Falcon Ledger Wallet Security Assessment

qXRP Falcon Ledger Wallet - Send/Receive & Backup/Restore Subsystems

Independent Security Review

2026-07-01

Contents

Classification: Informational - testnet application

Target: `beartec-jpg/Falcon-faucet-wallet` (qXRP Web Portal)

Network: Falcon Ledger testnet, Network ID `1001`

---

1 Third-Party Security Assessment

1.1 qXRP Falcon Ledger Wallet - Send/Receive & Backup/Restore Subsystems

Report type	Independent third-party security review

Target	`beartec-jpg/Falcon-faucet-wallet` (qXRP Web Portal)
Scope	Wallet **send / receive** and **backup / restore** flows
Network	Falcon Ledger **testnet**, Network ID `1001` (Falcon-512 post-quantum accounts)
Assessment date	2026-07-01
Methodology	Manual source code review (white-box), architecture & threat-model analysis, cryptographic design review
Classification	Informational - testnet application

Disclaimer. This report reflects a point-in-time review of the source code at the stated commit. It is not a penetration test and does not constitute a guarantee that the application is free of vulnerabilities. Findings are prioritized for a **testnet** deployment; several accepted risks would need to be re-rated **High/Critical** before any mainnet or real-value use.

1.2 1. Executive Summary

The wallet implements a **non-custodial, client-side** signing model. Falcon-512 key pairs are generated in the browser via WebAssembly (liboqs), the private key material (`falcon_secret`) is encrypted at rest with key material bound to a WebAuthn passkey, and transactions are signed locally so the secret never leaves the device. Backups are exported as passphrase-encrypted JSON files using AES-256-GCM with PBKDF2. The server-side surface is deliberately thin: it only relays account reads and pre-signed transaction blobs to a public RPC node.

Overall the design is **sound and security-conscious for a testnet product**. The codebase shows evidence of a prior audit (inline `M-2`, `L-5` markers) and applies several strong controls: fail-closed CSRF origin checks, fail-closed rate limiting, HTTPS enforcement in production, authenticated-payload cross-checks on backups, and best-effort secret zeroization.

The most material residual risks are:

1. **Weak-fallback key derivation** when the WebAuthn **PRF extension is unavailable** (encryption falls back to semi-public credential `rawId`).
2. **Content-Security-Policy weaknesses** (`script-src 'unsafe-inline'`, broad `connect-src`) that reduce resilience to XSS-based secret exfiltration.
3. **Plaintext transport defaults** in the example configuration (`http://` RPC / signer).
4. **Support for unencrypted plaintext backup files** on the restore path.

None of these are exploitable for a remote key compromise on their own on the current testnet configuration, but each should be closed before mainnet.

1.2.1 Findings at a glance

ID	Title	Area	Severity
F-01	PRF-unavailable fallback derives keys from semi-public `rawId`	Backup/Key mgmt	Medium
F-02	CSP allows `script-src 'unsafe-inline'` and broad `connect-src`	Send/Receive (XSS)	Medium
F-03	Plaintext HTTP transport defaults for RPC / signer proxy	Send/Receive (transport)	Medium

ID	Title	Area	Severity
F-04	Restore accepts unencrypted plaintext backup files	Backup/Restore	Medium
F-05	Backup KDF uses PBKDF2 rather than a memory-hard function	Backup/Restore	Low
F-06	Decrypted `falcon_secret` string cannot be zeroized in JS	Send/Receive	Low
F-07	Clipboard copy of raw secret exposes it to other apps	Backup	Low
F-08	NetworkID omitted for ID <= 1024 -> cross-network replay surface	Send/Receive	Low
F-09	Backup outer `label`/`createdAt` metadata not authenticated	Backup/Restore	Informational
F-10	Unauthenticated read API performs 3 upstream RPC calls (amplification)	Receive	Informational

---

1.3 2. Scope & Architecture

1.3.1 2.1 In scope

- **Send:** `src/app/wallet/page.tsx` (`handleSend`), `src/lib/falcon-tx-sign.ts`, `src/lib/wallet-sign-client.ts`, `src/app/api/wallet/submit/route.ts`.
- **Receive / account view:** `src/app/api/wallet/account/route.ts`, `src/lib/rpc.ts`.
- **Backup / restore:** `src/lib/wallet-backup.ts`, `src/lib/wallet-crypto.ts`, `src/lib/passkey.ts`, `src/lib/wallet-store.ts`, and the corresponding UI handlers in `src/app/wallet/page.tsx`.
- **Key material:** `src/lib/falcon-keys.ts`, `src/lib/falcon-512-browser.ts`, `src/lib/falcon-wasm.ts`.
- **Cross-cutting controls:** `src/lib/origin.ts`, `src/lib/signer-proxy.ts`, `next.config.mjs` (security headers/CSP), `.env.example`.

1.3.2 2.2 Out of scope

Faucet drip logic, DEX/marketplace, validator node dashboard/SSRF proxy, explorer, and the external xrpld node and signing proxy infrastructure (reviewed only where they intersect the wallet flows).

1.3.3 2.3 Trust model

- **Client (browser)** is the trust anchor: keys are generated, encrypted, and used there.
- The **WebAuthn platform authenticator** (Secure Enclave / TPM / Android Keystore) gates access to the decryption key material via user verification (biometric/PIN).
- The **IndexedDB store** holds only ciphertext (`EncryptedSeed`).
- The **server / RPC node** is treated as **untrusted for confidentiality of secrets** - it only ever sees pre-signed transaction blobs and public addresses. This is the correct posture for a non-custodial wallet.

Figure 1: Falcon Logo

1.3.4 2.4 Data flow (send)

```
passkey.authenticate() -- user verification ---> keyBytes (PRF or rawId)
        |
        v
decryptSeed(IndexedDB ciphertext, keyBytes) ---> falcon_secret (in-memory)
        |
        v
signPayment() -- liboqs WASM sign ---> tx_blob   (secret zeroized after sign)
        |
        v
POST /api/wallet/submit -- origin-checked ---> xrpld RPC 'submit'
```

The private key never traverses the network in the primary flow. This is the single most important property and it holds throughout the reviewed code.

1.4 3. Severity Methodology

Ratings combine **impact** (loss of funds/keys, privacy, availability) and **likelihood** (attacker position required, preconditions), adjusted for the **testnet** context.

- **Critical** - remote key/fund compromise with low preconditions.
- **High** - key/fund compromise requiring a plausible attacker position.
- **Medium** - meaningful weakening of a control; exploitable under additional conditions.
- **Low** - limited or defense-in-depth impact.
- **Informational** - hardening / hygiene, no direct security impact.

1.5 4. Detailed Findings

1.5.1 F-01 - PRF-unavailable fallback derives encryption key from semi-public `rawId` (Medium)

Location: `src/lib/passkey.ts` (registration/assertion), `src/lib/wallet-crypto.ts` (`deriveAesKey`).

Observation. The at-rest encryption key for `falcon_secret` is derived via HKDF-SHA256 from either (a) the WebAuthn **PRF extension** output (32 bytes, strong, bound to the passkey private key) or (b) a fallback of the credential **`rawId`** bytes when PRF is unavailable. HKDF is a key-*derivation* function with **no work factor**; it is appropriate only when the input keying material is high-entropy. The `rawId` is semi-public/attacker-observable in some contexts and is not a secret of comparable strength to a PRF output.

If an attacker obtains both the IndexedDB ciphertext and the credential `rawId`, the seed is decryptable without the authenticator.

Impact. On PRF-less authenticators, the confidentiality of the stored seed degrades from "protected by hardware-bound secret" to "protected by a semi-public identifier." The code explicitly documents this as `M-2` and restricts it to testnet.

Likelihood. Requires local ciphertext access plus `rawId` recovery; not remote.

Recommendation. - Before any mainnet use, **require PRF** (refuse wallet creation, or add a mandatory user passphrase as a second KDF input) when PRF is unavailable. - When falling back, mix a **user passphrase through a memory-hard KDF** (see F-05) so the ciphertext is not recoverable from `rawId` alone. - Surface a clear, non-dismissable UI warning when `hasPrf === false`.

1.5.2 F-02 - CSP permits `script-src 'unsafe-inline'` and broad `connect-src` (Medium)

Location: `next.config.mjs` (`headers()` -> `Content-Security-Policy`).

Observation. The CSP is generally strong (`object-src 'none'`, `frame-ancestors 'none'`, `base-uri 'self'`, `form-action 'self'`), but: - `script-src` includes `'unsafe-inline'`, which defeats the primary purpose of CSP as an XSS mitigation. - `connect-src 'self' https: wss:` allows the page to open connections to **any** HTTPS/WSS host, so an injected script could exfiltrate data to an attacker-controlled endpoint.

Impact. In the event of an XSS (e.g. via a compromised dependency), an attacker script could wait for the user to authenticate/sign, read the transiently decrypted `falcon_secret` from memory, and exfiltrate it. The passkey user-verification gate raises the bar (the attacker must ride a legitimate user action) but does not eliminate the risk.

Recommendation. - Remove `'unsafe-inline'` from `script-src`; adopt **nonce-** or **hash-based** inline script allowances. Next.js supports per-request nonces. - Tighten `connect-src` to the specific RPC/API origins actually used rather than `https:`. - Retain `'wasm-unsafe-eval'` (required by liboqs) but scope everything else.

1.5.3 F-03 - Plaintext HTTP transport defaults for RPC and signer proxy (Medium)

Location: `.env.example` (`XRPLD_RPC_URL=http://...:6005`, `SIGNER_PROXY_URL=http://...:3001`), `src/lib/rpc.ts`, `src/lib/signer-proxy.ts`.

Observation. The example configuration ships plaintext `http://` endpoints. Signed `tx_blob`s (and, on the legacy signing path, the signer bearer token / forwarded secret) would traverse an unencrypted channel. The code contains **good compensating controls**: it warns on plaintext in production and `signer-proxy.ts` **throws** unless `ALLOW_INSECURE_TRANSPORT=true`. However, the shipped defaults model an insecure deployment.

Impact. A network MITM on a plaintext deployment can observe addresses and transaction contents (privacy), and - because Falcon signatures cover the transaction - **suppress or delay** submissions (availability), though they cannot forge a valid signature. On the legacy server-sign path the bearer token would be exposed.

Recommendation. - Change example defaults to `https://` and document TLS/mTLS or an SSH tunnel as the supported transport. - Keep the production `throw` guard; consider extending the same hard failure to `rpc.ts` (currently a warning only) for the submit path.

1.5.4 F-04 - Restore path accepts unencrypted plaintext backup files (Medium)

Location: `src/lib/wallet-backup.ts` (`PlainBackupFile`, `parseBackupFile`), `src/app/wallet/page.tsx` (`handleRestoreFile`, `finishRestore`).

Observation. While the UI only *creates* encrypted backups (`createEncryptedBackup`), `parseBackupFile` and the restore handler still accept a `PlainBackupFile` with `encrypted: false` and a cleartext `falcon_secret`. This preserves the possibility (and therefore the temptation) of plaintext key files on disk, cloud sync, or messaging apps.

Impact. A plaintext backup is a full, unprotected private key. Any process, backup system, or cloud sync with file access obtains the account.

Recommendation. - If plaintext backups are not a required feature, **remove `PlainBackupFile` support** entirely (both parse and restore) so only encrypted files are accepted. - If they must remain (interop), gate acceptance behind an explicit, clearly-worded user confirmation and never generate them from the UI.

1.5.5 F-05 - Backup KDF uses PBKDF2 rather than a memory-hard function (Low)

Location: `src/lib/wallet-backup.ts` (`derivePassphraseKey`).

Observation. Encrypted backups use **PBKDF2-SHA256 at 210,000 iterations** with a 32-byte salt and AES-256-GCM - a reasonable, standards-aligned choice. However, PBKDF2 is cheap to parallelize on GPUs/ASICs. Because the backup file contains the **full `falcon_secret`** and may be stored in less-trusted locations, an offline attacker who obtains the file can mount a high-throughput dictionary attack against weaker passphrases.

Mitigating control. `validateBackupPassphrase` enforces >=12 characters and >=3 character classes, which meaningfully raises the cracking cost.

Recommendation. Migrate the backup KDF to a **memory-hard function (Argon2id**, or scrypt) via a versioned backup format so existing files remain decryptable. Keep the passphrase policy.

1.5.6 F-06 - Transiently decrypted `falcon_secret` cannot be zeroized in JavaScript (Low)

Location: `src/lib/wallet-crypto.ts` (`decryptSeed` returns a `string`), `src/lib/falcon-tx-sign.ts` (`signPrepared`), `src/app/wallet/page.tsx` (`handleSend`).

Observation. The code zeroizes the raw secret-key **byte arrays** (`zeroize(decoded.secretKey)`) after signing - good practice. However, the decrypted `falcon_secret` is handled as a JS **`string`**, which is immutable and garbage-collected; it cannot be reliably wiped and may persist in heap/GC memory (and in React state during the flow) until collection.

Impact. Widens the in-memory exposure window; primarily relevant in combination with an XSS (F-02) or a memory-inspection capability.

Recommendation. Where feasible, keep the decrypted secret as a `Uint8Array` end-to-end (decrypt -> decode -> sign) and zeroize it in a `finally` block, avoiding intermediate `string` representations and long-lived React state for the plaintext secret.

1.5.7 F-07 - Clipboard copy of the raw secret exposes it to other applications (Low)

Location: `src/app/wallet/page.tsx` (`copyFalconSecret`).

Observation. The "copy secret" action writes the plaintext `falcon_secret` to the system clipboard. The clipboard is readable by other apps and clipboard-history managers. The code includes a **thoughtful**

mitigation: it auto-clears the clipboard after ~30s, but only if it can read the clipboard back and confirm the value is unchanged (best-effort; may be blocked).

Impact. During the exposure window, malware or clipboard managers can capture the key.

Recommendation. Prefer the encrypted-file backup as the primary path; de-emphasize raw copy. Keep the auto-clear, shorten the window, and warn the user that clipboard managers may retain history.

1.5.8 F-08 - NetworkID omitted for network IDs <= 1024 leaves a cross-network replay surface (Low)

Location: `src/lib/falcon-tx-sign.ts` (`INCLUDE_NETWORK_ID = NETWORK_ID > 1024`), `src/lib/signer-proxy.ts` (same rule).

Observation. Per XRPL rules, `NetworkID` is only included in the signed transaction for networks with ID > 1024. The Falcon Ledger testnet uses **ID 1001**, so `NetworkID` is **not** part of the signed payload. Replay is still bounded by `Sequence` and `LastLedgerSequence`, but a transaction signed for this chain could, in principle, be valid on another chain sharing an ID <= 1024 where the same account/sequence exists.

Impact. Low on an isolated testnet; the practical preconditions (identical account and sequence on a second chain) are unlikely.

Recommendation. This is protocol-conformant and largely unavoidable at ID 1001; note the consideration for any future chain-ID selection (choosing an ID > 1024 enables domain separation via `NetworkID`).

1.5.9 F-09 - Backup outer metadata (`label`, `createdAt`) is not authenticated (Informational)

Location: `src/lib/wallet-backup.ts` (`EncryptedBackupFile`, `decryptBackupFile`).

Observation. AES-GCM authenticates only the ciphertext payload. The outer `address`/`label`/`createdAt` fields sit outside the AEAD. The code correctly mitigates the security-relevant case (`L-5`) by verifying that the **authenticated** `payload.address` matches the outer `address` after decryption. `label`/`createdAt` remain unauthenticated but are non-security display fields.

Recommendation. Optionally bind all displayed metadata into the AEAD as Additional Authenticated Data (AAD) for completeness. No functional impact today.

1.5.10 F-10 - Unauthenticated read API fans out to three upstream RPC calls (Informational)

Location: `src/app/api/wallet/account/route.ts`.

Observation. `GET /api/wallet/account` validates the address format but is unauthenticated and un-rate-limited, and it issues **three** upstream RPC calls (`account_info`, `account_tx`, `server_info`) per request. This is a mild request-amplification / DoS lever against the serverless function and the RPC node.

Recommendation. Apply lightweight per-IP rate limiting (the project already integrates Upstash) and/or cache `server_info` briefly to reduce fan-out.

1.6 5. Positive Observations (Controls Done Well)

- **Non-custodial by construction.** `falcon_secret` is generated and used in-browser via WASM and is never transmitted in the primary flow (`wallet-sign-client.ts`, `falcon-tx-sign.ts`).
- **Legacy server-signing endpoint disabled by default** and, when enabled, still enforces a strict origin allow-list (`src/app/api/wallet/sign/route.ts`).

- **Fail-closed CSRF defense.** `isOriginAllowed` denies in production when `ALLOWED_ORIGINS` is unset and compares exact origins (no suffix bypass) (`src/lib/origin.ts`).
- **Fail-closed rate limiting** for the funded faucet, protecting against drain (`src/lib/rate-limit.ts`).
- **Strong backup crypto primitives** - AES-256-GCM, 210k-iteration PBKDF2, random 32-byte salt / 12-byte IV, and a real passphrase-strength policy.
- **Authenticated-payload cross-check** on backup decryption (`L-5` mitigation).
- **Best-effort key hygiene** - secret-key byte arrays are zeroized after signing; WASM heap buffers are freed.
- **Sensible security headers** - `X-Frame-Options: DENY`, `nosniff`, `frame-ancestors 'none'`, restrictive `Permissions-Policy`.
- **Server error messages are sanitized** to avoid leaking internal details (`submit`/`account` routes).
- **Input validation** on destination address, amount, `tx_blob` shape, and `falcon_secret` length/format bounds (mitigates unbounded-hash inputs on `derive`).

1.7 6. Prioritized Recommendations

Before mainnet / real value (must-fix): 1. **F-01** - Require PRF or add a mandatory passphrase second factor; never protect a real-value key with `rawId` alone. 2. **F-02** - Remove `'unsafe-inline'` from `script-src` (nonce/hash) and tighten `connect-src`. 3. **F-03** - Enforce HTTPS/mTLS transport by default; extend the hard-fail guard to the submit RPC path. 4. **F-04** - Remove or hard-gate plaintext backup acceptance.

Hardening (should-fix): 5. **F-05** - Move backup KDF to Argon2id/scrypt with a versioned format. 6. **F-06** - Keep the plaintext secret as a zeroizable `Uint8Array`; avoid long-lived state. 7. **F-10** - Rate-limit / cache the account read API.

Hygiene (nice-to-have): 8. **F-07**, **F-08**, **F-09** - Clipboard UX warnings, chain-ID note, and AAD-bound metadata.

1.8 7. Testnet vs. Mainnet Posture

The application is **appropriately secured for its stated testnet purpose**, and the code consistently and honestly labels its accepted risks (`M-2`, `L-5`) as testnet-only. The README and inline comments repeatedly warn not to reuse these wallets for real value. That guidance is **correct and should be enforced in product** - the residual Medium findings above (PRF fallback, CSP, transport, plaintext backups) collectively represent the gap that must be closed before this wallet handles funds of real value.

End of report.