WIGGWIGG

Zero-knowledge identity & communications platform

WIGGWIGG Security Whitepaper

Version: 1.0 (draft) · **Date:** 2026-06

WIGGWIGG is a privacy-first phone-number and identity product. This document describes, in technical detail, how we protect your data: what we *cannot* read, what we *can* read and why, the cryptography we use, and where our guarantees end. It is written for a skeptical, technical reader.

Our cryptographic core is open source at github.com/WiggWiggDev/crypto (the `@wiggwigg/crypto` package): the key-derivation, sealing, and signing primitives this document describes are auditable independently of the closed application.

1. Summary and stance

- **Zero-knowledge by design.** Your password, vault entries, identities, contacts, and message *content* are encrypted on your device with keys we never hold. We cannot read them, and we cannot be compelled to produce a key we don't have.
- **Open-core, not "open source."** We open-sourced the cryptographic core so you can audit *how* keys and data are protected. The rest of the application (the operational and abuse-defense surfaces) is closed. We never imply otherwise.
- **The edges.** Phone numbers, emergency (E911) addresses, and routing metadata are server-readable because phone service requires it. We are not end-to-end-encrypted in transit (the phone network predates that). We do not yet have forward secrecy or post-quantum protection. Those limits are stated in sections 10 and 11.
- **Self-attested today, externally audited later.** This document is backed by open source code and a published vulnerability-disclosure policy. A third-party cryptographic and application audit is **planned**; we will publish it when complete and will not claim it before then.

2. Threat model and trust boundaries

We design against an **honest-but-curious server**: assume our infrastructure may be compromised, subpoenaed, or operated by someone who should not see your data. Under that assumption, your zero-knowledge data stays unreadable because the keys live on your device.

Boundary	What we assume	What it can see
Your device	Trusted; client-side crypto runs here	Everything you store (it holds your keys)
Your password / recovery phrase	Secret, known only to you	Nothing leaves the device
WIGGWIGG servers	Honest-but-curious, may be compromised	Server-encrypted operational data (section 4); never ZK data
Database	Protected by row-level security and triggers	Ciphertext only, for ZK data
Telecom carrier (Telnyx)	A "dumb pipe" that learns only what routing requires	Call/SMS metadata and content **in transit** (like all phone service)
Payment processors (Stripe, Blockonomics)	Data-minimized	Card-billing essentials, or a Bitcoin address and amount; never your vault or identities

We do **not** defend against a compromised endpoint (malware on your own device, or a malicious browser extension), nor against metadata inherent to placing a phone call or SMS through the public network. See section 10.

3. Cryptographic architecture

All primitives below come from audited libraries (`@noble/curves`, `@noble/hashes`, `@scure/bip39`). We do not roll our own.

- **Key derivation:** your password is stretched with **Argon2id** (128 MiB memory, 3 iterations). It is memory-hard and resistant to GPU/ASIC brute-force. The same parameters are byte-for-byte identical across web and mobile so your keys match on every device.
- **Vault-key indirection:** the password-derived *master key* wraps a separate, per-account *vault key*, and your data is encrypted under the vault key. Changing your password re-wraps the vault key only; it never re-encrypts your data.
- **Symmetric encryption: AES-256-GCM** (authenticated) for vault entries, identities, contacts, message content, and server-side operational fields.
- **Public-key sealing:** inbound message content is encrypted with AES-256-GCM under a one-time key, and that key is sealed to your device's **X25519** public key via X25519-ECDH, HKDF-SHA256, and XChaCha20-Poly1305 (a libsodium-style sealed box). We generate the sender-side keypair, use it once, and discard it, so we retain **no key** that can read your messages (see section 5).
- **Authentication:** proving you know your password without revealing it (section 6).
- **Key separation:** every derived key (auth, vault, communications, recovery) comes from the master key via **domain-separated HKDF**, so compromise of one never yields another.

- **Crypto agility:** every scheme carries a version, so we can rotate a primitive without breaking stored data. This is how the post-quantum upgrade in section 11 lands without a rewrite.

4. Data classification

What is and isn't zero-knowledge:

Tier	Data	Who can read it
Zero-knowledge	Identities, vault entries, contacts, SMS/MMS/voicemail **content** (by default), anti-phishing preferences	Only your device
Server-encrypted	Phone numbers (routing), **E911 addresses** (emergency dispatch), SIP credentials, billing metadata, call/SMS cost and carrier metadata	Server can decrypt (required to operate)
Blind-indexed	Searchable equality lookups (e.g. by phone) without storing plaintext	Server matches a keyed HMAC, not the value
Plaintext	Account ID (a public username), timestamps, primary and foreign keys, `is_admin`	Anyone with DB access (non-sensitive)

The carve-outs: phone numbers and E911 addresses are *server-readable* because the network must route your calls and emergency services must locate you. Voicemail content is zero-knowledge **unless** you turn on "Listen by phone," which keeps a server-readable copy so you can hear voicemail by calling in. These are deliberate trade-offs for a working phone service, not gaps in zero knowledge.

Server-encrypted fields are bound to their context (for example, an E911 address is cryptographically tied to its owner) so a ciphertext cannot be moved between records. Extending that binding to every server field, plus per-user key derivation, is on our published roadmap.

5. Communications: "server-blind," not forward-secret

Each inbound SMS, MMS, or voicemail is sealed to your X25519 public key, and we discard our sender-side keypair the instant the message is stored. The result: **we hold no key that can read your messages: not at rest, not later, not under a warrant served on us.** We can be compelled to produce the encrypted bytes; we cannot be compelled to produce a key we do not have.

What this is *not*: this is **not forward secrecy.** Your messages are sealed to your device's long-term key, which is the same key your device uses to re-read your history on a new device. If *your* private key were compromised, your stored history could be decrypted. We optimize for **server-blindness** (a server or warrant cannot read your content), which is the achievable and meaningful property for stored, re-readable message history. True forward secrecy conflicts with multi-device history and is not something we claim. Post-quantum hardening of this seal is on our roadmap (section 11).

Carrier-network transit (SS7/SIP) is visible to carriers like all phone service. For end-to-end encryption *in transit*, use Signal or WhatsApp with your **WIGGWIGG** number.

6. Authentication

Your password never leaves your device. Today, your device derives the master key (Argon2id) and computes an **HMAC-SHA256 proof** of it. The server stores only a hash of that proof and compares it on login. The server never sees your password, and an Argon2id-gated master key makes the proof costly to forge.

We are migrating login to an **Ed25519 challenge-response** scheme in which the server stores only a public key and you prove possession by signing a single-use, 60-second server challenge, so a database leak of the verifier cannot be replayed to log in. This is built and rolling out. We will not claim the stronger "a database leak can't authenticate" property until the legacy verifier is fully retired.

MFA: WebAuthn/passkeys (phishing-resistant) and TOTP. **Anti-phishing:** before you type your full password, the site proves itself with a per-account marker (phrase, colors, avatar, optional audio) generated server-side from a seed, and a partial-password check (first characters, HMAC-SHA256) gates the reveal.

7. Account recovery

Recovery uses a 24-word **BIP39** phrase (256 bits of entropy) that you hold and we never see. From it your device derives a recovery key. We store only a hash of the recovery public key and a copy of your master key wrapped to that recovery key. Recovery proves possession via challenge-response. If you lose **both** your password and your recovery phrase, your zero-knowledge data is unrecoverable by design: we cannot decrypt it for you. An admin-assisted reset can restore *access* to your account (phone numbers, billing) but only with your explicit, separate confirmation, and it does not recover the ZK data.

8. Key management and rotation

Server-side keys (the operational field key, the blind-index key, the per-user metadata seed) live in a managed secret store as a **versioned ring**, validated on startup. Each scheme (KDF, vault-key, asymmetric seal, server key) carries a version column, enabling zero-downtime rotation: new writes use the current version, and readers dispatch on the stored version. Crypto-bearing columns are protected by database triggers that refuse to silently null them.

9. Application and infrastructure security

- **Row-level security** is enforced on tenant tables (with `FORCE ROW LEVEL SECURITY`, a required-context function, and a CI guard) so one user can never read another's rows.
- **Transport:** TLS everywhere; security headers (CSP/HSTS); CSRF defended via SameSite cookies and strict origin allowlisting.
- **Network:** per-VPC isolation and per-VPC NAT as deliberate defense-in-depth.
- **Encryption at rest:** the database, cache, and object storage are encrypted with managed keys (AWS KMS), underneath the application-layer and zero-knowledge encryption above.
- **Defense in depth:** a managed web application firewall fronts the public edge, with continuous threat detection, control-plane audit logging, VPC flow logs, and encrypted, versioned backups.
- **Observability is anonymous by default:** no IP or user-agent in admin audit rows, and user identity in telemetry only on explicit opt-in. Operational metadata retention is 90 days.

- **Provider data-minimization:** the carrier learns only what routing requires; the card processor (Stripe) receives opaque identifiers and generic line items, and the Bitcoin processor (Blockonomics) sees only a payment address and amount. We never query them about you.

10. Limitations and non-goals

We state these plainly:

- **Endpoint compromise.** Malware or a malicious extension on *your* device can read your data after you unlock it. Client-side encryption cannot defend a compromised client.
- **No forward secrecy** on stored messages (section 5).
- **No post-quantum protection** yet (section 11): a future quantum adversary could decrypt today's X25519-sealed data ("harvest now, decrypt later").
- **Phone-service metadata** (who called whom, when, for how long) is inherent to the public network and visible to carriers. We minimize and retain it for 90 days, but we cannot make it private.
- **Server-readable operational data** (phone numbers, E911, billing) is not zero-knowledge (section 4), because it is required to operate.
- **Authentication is mid-migration** (section 6), so the strongest claim awaits the legacy verifier's retirement.

11. Roadmap

- **Post-quantum (the headline):** hybrid **X25519 + ML-KEM-768** key wrapping for the message seal and recovery wrap, so an attacker must break *both*. This directly addresses harvest-now-decrypt-later for long-term stored data.
- **Auth:** complete the Ed25519 challenge-response rollout and retire the legacy verifier, then evaluate an asymmetric PAKE (OPAQUE).
- **Retire RSA-2048** (admin support encryption) in favor of the X25519 sealed box.
- **Broaden context-binding** (AAD) and per-user key derivation across all server-encrypted fields.
- **Third-party audit:** a planned external cryptographic and application review.

12. Verification and disclosure

- **Audit the crypto:** github.com/WiggWiggDev/crypto. The KDF, seal, signing, and recovery primitives are exactly what this document describes. Open source proves the *design* and the *client-side* primitives; it does not, by itself, prove our servers run them unmodified, so reproducible client builds and published bundle hashes are on the roadmap.
- **Report a vulnerability:** see https://wiggwigg.ca/.well-known/security.txt. We offer a good-faith safe-harbor for responsible disclosure and respond promptly.

This document is versioned and dated. If reality and this document ever disagree, that is a bug in one of them. Tell us.

WIGGWIGG

Made in Quebec, Canada · security@wiggwigg.ca