VxBlocks Neptune Wallet App - Code Review by Keyper Labs

Date: October 28th, 2025

Disclaimer: This review is provided for informational purposes only. Keyper Labs AG makes no representations or warranties, express or implied, regarding the accuracy, completeness, or security of the reviewed code. This review does not constitute an audit or any form of certification. You are solely responsible for independently evaluating and verifying the security and suitability of any open-source software you choose to use.

Introduction

The following document outlines a code review of the Neptune Wallet created by VxBlocks.

The primary goal of the code review is to assess the overall security of the wallet, mainly concerning safekeeping private keys. We did not focus on privacy-related issues.

Full link: https://github.com/VxBlocks/neptune-wallet-app

The review was done on: https://github.com/VxBlocks/vxb_neptune_wallet/tree/v2.0.1

TLDR

We **do NOT recommend this wallet** due to multiple critical security problems.

Most Critical Problems

Below are the most critical problems found on the app.

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
FE-28	SQL Injection via generic query execution	CRITICAL	Code: `src/utils/storage/index.ts` Backend exposes a function that lets the frontend send raw SQL queries.
TAU-02	Content Security Policy Disabled	CRITICAL	Code: `src-tauri/tauri.conf.json:36` , `src-tauri/tauri.android.conf.json:36` The application **explicitly disables** Content Security Policy by setting `"csp": null` , expose the core process to several schema of attack (XSS attack, Key exfiltration, Transaction Manipulation, Supply Chain attack from frontend).
TAU-03	RPC handlers touch the full filesystem	CRITICAL	Code: `src-tauri/capabilities/migrated.json:34` `opener:allow-open-path` permission uses wildcard scope allowing access to any file on the system
RPA-01	No [profile.release] hardening in Cargo; defaults leave	CRITICAL	Code: `src-tauri/Cargo.toml.` Shipping binaries with wrap-on-overflow arithmetic and debug symbols

	overflow checks off, panic = "unwind", no LTO/strip		violates Rust hardening guidance.
RPA-02	RSA timing attack via sqlx-mysql → rsa (RUSTSEC-2023-0071)	CRITICAL	**Path:** `sqlx-mysql → sqlx → neptune-wallet` Disable MySQL features in sqlx; wallet only needs SQLite. Advisory: https://rustsec.org/advisories/RUSTSEC-2023-0071.html.
BCI-01	Auto-update pulls unsigned JSON and binaries from mutable main branch	CRITICAL	**Code:** `src-tauri/src/service/app.rs:33` downloads update.json from refs/heads/main with no signature or checksum validation; Tauri updater plugin isn't configured for signature enforcement (tauri.conf.json).

Architecture

The reviewed application is built using Tauri framework.

> *Tauri is a framework for building tiny, fast binaries for all major desktop and mobile platforms. It is used for building applications for desktop computers using a combination of Rust tools and HTML rendered in a Webview.*

In this implementation, the development team used React and Next.js for the frontend rendering. The diagram below provides a high-level overview of the main architectural components that were analyzed during the review.

From the Tauri documentation:

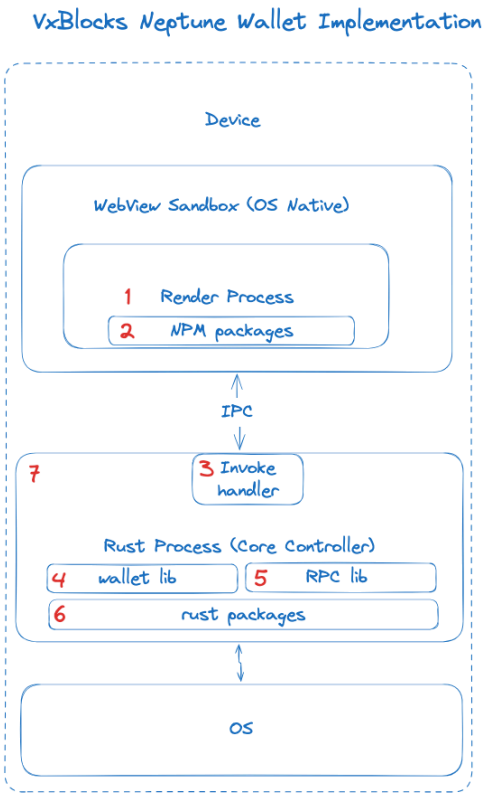
> *Each Tauri application has a core process, which acts as the application's entry point and which is the only component with full access to the operating system. The Core's primary responsibility is to use that access to create and orchestrate application windows, system-tray menus, or notifications. Tauri implements the necessary cross-platform abstractions to make this easy. It also routes all Inter-Process Communication through the Core process, allowing you to intercept, filter, and manipulate IPC messages in one central place. (source)*

In short, a Tauri application consists of **two main processes**:

1. **Core process** – the privileged process with direct OS access.
2. **Render process** – a webview running the application's UI.

It's critical to **strictly control communication between these processes**. The render process should be granted only the minimal required permissions, while the core process should be carefully scoped to limit potential attack surfaces.

The diagram below displays the VxBlocks Neptune Wallet implementation and how the different components interact within the Tauri application model. We will leverage the numbers in red to guide you through the analysis. Not featured in this diagram are the build process, CI/CD, and binaries, which can be found in Section 8.



1. Render Process

ASSURANCE: VERY LOW

INTRODUCTION

Significant concerns have been identified, including direct SQL execution from untrusted contexts, client-side generation of cryptographic seeds, and improper security boundary enforcement. These issues require remediation before the production release. While the wallet is potentially functional, these patterns do not meet the security standards expected for a cryptocurrency wallet handling user funds.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
FE-28	SQL Injection via generic query execution	CRITICAL	Code: `src/utils/storage/index.ts` Backend exposes a function that lets the frontend send raw SQL queries.
FE-27	Seed phrase and password in redux state	HIGH	Code: `src/store/wallet/wallet-slice.ts` the generated mnemonic is stored in the Redux store as plaintext until the application is closed or reloaded

2. NPM packages for render process

ASSURANCE: LOW

INTRODUCTION

The wallet includes unused NPM dependencies, which increase the attack surface and create unnecessary maintenance work. Additionally, version incompatibilities between packages cause conflicting peer dependencies, which can affect the results of a security audit and may also introduce certain risks or security gaps if not handled properly.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
NPM-25	Version incompatibilities and dependency conflicts	HIGH	Version incompatibilities have been identified among several project packages, resulting in conflicting peer dependencies. An NPM install process fails unless the `--legacy-peer-deps` flag is explicitly provided. This is the result of inconsistent dependency management and potential instability across environments.
NPM-24	Unused dependencies	MEDIUM	The project list multiple NPM dependencies that are not used in the codebase. While no known critical vulnerabilities were identified during a brief check of the dependencies at the time of this document, unused dependencies increase the supply chain attack surface unnecessarily and create maintenance overhead.

LIMITATIONS AND ASSUMPTIONS

The presence of conflicting peer dependencies limits the reliability of security auditing tools such as **NPM audit**. Because the dependency tree cannot be properly resolved without using the `--legacy-peer-deps` flag, the audit results may be incomplete or inaccurate, potentially overlooking vulnerabilities or producing false positives. Additional checks were conducted using Yarn and Retire.js. These tools were used to cross-check the dependency tree and verify the accuracy of the audit process, ensuring consistent results across different package managers.

---

3. Invoke Handler

ASSURANCE: LOW

INTRODUCTION

The invoke handler exposes an overly broad control surface. The renderer can obtain a deterministic bearer token and reuse it indefinitely, meaning any script that runs in the WebView permanently inherits backend privileges. Several IPC handlers rely on `unwrap()` (or similar panic-prone patterns); a single malformed payload from the UI can crash the Rust process. Input structures accept unknown fields and unbounded arrays, so an attacker can simply submit large blobs to exhaust memory or trigger logic faults.

Error handling compounds the problem: full response bodies are logged on failures, turning the logs into a side channel that can leak balances, UTXO lists, and other sensitive data. With no throttling or request-limiting in place, the invoke surface can be cheaply exploited for denial-of-service and data-exfiltration attempts rather than serving as a narrow, defensive IPC boundary.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
IH-13	caches passwords indefinitely in RAM	HIGH	Code: `src-tauri/src/config/mod.rs:26` Once unlocked, the password remains in RAM **indefinitely** (`Mutex<Option<String>>`) without time-to-live (TTL) or auto-lock mechanism. Extended memory exposure increases attack window for memory dump attacks.
IH-05	Returns a deterministic bearer token	HIGH	Code: `src-tauri/src/rpc/commands.rs:33` The RPC authentication system uses a **deterministic Bearer token** derived from a P-256 public key. The token: - Has **no expiration** - Has **no rotation mechanism** - Is **predictable** (same secret → same token) - Remains valid until password change If leaked once (via XSS, log files, memory dump), the attacker retains **permanent access** to the wallet RPC.
IH-06	Panic-Prone Error Handling	MEDIUM	Code: `rpc/mod.rs:245, rpc/mod.rs:277, rpc/mod.rs:169, rpc/tls/mod.rs:39, rpc/tls/aes.rs:25-28, session_store/*.rs:22,27,35,36,42,47, wallet/mod.rs:348, wallet/spend.rs:Multiple, config/mod.rs:Multiple` The codebase contains **78 instances of** `unwrap()` **and** `expect()` , many in critical RPC-exposed routes. These panic-prone patterns allow attackers to **crash the entire process** via malformed inputs, causing denial of service.
IH-07	Lax JSON Deserialization	MEDIUM	Code: `src-tauri/src/rpc/mod.rs : 340-353` RPC request structures lack `#[serde(deny_unknown_fields)]` and input validation, allowing clients to inject arbitrary fields that could exploit future parser bugs, logic errors, or cause DoS via unbounded inputs.

LIMITATIONS AND ASSUMPTIONS

This review focused exclusively on the IPC invoke layer at commit `054e972c17da468e09ba126294a1dc168246f269`. Analysis covered `src-tauri/src/command/commands.rs` and related invoke logic, with findings derived from static inspection of the active code paths. No runtime patches or mitigations were applied.

The scope was limited to functional and security behaviors observable in the source, particularly the handling of bearer tokens, deserialization boundaries, error management, and memory exposure.

All risks identified were inferred from the implemented Serde models, logging middleware, and panic-prone code present in that specific revision.

4. Wallet library in CoreBackend

ASSURANCE: LOW

INTRODUCTION

The wallet component handles key material with insufficient safeguards. Encryption keys are derived from a single SHA-256 iteration, empty passwords are accepted, and sensitive data remains in memory and logs after use. These gaps increase the risk of credential exposure and brute-force recovery.

Credential rotation is non-atomic, proof execution lacks isolation controls, and AES-GCM nonces are not managed defensively. A process crash or concurrent load can leave key material resident in memory, while unthrottled proof requests can cause resource exhaustion.

Overall, the wallet's cryptographic and memory-handling practices do not meet production security standards and require immediate hardening before deployment.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
WB-02	derives keys with a single SHA-256	HIGH	Code: `src-tauri/src/config/mod.rs:315` The Neptune Wallet derives encryption keys from user passwords using a single-pass SHA-256 hash, without salt, iteration count, or memory-hardness. This weak key derivation allows attackers with access to the encrypted database to run GPU-accelerated dictionary or brute-force attacks, potentially recovering passwords in a reasonable time-horizon depending on the systems used.
WB-03	Empty Password Allowed → No Disk Encryption	HIGH	Code: `src-tauri/src/config/mod.rs:159-172 (decrypt), 203-205 (set), 269-270 (encrypt_data)` The Neptune Wallet **allows users to set an empty password (`""`)**, which completely bypasses encryption and stores all sensitive data (mnemonics, secret keys, credentials) in **plaintext** on disk. This violates fundamental security principles for cryptocurrency wallets.
WB-11	#[derive(Debug)] on Structs with Secrets → Log Leakage	HIGH	Code: `src-tauri/src/config/mod.rs:23` The `Config` struct has `#[derive(Debug)]` while containing sensitive fields (`password`, `decrypt_key`). Any `debug!("{:?}", config)`, `error!("{:?}", config)`, or accidental

			logging will dump **plaintext passwords and keys to log files**.
WB-12	Non-Atomic Credential Updates in set_password	HIGH	Code: `src-tauri/src/config/mod.rs:189-220` The `set_password()` function performs multiple database writes **without wrapping them in a SQLite transaction**. If the process crashes between writes, the database ends up in an inconsistent state, causing permanent wallet lockout.
WB-18	CPU DoS - spawn_blocking Without Limits in Prover	MEDIUM	Code: `src-tauri/src/wallet/spend.rs:385-388` Proof generation uses `tokio::task::spawn_blocking` **without concurrency limits**. Each proof takes 1-5 minutes of CPU. Malicious clients can queue unlimited proofs, exhausting the thread pool and blocking legitimate transactions.
WB-04	Secrets in Memory Without Zeroization	MEDIUM	Code: `config/mod.rs:26,30,55` Sensitive data (passwords, encryption keys, mnemonics) are stored in memory as plain `String` and `Vec<u8>` types **without automatic zeroization on drop**. This leaves secrets vulnerable to memory dump attacks, debugger inspection, core dumps, and swap file leakage.
WB-20	AES-GCM Nonce Reuse Risk	MEDIUM	Code: `src-tauri/src/rpc/tls/aes.rs:25` AES-GCM encryption correctly uses `OsRng` for nonce generation, but **lacks tests verifying nonce uniqueness**. If `OsRng` fails or is compromised, nonce collisions = catastrophic cryptographic failure.
WB-22	Backtraces/Panic May Dump Sensitive Data	LOW	Panic backtraces in debug/development builds may include variable names and values containing secrets. Release builds should use `panic = "abort"` to prevent backtrace generation and potential secret exposure.

LIMITATIONS AND ASSUMPTIONS

The review was performed on the same scoped commit (`054e972c17da468e09ba126294a1dc168246f269`), focusing on key management, password handling, and proof-generation logic within `src-tauri/src/config/` and `src-tauri/src/wallet/` .

The assessment relied entirely on static analysis of the available Rust source code. No runtime modifications or patches were introduced, and all risks were inferred from the active implementation at that revision.

The analysis covered cryptographic key derivation, credential storage, zeroization practices, concurrency behavior, and error handling. Observations were derived from actual code inspection; empty-password

branches, single-pass SHA-256 key derivation, non-atomic writes, missing zeroization, and uncontrolled proof execution.

5. RPC library in CoreBackend

ASSURANCE: VERY LOW

INTRODUCTION

The RPC library exposes weak boundaries and relies on blind trust. It's still served over plain HTTP, no file access restriction, is bound to all interfaces, and accepts requests from any origin, effectively exposing the control plane to external access.

Blocks are fetched without proof verification, TLS isn't pinned, and the client accepts unverified data from the network. Authentication relies on a static token, error logs leak full responses, and there's no rate limiting or request throttling.

Together, these gaps leave the wallet surface wide open and highlight a core issue: missing the fundamental security controls expected in any production-grade wallet.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
RPC-01	The internal RPC server **binds to** `0.0.0.0` **(all network interfaces)** instead of `127.0.0.1` (loopback only)	HIGH	Code: `src-tauri/src/rpc/mod.rs:194` This exposes sensitive wallet operations to: - **Any process on the local machine** (malware, compromised apps) - **Potentially the local network** (if firewall allows) - **Cross-origin attacks** from malicious websites
RPC-19	HTTP client for Neptune-Core uses default settings without TLS	HIGH	Code: `src-tauri/src/rpc_client/mod.rs:59` HTTP client for Neptune-Core uses default `reqwest::Client::new()` without TLS configuration, certificate pinning, or hostname verification enforcement. Vulnerable to MITM attacks.
RPC-17	logs full response bodies on errors	MEDIUM	Code: `rpc/mod.rs:277` There are no restrictions in the logs, full response bodies are exposed on errors, leaking signatures and UTXOs into telemetry
RPC-08	no rate limiting middleware	MEDIUM	Code: `src-tauri/src/rpc/mod.rs` one hostile client can exhaust the RPC worker pool.

LIMITATIONS AND ASSUMPTIONS

The analysis was performed on the scoped commit `054e972c17da468e09ba126294a1dc168246f269`, focusing on the RPC server and client implementations under `src-tauri/src/rpc/` and `src-tauri/src/rpc_client/`. The review covered authentication, network boundaries, CORS behavior, TLS configuration, and block verification logic. All observations were made through static inspection of the existing codebase.

The review assumed the service was running under its documented defaults: binding to `0.0.0.0`, using plain HTTP without TLS pinning, and relying on a static bearer token for authentication. These conditions were validated as part of the configuration paths visible in the source.

6. Rust packages

ASSURANCE: VERY LOW

INTRODUCTION

The Rust package layer lacks the guard rails expected in production-grade wallet software. The release profile still relies on Cargo defaults, meaning builds ship with wrap-on-overflow arithmetic, unwinding panics, no link-time optimization, and unstripped symbols. This loose configuration is further compounded by an inherited RSA timing issue in `sqlx-mysql` and several outdated clipboard and GUI crates (`xcb`, `failure`, `gtk3`, `fxhash`) that either contain known memory-safety flaws or have been abandoned by their maintainers.

With the panic strategy set to *unwind* and no strict linting or `cargo audit/deny` enforcement, regressions, and vulnerable dependencies can easily slip into production. Hardening this component requires tightening the release profile, replacing or removing unmaintained crates, and introducing **CI/CD policies** that continuously validate the integrity of the dependency tree.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
RPA-01	No [profile.release] hardening in Cargo; defaults leave overflow checks off, panic = "unwind", no LTO/strip	CRITICAL	**Code:** `src-tauri/Cargo.toml.` Shipping binaries with wrap-on-overflow arithmetic and debug symbols violates Rust hardening guidance.
RPA-02	RSA timing attack via sqlx-mysql → rsa (RUSTSEC-2023-0071)	CRITICAL	**Path:** `sqlx-mysql → sqlx → neptune-wallet` Disable MySQL features in sqlx; wallet only needs SQLite. Advisory: https://rustsec.org/advisories/RUSTSEC-2023-0071.html.
RPA-03	XCB soundness issues (RUSTSEC-2021-0019) via x11-clipboard → clipboard-ext → neptune-wallet	HIGH	**Path:** `x11-clipboard` → `clipboard-ext` → `neptune-wallet` Upgrade xcb to ≥ 1.0 or replace the clipboard stack to eliminate memory-safety bugs. Advisory: https://rustsec.org/advisories/RUSTSEC-2021-0019.html
RPA-04	Deprecated failure crate pulled in through clipboard stack (RUSTSEC-2020-0036)	HIGH	**Path:** `which` → `clipboard-ext` → `neptune-wallet` Replace clipboard-ext / migrate off failure. Advisory: https://rustsec.org/advisories/RUSTSEC-2020-0036.html, https://rustsec.org/advisories/RUSTSEC-2022-0056.html
RPA-05	Unmaintained GUI/clipboard deps (xcb, GTK3 bindings, fxhash) flagged by	HIGH	**Multiple crates:** `atk, atk-sys, gdk, gdk-sys, gtk, gtk-sys, etc.` Affects clipboard-ext and Tauri GUI stack; upgrade

	RustSec (RUSTSEC-2024-0413-0240)		to maintained forks or remove features. Advisory: https://rustsec.org/advisories/RUSTSEC-2024-0413.html, ... https://rustsec.org/advisories/RUSTSEC-2024-0420.html
RPA-06	Panic strategy left at unwind for release builds	HIGH	**Code:** `src-tauri/Cargo.toml`. Adopt panic = "abort" per Rust/Tauri secure build recommendations.
RPA-07	No strict Rust linting/audit pipeline	MEDIUM	**Code:** `src-tauri/src/lib.rs:1` Missing #![deny(warnings)], clippy pedantic, and CI cargo audit/deny gates.

LIMITATIONS AND ASSUMPTIONS

This Code Review focused exclusively on dependencies tied to wallet functionality. Analysis was performed using `cargo-audit` v0.21.2 against the RustSec advisory database, covering 799 crates. The review targeted cryptographic, database, and data-handling libraries within `src-tauri/src/wallet/` and related configuration files. Additional validation was done through `cargo tree` to isolate wallet-specific dependencies and assess the integrity of their transitive graphs. Findings and recommendations are therefore limited to the scope of wallet-related crates and their direct impact on security posture.

---

7. Core Controller Process

ASSURANCE: VERY LOW

INTRODUCTION

The current Tauri V2 in Neptune Wallet presents several critical security gaps, many of which stem from features that are either missing or incorrectly implemented under Tauri v2's new security model.

The release build lacks hardening in its Cargo profiles, and the Content Security Policy is completely disabled. This effectively removes one of the main defenses against untrusted content execution. File access permissions are defined with broad wildcards (`**/*`), granting unrestricted read/write capability across the application space. On macOS, private API access remains enabled, exposing undocumented system interfaces.

Development features such as DevTools auto-open and verbose backtraces are still active in production, leaking internal state and runtime details. Capability definitions are incomplete, with missing or overly permissive scopes, and there are no runtime authority or IPC authorization checks in place.

These findings highlight a deeper issue: critical Tauri v2 mechanisms—Capabilities, Permissions, and Runtime Authority, are either not enforced or only partially configured. As a result, the wallet runs without the protective boundaries expected in a hardened production build.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
TAU-02	Content Security Policy Disabled	CRITICAL	Code: `src-tauri/tauri.conf.json:36`, `src-tauri/tauri.android.conf.json:36` The application **explicitly disables** Content

			Security Policy by setting `"csp": null`, expose the core process to several schema of attack (XSS attack, Key exfiltration, Transaction Manipulation, Supply Chain attack from frontend).
TAU-03	RPC handlers touch the full filesystem	CRITICAL	Code: `src-tauri/capabilities/migrated.json:34` `opener:allow-open-path` permission uses wildcard scope allowing access to any file on the system
TAU-06	Insufficient Capabilities Configuration	HIGH	Code: `src-tauri/src/lib.rs:24` There are no custom capability manifests or runtime-authority policies beyond that migrated file—no entries under src-tauri/permissions/ and no authority check the v2 permission model hasn't been adopted
TAU-08	missing Runtime Authority policies	HIGH	Code: `src-tauri/src/lib.rs` missing Runtime Authority policies in `src-tauri/src/lib.rs` means the renderer can escalate to privileged invokes with no audit trail
TAU-04	macOS Private API Enabled Without Justification	HIGH	Code: `src-tauri/tauri.conf.json:38`, `src-tauri/Cargo.toml:28` - **App Store rejection:** Apple prohibits private API usage - **Security vulnerabilities:** Private APIs may bypass sandboxing/security checks
TAU-05	Debug Features Enabled in Production Builds	MEDIUM	Code: `src-tauri/src/gui.rs:197-200` - **Information disclosure:** DevTools reveal wallet state, RPC endpoints, internal logic - **Debugging aid for attackers:** Inspect IPC calls, local storage, network traffic - **Verbose error messages:** Stack traces expose file paths, function names, internal state

LIMITATIONS AND ASSUMPTIONS

The review was performed on the scoped commit `054e972c17da468e09ba126294a1dc168246f269`, focusing on configuration, capability, and build-profile settings under `src-tauri/`. Despite legacy references to LevelDB, the wallet currently uses SQLite, confirmed through inspection of `src-tauri/src/config/`.

Analysis relied entirely on static code review — no runtime instrumentation or modified builds were used. The review included configuration files, capability manifests, and Cargo profile settings to identify gaps in sandboxing, permission control, and build hardening.

Assumptions were based on the current defaults: files created with standard permissions, no external disk encryption applied, and sensitive material protected only by the wallet's password logic. Under these conditions, an empty password directly exposes plaintext data.

8. CICD, Binaries, Distribution mechanism, and Auto-updaters

ASSURANCE: VERY LOW

INTRODUCTION

The build and delivery process remains the weakest point in Neptune Wallet's security posture. Binaries are still produced manually on developer machines, with no continuous integration or deployment pipeline in place. There is no provenance tracking or verification mechanism to ensure that distributed builds originate from trusted sources.

The auto-update mechanism retrieves an unsigned `update.json` directly from the mutable `main` branch, with no signature or checksum enforcement. This leaves the update path exposed to tampering or remote code execution in the event of a compromised repository or access token.

Release artifacts across all platforms remain unsigned—macOS builds are not notarized, Windows bundles lack Authenticode signatures, and there is no reproducibility or verifiable build trace. Additionally, the only Git dependency (`neptune-cash`) is pinned to a branch rather than a fixed commit, allowing upstream changes to silently alter build outputs.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
BCI-01	Auto-update pulls unsigned JSON and binaries from mutable main branch	CRITICAL	**Code:** `src-tauri/src/service/app.rs:33` downloads update.json from refs/heads/main with no signature or checksum validation; Tauri updater plugin isn't configured for signature enforcement (tauri.conf.json).
BCI-02	No automated CI/CD or supply-chain gates; builds are entirely manual	HIGH	**No CI/CD detected:** No continuous integration or deployment pipelines were found under `.github/workflows/` or equivalent directories. Local builds are driven by `taskfile.yml`, which already highlights the absence of CI-based checks such as `cargo audit` or `cargo deny`.
BCI-03	Release artifacts are unsigned/not notarized across platforms	HIGH	**Code:** `src-tauri/tauri.conf.json` lacks macOS signing/notarization fields; no Authenticode config; docs mention Gatekeeper bypass. Without signatures, MITM or tampered downloads go undetected.
BCI-04	Build provenance and integrity controls absent	HIGH	**Code:** `taskfile.yml` shows builds run from dev workstations; no reproducible builds, no SBOM, no checksums or cosign/SLSA provenance. Aligns with TAURI configuration audit recommendations for release hardening.
BCI-05	Git dependency neptune-cash pinned only to a branch, not a commit	MEDIUM	**Code:** `src-tauri/Cargo.toml:18-24` uses branch = "wallet" without rev, making builds non-deterministic and vulnerable to upstream tampering; violates reproducibility guidance.

LIMITATIONS AND ASSUMPTIONS

This review focused exclusively on the build, packaging, and update pipeline for Neptune Wallet at commit `054e972c17da468e09ba126294a1dc168246f269`. The analysis relied on static inspection of the project's configuration files—primarily `taskfile.yml`, `src-tauri/tauri.conf.json`, and `src-tauri/src/service/app.rs`—as well as dependency definitions in `Cargo.toml`. No CI/CD pipelines or automation scripts were found under `.github/workflows/` or equivalent directories, and no builds were executed during this review.

The evaluation assumed the current public release process mirrors the observed state: binaries produced manually on developer workstations, updates pulled from a mutable `main` branch, and no cryptographic signing, notarization, or provenance applied to release artifacts. The same assumption extends to dependency integrity, as the dependency `neptune-core` is fetched directly from GitHub and pinned only to a branch rather than a fixed commit (`rev`)