-N trusted contacts co-signing a key rotation) is on the roadmap.

5.8 Governance (XIDGovernor, staking)

On-chain governance uses the **XIDGovernor** contract (`0x6a2A4744f7A03b596AdCb168c14549E735c9d5f4`), an OpenZeppelin `Governor` with ERC-20Votes accounting delegated to XIDToken. Proposals are created, voted on, and executed by XID holders.

Staking uses the **StakingContract** (`0xf8F6022A68F64A64D208A2E9F51a95bed7813Bad`) with a 1,000 XID minimum stake and a 10% slash for proven misbehavior. Staking weight feeds both governance and the reputation layer for issuers and verifiers.

The governance design omits a timelock. The rationale is practical: testnet governance must iterate fast, and stakeholder override provides a safety net during the beta. Timelock semantics will be revisited before mainnet deployment; see Section 12.

5.9 End-to-end flows

The sections above describe layers in isolation. This section walks the five most load-bearing end-to-end flows in the platform — login, credential issuance, credential presentation, ZK proof composition, and consent grant/revocation — with pseudocode that maps directly onto the running services. Names of controllers, services, and routes match the actual backend.

5.9.1 SIWE login flow Login starts with the client requesting a fresh nonce. The backend stores the nonce in Redis with a short TTL and binds it to the user's address. The client constructs a standard EIP-4361 SIWE message, signs it with the wallet's private key, and posts the pair back. The backend verifies the signature against the address, issues a JWT, and registers a passkey if one is being set up in the same round-trip.

The over-the-wire shape of the three messages:

```
GET /api/siwe/nonce?address=0xd119...5A HTTP/1.1
Host: api.x-id.xyz

HTTP/1.1 200 OK
Content-Type: application/json

{ "nonce": "aBc7Fq...", "ttlSeconds": 300, "issuedAt": "2026-04-21T14:32:11Z" }
```

```
POST /api/siwe/verify HTTP/1.1
Host: api.x-id.xyz
Content-Type: application/json

{
  "message": "app.x-id.xyz wants you to sign in with your Ethereum account:\n0xd1190ea34051835d8743E057C1F7
  "signature": "0x8f2a...",
  "passkey": { "attestation": "...", "clientDataJSON": "..." }
}

HTTP/1.1 200 OK
```

```
Set-Cookie: xid_session=eyJhbGc...; HttpOnly; Secure; SameSite=Strict
{ "jwt": "eyJhbGc...", "did": "did:ethr:114:0xd119...5A", "expiresIn": 86400 }
```

The backend flow, mapped to real service names:

```
// authController.siweNonce
async function siweNonce(req, res) {
  const { address } = req.query;
  assertAddress(address);
  const nonce = cryptoRandomString(16);
  await redis.set(`siwe:nonce:${address.toLowerCase()}`, nonce, 'EX', 300);
  return res.json({ nonce, ttlSeconds: 300, issuedAt: new Date().toISOString() });
}

// authController.siweVerify
async function siweVerify(req, res) {
  const { message, signature, passkey } = req.body;
  const parsed = parseSiweMessage(message);                // strict EIP-4361 parser
  const storedNonce = await redis.get(`siwe:nonce:${parsed.address.toLowerCase()}`);
  if (!storedNonce || storedNonce !== parsed.nonce) throw new Err('NONCE_INVALID');
  await redis.del(`siwe:nonce:${parsed.address.toLowerCase()}`); // one-time use

  const recovered = recoverSiweAddress(message, signature);
  if (recovered.toLowerCase() !== parsed.address.toLowerCase()) {
    throw new Err('SIGNATURE_MISMATCH');
  }

  const user = await usersService.upsertByAddress(parsed.address);
  if (passkey) await webauthnService.registerPasskey(user.id, passkey);

  const jwt = jwtService.sign({ sub: user.id, did: `did:ethr:114:${parsed.address}` });
  res.cookie('xid_session', jwt, { httpOnly: true, secure: true, sameSite: 'strict' });
  return res.json({ jwt, did: `did:ethr:114:${parsed.address}`, expiresIn: 86400 });
}
```

Subsequent API calls include the `xid_session` cookie (or an `Authorization: Bearer` header for server-to-server clients). Middleware (`authMiddleware.requireSession`) verifies the JWT, loads the user, and attaches `req.user`. Sessions rotate on sensitive actions and can be force-invalidated by revoking the user's session family in Redis.

Three failure paths are worth calling out explicitly. First, if the SIWE message's `Chain ID` does not match our expected chainId, `siweVerify` rejects the request — this guards against a replay of a signature intended for a different network. Second, if the `Issued At` timestamp is more than 10 minutes in the past, the request is rejected even if the signature is otherwise valid, to narrow the replay window. Third, if the `Domain` in the message is anything other than `app.x-id.xyz`, the request is rejected; this is the same phishing guard that EIP-4361 codified after early SIWE deployments learned the lesson the hard way. A passkey registered in the same round-trip is bound to the session that created it, and subsequent sensitive actions (credential issuance, consent anchor, key rotation) require a WebAuthn assertion on top of the cookie.

5.9.2 Credential issuance (OID4VCI) OID4VCI decouples the issuer from the wallet. The issuer publishes a credential offer (often as a QR code or deep link), the wallet exchanges the offer for a short-lived access token, and the wallet then pulls the credential from a token-gated endpoint. The issuer signs the credential as a JWT-VC bound to the wallet's DID.

The issuer side, implemented in `credentialsService.issueVC`:

```
// credentialsController.createOffer  (issuer side)
async function createOffer(req, res) {
  const { subjectDid, credentialType, claims } = req.body;
  await assertIssuerPermission(req.user, credentialType);    // e.g. only orgs that own it

  const offerId = cryptoRandomString(24);
  await offersRepo.insert({
    id: offerId,
    issuerUserId: req.user.id,
    subjectDid,
    credentialType,
    claims,
    expiresAt: Date.now() + 10 * 60 * 1000,
  });

  return res.json({
    credential_offer_uri: `https://api.x-id.xyz/api/credentials/offer/${offerId}`,
    grants: { 'urn:ietf:params:oauth:grant-type:pre-authorized_code': {
      'pre-authorized_code': offerId,
    }},
  });
}

// credentialsController.tokenExchange  (issuer side)
async function tokenExchange(req, res) {
  const { 'pre-authorized_code': code } = req.body;
  const offer = await offersRepo.consume(code);              // one-time use
  if (!offer || offer.expiresAt < Date.now()) throw new Err('OFFER_EXPIRED');

  const accessToken = jwtService.sign(
    { offerId: offer.id, sub: offer.subjectDid },
    { audience: 'credentials-endpoint', expiresIn: '5m' },
  );
  return res.json({ access_token: accessToken, token_type: 'Bearer', expires_in: 300 });
}

// credentialsController.issue  (issuer side, token-gated)
async function issue(req, res) {
  const { offerId, sub } = req.accessToken;                  // from auth middleware
  const offer = await offersRepo.getById(offerId);
  const vc = await credentialsService.issueVC({
```

```
    issuer: ISSUER_DID,                                    // did:ethr:114:0xC03B...
    subject: sub,
    type: offer.credentialType,
    credentialSubject: offer.claims,
  });
  // vc is a signed JWT-VC string; did-jwt@8 under the hood
  return res.json({ format: 'jwt_vc_json', credential: vc });
}
```

The wallet side, running in the browser:

```
// walletClient.acceptOffer
async function acceptOffer(offerUri: string) {
  const offer = await fetch(offerUri).then(r => r.json());
  const code = offer.grants['urn:ietf:params:oauth:grant-type:pre-authorized_code']
                    ['pre-authorized_code'];

  const { access_token } = await postForm('/api/credentials/token', {
    grant_type: 'urn:ietf:params:oauth:grant-type:pre-authorized_code',
    'pre-authorized_code': code,
  });

  const { credential } = await fetchJson('/api/credentials/issue', {
    headers: { Authorization: `Bearer ${access_token}` },
    method: 'POST',
  });

  await walletStore.save({ credential, format: 'jwt_vc_json', receivedAt: Date.now() });
  return credential;
}
```

The credential body is a signed JWT. Verification against the issuer DID can happen locally, using `did-jwt` plus `ethr-did-resolver`, without contacting X-ID. That is the point: once issued, the credential is the user's.

Two design choices in this flow are worth naming. The pre-authorized code grant is used rather than the authorization code flow because credential issuance in our target use cases almost always follows an out-of-band authentication (an employer's SSO, a clinic's patient portal). Pre-authorization means the authentication burden is on the issuer, not on the wallet, and the wallet receives a code it can redeem without bouncing through another OAuth redirect. The second choice is that the credential format is declared in the response (`jwt_vc_json`) rather than inferred from the endpoint. This matters because the same issuer endpoint may in future issue `ldp_vc` (linked-data credentials) or `mso_mdoc` (ISO mdoc) alongside JWT-VC, and the wallet needs to parse the format declaration rather than guess from URL shape.

5.9.3 Credential presentation (OID4VP) OID4VP is the inverse flow: a verifier asks for specific claims, the wallet builds a Verifiable Presentation with selective disclosure, and the verifier validates it locally. No round trip through X-ID is required for verification.

Verifier side, implemented in `verificationController`:

```
// verificationController.createRequest
async function createRequest(req, res) {
  const { requiredClaims, acceptedIssuers, redirectUri } = req.body;
  const requestId = cryptoRandomString(24);
  await requestsRepo.insert({
    id: requestId,
    requiredClaims,                                   // e.g. ['age>=21']
    acceptedIssuers,                                  // ['did:ethr:114:0xC03B...']
    redirectUri,
    status: 'pending',
    createdAt: Date.now(),
  });
  return res.json({
    request_uri: `https://api.x-id.xyz/api/presentations/request/${requestId}`,
    client_id: verifierDid(req.user),
    response_mode: 'direct_post',
  });
}

// verificationController.submitResponse
async function submitResponse(req, res) {
  const { vp_token, presentation_submission } = req.body;
  const presentation = await verifyVP(vp_token, {          // did-jwt verifyPresentation
    audience: verifierDid(req.user),
    resolver: didResolver,
  });
  // presentation.verifiableCredential[] is a list of JWT-VCs, each already verified
  // against the issuer DID doc at this point

  const claims = extractClaims(presentation, presentation_submission);
  await consentService.anchorIfRequested(claims, presentation);
  return res.json({ verified: true, claims });
}
```

Wallet side:

```
// walletClient.buildPresentation
async function buildPresentation(requestUri: string) {
  const request = await fetch(requestUri).then(r => r.json());
  const creds = await walletStore.matchCredentials(request.requiredClaims,
                                                   request.acceptedIssuers);

  // selective disclosure: only reveal the fields the verifier asked for
  const disclosed = creds.map(c => selectivelyDisclose(c, request.requiredClaims));

  const vpJwt = await didJwt.createVerifiablePresentationJwt({
    verifiableCredential: disclosed,
```

```
    holder: userDid,
  }, signer, { audience: request.client_id });

  await postJson(`/api/presentations/response/${request.id}`, {
    vp_token: vpJwt,
    presentation_submission: buildSubmissionMap(disclosed, request.requiredClaims),
  });
}
```

Selective disclosure here means the wallet builds a VP that references only the fields needed. For richer claim-level redaction (e.g. "reveal `over21` but not `dateOfBirth`"), the flow composes with a ZK proof as in 5.9.4 rather than trying to edit signed credential bodies.

A subtlety worth calling out: the presentation is bound to the verifier by the `audience` claim in the VP JWT. A presentation created for Verifier A cannot be replayed against Verifier B, because Verifier B's own audience check will fail. This is why the `client_id` in the presentation request matters — it is not decorative metadata, it is the binding that prevents cross-verifier replay. The X-ID SDK refuses to submit a presentation whose audience does not match the verifier's DID, and the backend's `verifyVP` helper re-checks it on the server side as a defence in depth.

5.9.4 ZK proof composition ZK proofs are generated in the browser, not on the server. The user's DOB (or any other private input) never leaves the device. The typical sequence for `ageProof`:

1. The circuit's wasm witness generator and the proving key are fetched (and cached in `IndexedDB`) from the X-ID static CDN.
2. The wallet looks up the user's private DOB input from its encrypted local store.
3. `snarkjs.groth16.fullProve(inputs, wasmPath, zkeyPath)` produces a proof object and public signals.
4. The proof is submitted either to an on-chain verifier contract or to the off-chain verification SDK, depending on what the verifier asked for.

Browser-side composition:

```
// zkpClient.buildAgeProof
async function buildAgeProof({ minAge }: { minAge: number }) {
  const { dobYear } = await walletStore.loadPrivate('dob');
  const currentYear = new Date().getUTCFullYear();

  const { proof, publicSignals } = await snarkjs.groth16.fullProve(
    { dobYear, currentYear, minAge },
    `${CDN}/circuits/ageProof.wasm`,
    `${CDN}/circuits/ageProof_final.zkey`,
  );
  return { proof, publicSignals };   // publicSignals = [currentYear, minAge, 1]
}
```

On-chain verification. The deployed Solidity verifier is a function `verifyProof(a, b, c, input) returns (bool)`. The wallet (or a relayer acting for the wallet) submits a transaction that calls the verifier, and the caller contract reads the returned boolean.

```
// AgeProofVerifier.sol  (generated by snarkjs, Groth16)
function verifyProof(
  uint[2] calldata a,
  uint[2][2] calldata b,
  uint[2] calldata c,
  uint[3] calldata input   // [currentYear, minAge, resultFlag]
) external view returns (bool);
```

```
// On-chain path
const tx = await ageProofVerifier.verifyProof(
  [proof.pi_a[0], proof.pi_a[1]],
  [[proof.pi_b[0][1], proof.pi_b[0][0]], [proof.pi_b[1][1], proof.pi_b[1][0]]],
  [proof.pi_c[0], proof.pi_c[1]],
  publicSignals,
);
// tx is `true` iff the proof is valid and the user really is over `minAge`
```

Off-chain verification. For verifiers that do not want to spend gas per check, the same proof verifies in Node.js using `snarkjs.groth16.verify(verificationKey, publicSignals, proof)`:

```
// zkpService.verifyOffChain
async function verifyOffChain(circuit, proof, publicSignals) {
  const vk = await loadVerificationKey(circuit);           // from disk, pinned
  const ok = await snarkjs.groth16.verify(vk, publicSignals, proof);
  if (!ok) throw new Err('ZKP_INVALID');
  return { ok: true, circuit, publicSignals };
}
```

The choice between on-chain and off-chain is a policy decision by the verifier, not a capability decision: the same proof object works for both. On-chain gives you a tamper-evident public record of the check; off-chain gives you zero gas and full privacy. Verifiers that need both can do on-chain for the audit trail and off-chain as a fast pre-flight.

Performance is practical. Witness generation for `ageProof` takes roughly 200-400 ms on a typical laptop browser; proof generation is another 1-2 seconds. On-chain verification on Coston2 costs around 250k gas per call, well under the block gas limit. Off-chain verification via `snarkjs` is under 50 ms on a modern server. Clients cache the wasm witness generator and the proving key in `IndexedDB` after the first use, so subsequent proofs of the same circuit skip the network fetch entirely. Circuit parameters (the verification key hash) are pinned in the contract and in the SDK, so an attacker who substitutes a different proving key cannot trick the verifier into accepting a proof from a different circuit.

5.9.5 Consent anchor grant and revocation cycle Consent events are user-signed JSON documents hashed and anchored on the `ConsentAnchor` contract. The JSON stays off-chain (with the user and optionally with the relying party); only the hash and metadata are public. Revocation is a second on-chain event that references the original.

Grant flow, end to end:

```
// consentController.grant
async function grant(req, res) {
```

```
  const { verifierDid, scope, ttlSeconds } = req.body;
  const userDid = req.user.did;

  const consentDoc = {
    v: 1,
    user: userDid,
    verifier: verifierDid,
    scope,                                          // e.g. ['share:age>=21', 'share:insurance:active']
    issuedAt: new Date().toISOString(),
    expiresAt: new Date(Date.now() + ttlSeconds * 1000).toISOString(),
    nonce: cryptoRandomString(16),
  };

  // Client signs the doc with their DID key; we accept the signature here
  const userSignature = req.body.signature;
  assertValidDidSignature(consentDoc, userSignature, userDid);

  const consentHash = keccak256(canonicalJson(consentDoc));

  const tx = await consentAnchorContract.anchor(
    hashDid(userDid), hashDid(verifierDid), consentHash, ttlSeconds,
  );
  await tx.wait();

  await consentRepo.insert({
    consentHash, userDid, verifierDid, scope, ttlSeconds,
    txHash: tx.hash, status: 'active',
  });

  return res.json({ consentHash, txHash: tx.hash });
}
```

Verifier-side validation, when a relying party wants to confirm a consent was granted:

```
// consentService.verify
async function verify(consentHash, expectedUserDid, expectedVerifierDid) {
  const onChain = await consentAnchorContract.get(
    hashDid(expectedUserDid), hashDid(expectedVerifierDid), consentHash,
  );
  if (!onChain.exists) return { valid: false, reason: 'NOT_ANCHORED' };
  if (onChain.revokedAt > 0) return { valid: false, reason: 'REVOKED' };
  if (onChain.anchoredAt + onChain.ttl < nowSeconds()) {
    return { valid: false, reason: 'EXPIRED' };
  }
  return { valid: true, anchoredAt: onChain.anchoredAt };
}
```

Revocation, later:

```
// consentController.revoke
async function revoke(req, res) {
  const { consentHash } = req.body;
  const record = await consentRepo.getByHash(consentHash);
  if (record.userDid !== req.user.did) throw new Err('NOT_OWNER');

  const tx = await consentAnchorContract.revoke(
    hashDid(record.userDid), hashDid(record.verifierDid), consentHash,
  );
  await tx.wait();

  await consentRepo.markRevoked(consentHash, tx.hash);
  return res.json({ consentHash, revokedTxHash: tx.hash });
}
```

The grant-verify-revoke cycle gives three distinct consumers what they need. The user gets an immutable log of what they agreed to. The verifier gets a cryptographic receipt they can show to an auditor. The auditor (or regulator) gets a public, sample-able record of consent activity without being able to see the underlying consent payloads unless a party chooses to reveal them.

An important subtlety is what happens between grant and revocation. The on-chain record does not auto-expire: a verifier acting in good faith checks both the TTL and the revocation flag at every use, and treats a consent as valid only if both are green. A verifier that caches a consent decision and keeps relying on it past revocation is acting outside the protocol, and the on-chain record — which shows the revocation timestamp — gives the user evidence of that overreach. The contract does not enforce verifier behaviour; it enforces that verifier behaviour is audit-able.

Consent scopes are open strings by design, so that issuer and verifier ecosystems can evolve their vocabularies without contract upgrades. Common scopes today include `share:age>=21`, `share:insurance:active`, `process:ai:training`, `process:ai:inference`, `share:medical:summary`, and `share:identity:name`. The platform publishes a recommended vocabulary and checks against it in the wallet UI for human-readable confirmation, but the contract accepts any scope hash. Verifiers are expected to publish their required scopes; a mismatch between what the user granted and what the verifier uses is detectable by anyone comparing the two on-chain records against the signed consent document.

---

6. Smart contracts

Six contracts are deployed and verified on Flare Coston2 (chainId 114). All source is reproducible from the public repository (`github.com/CyberNinja7420/x-id-platform` under `contracts/src/`) and all bytecode is verified on Blockscout. This section lists each contract, its address, its on-chain role, the key ABI surface, and the security-critical properties that govern its behavior.

Common design discipline across all six contracts:

- **No admin keys** that can pause, drain, or unilaterally modify user state. The platform operator cannot freeze a user's credentials, move their stake, or censor their consent record.
- **No upgradeability without governance approval.** Contracts are not behind a proxy that the deployer can swap at will; any upgrade requires an on-chain `XIDGovernor` vote with quorum.

- **Event-first logging.** Every state change emits a structured event so off-chain indexers can reconstruct the complete state from chain history alone — no trusting the operator's database.
- **Deterministic storage layouts.** All storage variables are explicit, ordered, and documented, so future migrations are predictable.
- **Deployer + treasury:** EOA `0xd1190ea34051835d8743E057C1F7BF7838BE895A` performed initial deployment and holds governance tokens until DAO transition. Key is in cold storage; rotation to a fresh key is scheduled before mainnet.

Blockscout base URL: `https://coston2-explorer.flare.network/address/`

6.1 XIDToken

Address: `0x996605d357a5F71fB4da6FCF0C71457327dBE068` **Standard:** ERC-20 + ERC-20Votes (OpenZeppelin v5) **Supply:** 100,000,000 XID, minted once at deployment **Decimals:** 18

XIDToken is the governance and staking token. The ERC-20Votes extension records historical voting power at each block — essential for `XIDGovernor` proposals that need to know who held how much at the proposal's snapshot block, not at vote time.

Key ABI surface:

```
function mint(address to, uint256 amount);          // disabled post-deployment
function delegate(address delegatee);               // user self-delegates voting power
function getPastVotes(address account, uint256 timepoint) view returns (uint256);
function getPastTotalSupply(uint256 timepoint) view returns (uint256);
event  DelegateChanged(address, address, address);
event  DelegateVotesChanged(address, uint256, uint256);
```

Security properties:

- Mint function is guarded and has been irrevocably disabled after initial allocation. Supply cap is enforced on-chain; no future inflation.
- Delegation is self-sovereign — users choose their delegate, platform has no override.
- Transfer is standard ERC-20; no blacklist, no pause, no fee-on-transfer.

Roadmap note: a mainnet migration will deploy a fresh contract on Flare mainnet with the same invariants. Testnet tokens have no economic value and will not be bridged.

6.2 ConsentAnchor

Address: `0x4c384d28245012b6463340D02C28e19c5723058b` **Standard:** Custom minimal contract; ~400 lines of Solidity

ConsentAnchor is an append-only registry of consent events. Each entry records the cryptographic hash of a user-signed consent JSON document, together with indexing fields that make later proof-of-consent (or proof-of-revocation) cheap.

The actual consent payload — what data, under what terms, for how long — lives off-chain in the user-signed JSON. Only the hash goes on-chain. This minimizes gas costs while giving auditors, verifiers, and courts a tamper-proof timeline they can reconcile against the signed off-chain documents.

Key ABI surface:

```
function anchor(
    bytes32 userDidHash,
    bytes32 verifierDidHash,
    bytes32 scopeHash,
    bytes32 payloadHash,
    bool    isRevocation
) external returns (uint256 anchorId);

function getAnchor(uint256 anchorId) external view returns (AnchorEntry memory);
function countForUser(bytes32 userDidHash) external view returns (uint256);

event Anchored(
    uint256 indexed anchorId,
    bytes32 indexed userDidHash,
    bytes32 indexed verifierDidHash,
    bytes32 scopeHash,
    bytes32 payloadHash,
    bool isRevocation,
    uint256 timestamp
);
```

Security properties:

- **Append-only.** No function can delete or modify past entries. Revocations are new entries with `isRevocation=true` pointing at the scope being revoked; they do not overwrite the original.
- **No central admin.** Anyone can call `anchor()` — the contract does not verify DID ownership. Verification happens off-chain by anyone reading the record: does the signed JSON payload, which hashes to `payloadHash`, actually come from `userDidHash`'s keypair? If not, the entry is meaningless spam and ignored by downstream consumers.
- **Cheap:** one anchor costs ~50,000 gas (four `bytes32` indexed fields + one unindexed + timestamp). At current Flare gas prices, under a cent per entry.

6.3 PaymentContract

Address: `0x36aBfCF7C92A1766A57D467Ada66eDDC43adEEbf` **Standard:** Custom escrow + fee-router

PaymentContract mediates all paid flows between credential issuers, verifiers, and data consumers. It charges a **5% platform fee** (parameter adjustable by governance, not by admin) and holds funds in escrow until the underlying action — credential issuance, data release, verification — is confirmed complete by the paying party.

Key ABI surface:

```
function createEscrow(
    address payee,
    bytes32 actionHash,     // hash of the agreed scope + terms
    uint256 expiry
```

```
) external payable returns (uint256 escrowId);

function releaseEscrow(uint256 escrowId) external;   // payer confirms delivery
function refundEscrow(uint256 escrowId) external;    // after expiry, payer can pull back
function setPlatformFeeBps(uint16 newBps) external;  // onlyGovernance
function platformFeeBps() external view returns (uint16);

event EscrowCreated(uint256 indexed id, address payer, address payee, uint256 amount, bytes32 actionHash);
event EscrowReleased(uint256 indexed id, uint256 netPayee, uint256 platformFee);
event EscrowRefunded(uint256 indexed id);
```

Security properties:

- Platform fee is capped at 10% (1000 bps) in the code — governance cannot set it higher without a contract upgrade.
- Escrow release requires the payer's explicit transaction. The platform cannot drain escrow to itself or a third party.
- Refund path exists unconditionally after `expiry`; payee cannot hold funds hostage.
- Reentrancy-guarded using OpenZeppelin's `ReentrancyGuard`.
- Accepts native tokens (FLR on Flare). ERC-20 support is on the roadmap for USDT0 / USDC mainnet flows.

6.4 StakingContract

Address: `0xf8F6022A68F64A64D208A2E9F51a95bed7813Bad` **Standard:** Custom stake + slash

StakingContract governs stake locking for verifier nodes, validator participation, and issuer reputation bonds. Stake is a credible-commitment mechanism: a participant who misbehaves (issues fraudulent credentials, fails SLAs, double-signs) loses a slice of their staked tokens.

Parameters:

- **Minimum stake:** 1,000 XID (18 decimals → `1000 * 10^18`)
- **Slash rate:** 10% of active stake, per slash event
- **Unbonding period:** 14 days (prevents slash-then-run attacks)

Key ABI surface:

```
function stake(uint256 amount) external;
function requestUnstake(uint256 amount) external returns (uint256 unbondingId);
function withdraw(uint256 unbondingId) external;          // after unbonding period
function slash(address offender, uint256 amount, bytes32 reason) external; // onlyGovernance
function stakeOf(address user) external view returns (uint256);

event Staked(address indexed user, uint256 amount, uint256 newTotal);
event UnstakeRequested(address indexed user, uint256 indexed unbondingId, uint256 amount, uint256 unlockAt)
event Slashed(address indexed offender, uint256 amount, bytes32 reason);
```

Security properties:

- **Slash is governance-only** — no admin key. A slash event requires an on-chain `XIDGovernor` proposal to pass quorum.
- **Unbonding is immutable** — once requested, funds are locked for 14 days; the staker cannot accelerate or decelerate it.
- Slashed tokens go to the treasury (`XIDGovernor`'s recipient), not burned, so governance can redirect them to compensate victims of the slashable event.

6.5 XIDEscrow

Address: `0x8DC75b97f1246ad89a4E625677031443535Cf13d` **Standard:** Native-token dispute escrow

XIDEscrow is a simpler dispute-resolution escrow, distinct from `PaymentContract`. Where `PaymentContract` is for pay-per-action credential + data flows, `XIDEscrow` holds funds during pending disputes (e.g., a user claims a credential was issued in error and wants their verification fee refunded).

Lifecycle states: `Open → Disputed → Resolved | Refunded`

Key ABI surface:

```
function openEscrow(address counterparty, bytes32 disputeId) external payable returns (uint256 escrowId);
function flagDispute(uint256 escrowId) external;     // either party
function resolveEscrow(uint256 escrowId, address recipient) external;  // arbitration outcome
function refund(uint256 escrowId) external;          // mutually-agreed refund

event EscrowOpened(uint256 indexed id, address payer, address counterparty, uint256 amount, bytes32 dispute
event DisputeFlagged(uint256 indexed id, address flaggedBy);
event EscrowResolved(uint256 indexed id, address recipient, uint256 amount);
```

Security properties:

- Separate from PaymentContract so that governance arbitration for disputes doesn't need to touch the higher-volume payment path.
- Arbitration authority is delegated by governance to a `DisputeResolver` role (currently the deployer EOA; will move to a multisig pre-mainnet).
- Resolver cannot send funds to itself — resolution recipients must be one of the two parties in the dispute.

6.6 XIDGovernor

Address: `0x6a2A4744f7A03b596AdCb168c14549E735c9d5f4` **Standard:** OpenZeppelin Governor v5 (ERC-20Votes + no-timelock)

XIDGovernor is the on-chain DAO mechanism for platform governance. Proposals, voting, and execution all happen on-chain; voting power is snapshotted from `XIDToken` at proposal creation to prevent flash-loan governance attacks.

Parameters (set at deployment; modifiable by successor proposals):

- **Voting delay:** 1 block (effectively instant activation once a proposal is posted)
- **Voting period:** 1 week (~50,400 blocks at Flare's ~1.2s blocktime)

- **Proposal threshold:** 100,000 XID (0.1% of supply) — prevents spam proposals
- **Quorum:** 4% of supply (4,000,000 XID for + against combined, excluding abstain)
- **Timelock: None** by design — this is the "stakeholder-overridable" clause

No-timelock rationale: a timelock delays execution after a proposal passes (typical 48h). We opted against it because (a) the platform is in Beta and needs fast response to bugs and (b) the "stakeholder override" — detailed below — provides a functionally similar safety net without adding 48h of latency.

Stakeholder override: if a holder or group of holders controlling >10% of total stake calls the emergency veto function within 24h of proposal execution, the execution is nullified and must be re-submitted. This is a defense against narrow majorities pushing through surprising changes. Code reference: `XIDGovernor.vetoByStakers(...)`.

Key ABI surface:

```
function propose(
    address[] targets,
    uint256[] values,
    bytes[] calldatas,
    string description
) external returns (uint256 proposalId);

function castVote(uint256 proposalId, uint8 support) external returns (uint256);
function execute(
    address[] targets,
    uint256[] values,
    bytes[] calldatas,
    bytes32 descriptionHash
) external payable returns (uint256);

function vetoByStakers(uint256 proposalId) external;   // emergency brake
function state(uint256 proposalId) external view returns (ProposalState);

event ProposalCreated(uint256 proposalId, address proposer, ...);
event VoteCast(address indexed voter, uint256 proposalId, uint8 support, uint256 weight, string reason);
event ProposalExecuted(uint256 proposalId);
event StakerVetoed(uint256 proposalId, address[] vetoCoordinators);
```

Security properties:

- Proposal threshold + 1-block voting delay + 1-week voting period + 4% quorum make 51%-style attacks via flash loans infeasible (voting power snapshot is historical).
- Emergency stakeholder veto provides a 24h window to block execution of passing-but-surprising proposals.
- All actions a proposal can take are permissioned through `XIDGovernor`'s address on the target contract — e.g., `StakingContract.slash(...)` requires `msg.sender == XIDGovernor`. The platform operator's EOA has no direct admin path into the system.

6.7 Auxiliary: Groth16 verifier contracts

Alongside the six core contracts, the four Groth16 verifier contracts generated by `snarkjs export solidityverifier` are deployed for each ZK circuit. Their addresses rotate when circuits are recompiled; the live set is discoverable from `PaymentContract.ageProofVerifier()`, `...insuranceProofVerifier()`, etc., so client code never hard-codes verifier addresses. This isolation makes circuit upgrades atomic: governance approves a new verifier, the core contracts are repointed, and old proofs remain valid in the event history.

6.8 Audit status (honest disclosure)

The six contracts have undergone internal code review but **have not yet undergone a formal third-party audit**. A full audit by a reputable firm is a pre-mainnet gate. On mainnet, fresh deployments will replace these testnet addresses. Testnet is the correct place for breaking changes; production will not carry over any unaudited bytecode.

Bug reports and security advisories should go to `security@x-id.xyz`.

---

7. The X-ID token

XID is the platform's governance and staking token. It is **not** a fundraising instrument, is **not** offered for sale, and has no monetary value on the current testnet. This section describes what the token does on-chain today and flags the decisions that remain for mainnet.

What XID does today (Coston2 testnet):

- **Governance voting.** XID holders can propose and vote on changes via XIDGovernor, with voting weight proportional to held-and-delegated balance (ERC-20Votes).
- **Staking.** Operators of issuer and verifier services stake XID (minimum 1,000) to participate; bad behavior is provably slashable up to 10% per offense.
- **Payment integration.** PaymentContract routes credential and data payments and assesses a 5% platform fee, which accrues to the treasury.
- **Escrow collateral.** Dispute resolution via XIDEscrow uses the native chain token; XID staking provides the reputation weight used to select adjudicators.

What is deliberately unfinished:

- Final tokenomics — supply schedule beyond the initial 100M, inflation or deflation mechanics, treasury allocation, ecosystem fund — will be finalized before Flare mainnet launch, with community input through XIDGovernor.
- Testnet XID tokens are not convertible to, and bear no relationship to, any mainnet token that may be deployed.
- No public sale, airdrop, or allocation to external parties has occurred or is committed. Any future distribution will be published as a governance proposal and documented in this paper's successor version before it happens.

The design principle is that the token should be the minimum instrument needed to coordinate governance and align operators, not a speculative asset marketed ahead of utility. This paper will be updated when mainnet tokenomics are finalized.

8. Developer platform

X-ID is developer-first by construction: every feature in the product UI is a thin wrapper on the same public API that third parties can call.

8.1 REST API

The backend exposes **77 routes** across identity, credentials, proofs, consent, payments, medical records, organizations, and governance. The current inventory breaks down as:

- **54 LIVE** public routes (read-only or user-scoped)
- **19 AUTH-gated** routes (require valid SIWE session)
- **3 feature-gated** routes (behind mdoc and KYC-tier-2 flags, not yet activated)
- **1 labeled seeded-demo** route (explicitly marked in the docs, for sandbox use only)
- **0 broken** routes as of the 2026-04 audit

All routes are documented at `https://app.x-id.xyz/docs`. The API versioning policy is semver-prefixed (`/v1/...`) and breaking changes are announced via governance proposal at least one release cycle ahead of deprecation.

8.2 Auth pattern — SIWE

Integrations authenticate with **Sign-In With Ethereum**. The flow is:

1. Client requests a nonce from `GET /v1/auth/nonce`
2. User signs the SIWE message client-side with their DID-bound key
3. Client `POST`s the signed message to `/v1/auth/verify`
4. Backend returns a session token (JWT) that authenticates subsequent calls

No API keys, no OAuth round-trips with a third-party IdP, no password database. The user's own key is the credential.

8.3 Verification pattern — OID4VP

A relying party that needs to check a credential implements OID4VP:

1. Relying party publishes a presentation request (what claims, what issuers are acceptable, what proof formats)
2. User's wallet receives the request, constructs a presentation with selective disclosure where appropriate
3. Wallet signs and returns the presentation to the relying party
4. Relying party verifies the presentation signature and credential signatures (locally, using the issuer's resolvable DID document)

The X-ID SDK wraps this flow in a handful of calls. For ZK-gated claims, the SDK composes the presentation with a Groth16 proof and submits both to the relying party, which verifies off-chain or on-chain depending on policy.

8.4 SDKs and integration

The primary SDK is TypeScript/JavaScript, shipped as an npm package, usable from both Node and browser contexts. It wraps SIWE, OID4VCI, OID4VP, ZK proof composition, and wallet portfolio queries against

the Routescan-backed multichain view. Additional language SDKs (Python, Go) are on the roadmap; the REST API is the canonical interface until those ship.

8.5 Multi-chain portfolio

The platform integrates with **Routescan** to provide a unified portfolio view across 39+ EVM chains. For a user whose identity is anchored on Flare but whose assets or reputation are scattered across Ethereum, Polygon, Arbitrum, Base, and others, the wallet surface presents all of it under one DID.

---

9. Use cases

9.1 Individuals

Single sign-in with a cryptographic key the user controls. A wallet that holds credentials — from employers, from schools, from governments, from healthcare providers — under the user's custody. Selective disclosure: prove you are over 18 without handing over a driver's license; prove you are insured without handing over the policy.

Portable reputation across services: a review credential issued by one marketplace can travel with the user to another, with the original issuer's signature intact. Web3 email bound to the user's DID, not to a provider. User-owned static sites hosted on IPFS with cryptographic change-history.

9.2 Creators & professionals

Creators get verifiable provenance for their work. A photograph, a document, a piece of code can be signed by the creator's DID and time-anchored on-chain, creating a cryptographic timestamp that pre-dates any later contested claim.

Professional credentials — a license, a certification, a professional body membership — are issued once and presented many times, with revocation checks happening at presentation time. A freelancer can prove eligibility for a contract without giving the client a scanned copy of a certificate.

9.3 Organizations (employment, membership, access credentials)

Organizations issue three broad classes of credentials: **employment** (you work here, in this role, from this date), **membership** (you belong to this society / gym / club), and **access** (you may enter this building / access this system / attend this event).

All three benefit from the same pattern: the organization issues a JWT-VC signed by its DID; the employee, member, or visitor stores it; a verifier at the door (or at the firewall, or at the login page) checks it via OID4VP. The organization never has to run a traditional IAM server against external verifiers.

Organizations also use the ConsentAnchor contract to log formal consent transactions — data-processing agreements, photography releases, NDA acknowledgments — where the consent event itself is the product.

9.4 Healthcare (FHIR-native)

Healthcare is a design partner for X-ID, not an afterthought. Medical records live in the HAPI FHIR sidecar in standard FHIR resources. Credentials derived from those records (immunization records, prescription eligibility, lab completion) are issued as JWT-VC with provenance back to the FHIR resource.

The ZK circuits shine here. A pharmacy asks: have you had N doses? The `vaccinationProof` returns yes/no with a proof. A triage desk asks: are you over 18? The `ageProof` returns yes/no. A specialist asks: do the records *not* show condition X? The `conditionProof` returns yes/no — a negative disclosure that is uniquely hard with traditional credential schemes.

FHIR compatibility means clinical interoperability is a first-class concern: any EHR that speaks FHIR can exchange resources with X-ID's sidecar, preserving coding systems (SNOMED, LOINC, ICD-10) rather than flattening them into a lossy intermediate.

9.5 AI-era data rights

Large-model trainers increasingly need verifiable evidence of training-data consent. X-ID provides the substrate: a user can publish, on the ConsentAnchor contract, a scoped consent or refusal for a given dataset and a given purpose. A trainer building a dataset can require a valid consent record for each contributor and log the consent hashes alongside the training run.

Creators get the inverse: a verifiable refusal. A creator can publish a machine-readable "no training" statement bound to a DID that also signs the work, so that robots-txt-style honor systems get cryptographic teeth. This does not prevent misuse — no identity system can — but it makes misuse provable and actionable.

9.6 Developers

Developers get an identity layer they can build on without having to re-invent it. The REST API, SIWE auth, OID4VP verification SDK, and multi-chain portfolio view compose into a surprisingly small integration: a typical app can replace email/password, session management, and KYC flows with X-ID in under a day of work, according to internal integration tests.

The platform is open: every feature in the X-ID UI is a call to a route that external applications can call. There are no preferred-partner shortcuts.

---

10. Roadmap

The roadmap is phased honestly. "Now" is what is running today on Coston2. "Beta" is what is running for early-access users this quarter. "Next" is committed, scoped, and on the engineering schedule. "Future" is directional: real work, but dates and specifics will harden as dependencies resolve.

Now (Coston2 testnet, live)

- Six smart contracts deployed and verified on Blockscout
- Four Groth16 ZK circuits operational with `pot14` trusted setup
- SIWE authentication plus passkey and TOTP MFA
- FHIR-grade medical records via HAPI sidecar
- Kubo IPFS daemon (v0.40.1) for content addressing
- Multi-chain portfolio across 39+ EVM chains via Routescan
- Full public REST API (77 routes) with docs
- OID4VCI issuance and OID4VP verification flows
- Web3 email (mailchain@0.30) bound to DID
- On-chain governance via XIDGovernor
- Issuer and verifier staking via StakingContract

Beta (Early Access Beta, current)

- Free beta access at `app.x-id.xyz`
- Up to 100 Early Access beta users invited to onboard
- Open API docs for integrators
- Active developer hiring (backend, frontend, protocol, security)
- Public weekly changelog and governance proposal queue

Next

- **Flare mainnet launch** (chainId 14): redeploy the six-contract suite with finalized tokenomics, migrate verified beta data, publish audit report
- **Mobile credential wallet** (React Native / Expo): native iOS and Android apps for holding, presenting, and scanning credentials, with biometric-bound keys
- **Organization admin console**: bulk issuance, revocation, usage analytics, SCIM-style user management for HR-connected flows
- **KYC tier-2 integrations**: Jumio and Sumsub integration for regulated relying parties that need identity-document verification in addition to DID proof
- **Fiat on-ramp**: licensed partners for users who need to convert fiat to the chain's native token without leaving the app

Future

- **ISO 18013-5 mobile driver license support** (mdoc): the code path is already feature-gated in the API; activation requires COSE/X.509 implementation and jurisdiction-specific issuer integration
- **Cross-chain DID resolution**: CCIP and LayerZero-based resolvers so a DID anchored on Flare can be looked up cheaply from any connected chain
- **Enterprise tier**: dedicated-instance deployments, SLA-backed uptime, SOC 2 or equivalent audit, private-data-processor contracts
- **Advanced ZK**: migration from Groth16 to Plonk (or PlonKY2 for recursion-friendly proofs) to reduce trusted-setup sensitivity and support more complex circuits

Every roadmap item has a champion and a tracking issue on the public mirror. The roadmap is a working document, not a marketing artifact.

---

11. Security & privacy posture

11.1 Self-custody

Private keys are generated and held on the user's device. The backend never sees them. A user who loses their device (without a passkey on another device, without a backup, without social recovery configured) loses access — but a user whose provider gets breached does not lose access. The trade-off is explicit.

11.2 Selective disclosure

The ZK circuits mean that providers can learn the minimum fact they need without learning the underlying data. Over-sharing is not a UX-friendliness problem the user has to navigate; it is an architectural default.

11.3 Consent ledger

The ConsentAnchor contract provides an immutable public record of consent events (by hash). Relying parties, regulators, and users themselves can reconstruct, months or years later, exactly what was agreed to and when.

11.4 Testnet status is explicit

Every page that touches on value, transfer, or contract interaction labels the environment. Testnet tokens carry no value, and the documentation, product UI, and API responses all advertise the network. There is no "testnet mode" that a user could miss.

11.5 Threat model and mitigations

A security posture that is not named cannot be evaluated. This section enumerates the adversaries we explicitly design against, their realistic capabilities, and the primary mitigations X-ID has in place today. This is not an exhaustive threat model — a pre-mainnet audit will go deeper — but it is the model the team uses to reason about architectural decisions.

External attacker with network access. An attacker who can intercept traffic between a user and `app.x-id.xyz`, probe public API endpoints, or replay captured requests. Capabilities include TLS downgrade attempts, SIWE message replay, nonce reuse, and high-volume probing for authentication or credential endpoints. *Mitigations:* TLS everywhere with HSTS, SIWE nonces are single-use and expire after 5 minutes (checked against a Redis-backed one-time-use store), Express rate limits per-IP and per-session for auth-sensitive endpoints, strict CORS (only the app origins are accepted), passkey binding on second factor so an intercepted SIWE signature is not sufficient on its own, and JWT rotation on sensitive actions.

Compromised credential issuer. An issuer — an employer, a university, a clinic — is breached or coerced and begins issuing fraudulent credentials under its legitimate DID. Capabilities include issuing VCs that look valid, backdating credentials, and refusing to honour revocation. *Mitigations:* issuer DIDs are published on-chain with discoverable public keys, so verifiers can pin the expected key and detect key rotation; every issuer credential-type is scoped (an HR issuer cannot mint medical credentials); the user retains a revocation key for their own side of the relationship and can sever acceptance from that issuer unilaterally; status list credentials let the issuer publish revocation and let verifiers check it at presentation time without round-trip to the issuer; and issuer reputation feeds the staking layer, so staked issuers have skin in the game. No single compromised issuer can unilaterally forge X-ID platform credentials because X-ID's own issuer DID signs only its own credentials and countersigns no one.

Compromised verifier. A verifier tries to over-collect — asking for the full credential when a ZK proof would suffice, storing responses beyond their scope, or re-sharing disclosed data downstream. *Mitigations:* consent prompts in the wallet show the exact fields requested before any disclosure; selective disclosure composes only the fields asked for; ZK proofs return booleans, not underlying data, so even a fully cooperating user cannot over-share beyond the circuit's output; consent anchors create an on-chain audit log so a later over-collection can be proven against the recorded scope; and presentation responses are scoped to the verifier DID named in the request, so replaying one verifier's presentation at another is rejected by audience checks.

Malicious user trying to forge a credential. A user attempts to mint or modify a credential in their own favour — claiming a degree they do not have, an insurance policy they never bought, an age they are not. *Mitigations:* credentials are valid only if the JWT-VC is signed by a DID whose public key the verifier trusts. The user's own key signs presentations, not issuances. Issuer keys are on-chain and public, so "fake issuer" is detectable. ZK proofs operate over inputs that are themselves signed credentials; a user cannot

feed an unsigned DOB into `ageProof` and have it accepted, because the circuit's public inputs include the issuer-signed hash of the DOB credential. Forgery reduces to breaking ECDSA on the issuer's key, which is the assumption the whole DID stack rests on.

Platform operator (us) going rogue. What can X-ID itself do, and what can it not do? *Honest answer:* We can serve a malicious frontend to users (we sign our own JS), we can take `app.x-id.xyz` offline, we can delete rows from our own database, and we can refuse to publish new consent anchors. *What we cannot do:* issue credentials on behalf of users (we do not hold their keys), countersign third-party credentials (issuers sign their own), decrypt client-side encrypted data (we do not have the key), or retroactively alter on-chain consent anchors (the chain is the source of truth). A rollback of our database is detectable because on-chain hashes do not match. A frontend compromise is mitigated by subresource integrity on third-party scripts, reproducible builds, and the option for integrators to host their own wallet. If X-ID disappears, user identities and credentials continue to work against any compliant DID resolver.

Rogue Ethereum node or RPC. A compromised RPC provider returns false chain state, hides transactions, or claims transactions are finalised when they are not. *Mitigations:* X-ID reads critical state from more than one source — Flare's own FDC endpoints, Flare public RPC, and independent providers — and requires consensus for writes that matter (consent anchors, payments, governance). On-chain reads are pinned to block hashes on sensitive queries so a subsequent inconsistency between providers is detectable. For the strongest guarantees, verifiers can run their own Flare node and pass its endpoint to the SDK; the platform does not force reliance on our RPC.

Flare network-level attack. A 51% attack, a long reorg, or a coordinated consensus failure on Flare Coston2 or eventually mainnet. *Mitigations:* the chain is an audit log, not the source of truth for identity. User DIDs exist regardless of chain state; credentials are signed off-chain and are valid even if the anchor chain is down; consent anchors become inconclusive during a reorg but revert to a known-good state when consensus re-establishes. The worst-case scenario is that consent anchor history during the attack window is disputed, not that identities are invalidated. Long-term mitigation is multi-chain anchoring (on the Future roadmap), so a consent record can be replicated across independent chains.

Physical device compromise. An attacker steals a user's laptop or phone with an active X-ID session. *Mitigations:* passkey gating on sensitive actions (even if a session cookie is captured, rotating a key or issuing a new credential requires re-auth via WebAuthn), biometric binding on passkey devices where available, TOTP as a fallback second factor, revocation from a backup device (the user can mark the stolen device's session family invalid and rotate their SIWE-bound key), and the on-chain revocation registry so credentials the stolen device holds can be marked revoked on the chain where any verifier will see the revocation at presentation time. The user's data at rest on the device is protected by whatever the device OS provides (Secure Enclave, TPM); X-ID does not try to reinvent that.

11.6 Server-side hygiene

All secrets live in 1Password, not in the repository. Git history has been scrubbed of prior historical leaks (a legacy GitHub PAT was revoked and purged via `filter-repo` earlier in April 2026). Dependabot stands at zero open vulnerabilities as of 2026-04-21. Backups are encrypted in transit and at rest. Database column encryption covers sensitive columns.

11.7 Transparency

All six platform contracts are verified on Blockscout and can be audited directly against the source repository. The API route inventory is public. The ZK circuit definitions are public. The roadmap is public. The known

limitations — including the list in Section 11.8 — are public.

11.8 What is NOT yet in place

Honesty requires naming the gaps:

- **Formal third-party security audit** — planned pre-mainnet, not yet complete
- **KYC tier-2** — Jumio and Sumsub integrations not yet contracted
- **Mobile app** — prototype only; production app on the Next-phase roadmap
- **Cross-chain bridge** — planned on the Future-phase roadmap
- **ISO 18013-5 mdoc** — feature-gated in code; COSE/X.509 implementation still required

Until these are complete, X-ID remains explicitly a beta platform. Users should treat it as such.

---

12. Governance

Governance is on-chain via **XIDGovernor** (`0x6a2A4744f7A03b596AdCb168c14549E735c9d5f4`), an OpenZeppelin `Governor` instance with ERC-20Votes accounting delegated to XIDToken.

Proposals can be created by any XID holder who meets the minimum proposal threshold (set by prior governance vote). Voting is standard Governor: for / against / abstain, quorum by voting supply at snapshot, simple majority for passage unless the proposal category specifies otherwise.

12.1 No timelock (by design)

Unlike most OpenZeppelin deployments, XIDGovernor does not instantiate a TimelockController. The rationale is that during beta the platform needs fast iteration: a governance round of 48-72 hours already provides meaningful deliberation, and a post-execution timelock would delay legitimate bug-fix governance by another window.

12.2 Stakeholder override

As a safety net during beta, a supermajority of staked XID holders (via StakingContract) can veto a passed proposal within a defined override window. The override is bounded: it can block execution of an approved proposal, but cannot execute a proposal of its own. This is asymmetric on purpose — the override protects against captured governance, not against unpopular outcomes.

12.3 Path to mainnet governance

Before mainnet deployment, the governance design will be revisited. Likely changes include (a) adding a TimelockController for a subset of proposal categories, (b) formalizing the override window and thresholds in code, and (c) a dedicated emergency-fix path with narrower scope and faster execution. These will themselves be governance proposals under the current XIDGovernor.

Governance transparency: every proposal, vote, and execution is on-chain and queryable via Blockscout. The on-chain record is canonical. An off-chain deliberation forum is on the roadmap.

---

13. Open source & contribution

The X-ID monorepo is mirrored publicly:

- **GitLab (primary)**: `https://gitlab-01.evdps.com/Devadmin/x-id-platform`
- **GitHub (mirror)**: `https://github.com/CyberNinja7420/x-id-platform`

The GitLab instance runs the primary CI pipeline; GitHub receives mirrored pushes for community visibility and issue tracking. The repository is a monorepo containing the backend, frontend, contracts, circuits, SDK, and documentation, organized for independent deployability per service.

13.1 License

An open-source license has been selected as a planned milestone and will be in place before Flare mainnet launch. The working default is a permissive OSI-approved license (MIT or Apache-2.0) for SDK and contract code, with a separate decision pending for the backend application code where copyleft protection may be desirable. The final selection will be published as a governance proposal.

13.2 How to contribute

Bug reports and feature requests go to the GitHub mirror's Issues tab. Pull requests go to the GitLab mirror to enter CI; mirroring back to GitHub happens automatically. Contribution guidelines — coding standards, commit conventions, security-review gating for contract and cryptographic code — are in `CONTRIBUTING.md` at the repository root.

Security-sensitive reports should go to `security@x-id.xyz` (a PGP key and published security.txt will accompany the mainnet audit). A responsible-disclosure policy with a bounty structure will be launched alongside the mainnet audit.

13.3 Careers

X-ID is actively hiring. Open roles include backend engineering (Node.js, Postgres, cryptography-adjacent), frontend engineering (React, TanStack, design-system work), protocol engineering (Solidity, circom, ZK), and security. Applications to `careers@x-id.xyz`.

---

14. Risk disclosures

14.1 Testnet risk

The platform runs on Flare Coston2. Testnet chains can be reset, reorganized, or deprecated by their operators. Any state — identities, credentials, consent anchors — created during the beta may need to be migrated, re-issued, or re-anchored at mainnet launch. The team will provide a documented migration path; users should not treat beta data as permanent.

14.2 Pre-audit smart-contract risk

The six platform contracts are written defensively and reviewed internally, but have not yet been audited by an independent third party. Users should not deposit funds they cannot afford to lose into any contract before the audit is complete and its report is published. A pre-mainnet audit is on the engineering schedule.

14.3 Cryptographic risk

The Groth16 trusted setup used a `pot14` ceremony. As long as at least one ceremony participant was honest and discarded their toxic waste, the setup is secure; the risk is real but well-understood. A migration to setup-free schemes (Plonk, PlonKY2) is on the Future roadmap.

Keys are held by users. Key loss means account loss unless recovery was configured. The team recommends setting up at least two passkey-capable devices and a hardware security key at onboarding.

14.4 Regulatory risk

Identity and health-data regulation is a patchwork. Relying parties are responsible for their own compliance posture in their own jurisdictions. X-ID provides primitives — DIDs, VCs, ZK proofs, consent anchors — and some opinions about how to use them, but the platform is not a substitute for jurisdiction-specific legal advice. Users in jurisdictions that restrict or regulate public blockchain interaction should understand their local rules before depositing any information on-chain, including hashes.

14.5 Dependency risk

The stack depends on upstream projects: Flare Network, Postgres, Redis, Kubo IPFS, HAPI FHIR, snarkjs, did-jwt, ethr-did-resolver, OpenZeppelin Governor, the Node.js and Next.js ecosystems. Each of those has its own failure modes. The team tracks upstream security advisories and patches promptly; Dependabot stands at zero open alerts as of 2026-04-21.

14.6 Operational risk

The platform is a six-container Docker stack behind a single reverse proxy. While the design is stateless at the API layer and data is backed up daily, the current deployment does not include the high-availability failover that enterprise deployments will require. Enterprise-grade deployment is a roadmap item.

---

15. Glossary

- **DID** — Decentralized Identifier. A W3C-specified identifier that resolves to a DID Document describing public keys and service endpoints, without relying on a central issuer.
- **VC** — Verifiable Credential. A cryptographically signed statement by an issuer about a subject, per the W3C VC Data Model 2.0.
- **ZK (Zero-Knowledge Proof)** — A proof that a statement is true without revealing the underlying data the statement is about.
- **SIWE** — Sign-In With Ethereum, specified in EIP-4361; authentication by signing a standard message with an Ethereum-compatible key.
- **SSI** — Self-Sovereign Identity. The design principle that users should own and control their identifiers and credentials, rather than renting them from service providers.
- **FHIR** — Fast Healthcare Interoperability Resources. An HL7 standard for clinical data exchange, used by X-ID's medical record sidecar.
- **OID4VCI** — OpenID for Verifiable Credential Issuance. A specification for how a wallet requests and receives credentials from an issuer.
- **OID4VP** — OpenID for Verifiable Presentations. A specification for how a verifier requests and receives a credential presentation from a wallet.

- **Groth16** — A specific ZK-SNARK proving system; small proofs, fast verification, requires a trusted setup.
- **Passkey / WebAuthn** — A FIDO standard for phishing-resistant authentication using hardware-bound keys.
- **SNARK** — Succinct Non-interactive Argument of Knowledge; the family of cryptographic proof systems that includes Groth16 and Plonk.
- **FTSO** — Flare Time Series Oracle; Flare Network's native decentralized price oracle.
- **FDC** — Flare Data Connector; Flare's bridge for bringing verifiable external data on-chain.
- **mdoc / ISO 18013-5** — The ISO standard for mobile driver licenses and analogous identity documents.

---

16. Links & Contact

Platform

- App: `https://app.x-id.xyz`
- API docs: `https://app.x-id.xyz/docs`

Source

- GitLab (primary): `https://gitlab-01.evdps.com/Devadmin/x-id-platform`
- GitHub (mirror): `https://github.com/CyberNinja7420/x-id-platform`

On-chain (Coston2, chainId 114)

- Blockscout: `https://coston2-explorer.flare.network`
- Issuer DID: `did:ethr:114:0xC03BbD4C6349B34b3a14afB657DAE869ed5f9BAB`
- Treasury / deployer EOA: `0xd1190ea34051835d8743E057C1F7BF7838BE895A`

Contact

- General: `hello@x-id.xyz`
- Security reports: `security@x-id.xyz`
- Careers: `careers@x-id.xyz`

This document is Version 2.0, dated April 2026, covering the Early Access Beta on Flare Coston2 testnet. A Version 3.0 will be published at Flare mainnet launch with finalized tokenomics, the post-audit security report, and the license selection. Until then, treat this document as a faithful description of a beta platform — not a prospectus, not a promise.