Security Audit Review

Perpetual Bitcoin Protocol

Review Date	April 4, 2026
Reviewer	AI-assisted manual source review
Solidity Version	0.8.24
Target Chain	PulseChain

> *This review is a manual source-level security assessment based on reading, tracing, and adversarial analysis of all contracts in scope. It is not a substitute for a formally contracted independent audit with legal liability coverage. All findings reflect the state of the code at the time of review.*

Scope

Contract	Role
Vault.sol	Core protocol: buys, netting, unlocks, LP, VLock
PB.sol	Liquid ERC20 token (21M supply)
PBc.sol	Non-transferable locked claim token (21M supply)
PBt.sol	Non-transferable ERC721 position tracker
PBr.sol	Non-transferable ERC1155 recovery badge
PBi.sol	Non-transferable ERC1155 inheritance badge
LaunchConverter.sol	One-time batched presale-to-PBt converter
PulseXInterface.sol	Read-only spot price feed from PulseX pair
PBRemoveUserLP.sol	LP removal middleware (pair → vault → user)
PresaleIOU.sol	Presale IOU NFT (pre-launch)
VaultViews.sol	Read-only view companion to Vault

Executive Summary

Perpetual Bitcoin implements a vault-centric, fully immutable (no-admin, no-upgrade) architecture. All PB purchases are routed through Vault.sol, which enforces a deterministic 3.69% liquid / 96.31% locked split, tracks positions via non-transferable PBt NFTs, and executes geometric-progression unlocks.

The protocol's most significant security property is PB.sol's pair guard: direct pair-to-user PB transfers are blocked (from == PAIR && to != VAULT → revert). This effectively prevents flash-loan-based price manipulation on the PB/USDL pair, since the only path to increase price is through buyPBDirect() which commits real economic value.

Key Findings

Severity	Count	Summary
HIGH ✓ **RESOLVED**	1	Compilation blocker: undefined _escapeJsonString in PBr.sol and PBi.sol — RESOLVED
MEDIUM	1	Password plaintext in calldata (structurally non-exploitable)
LOW	1	No recovery time-lock (design trade-off)
INFORMATIONAL	7	Precision loss; gas patterns; dead TWAP storage; permissionless functions; internal minPBOut=1; unchecked PBc returns; VLock bonus decay; FOAD distribution bypass

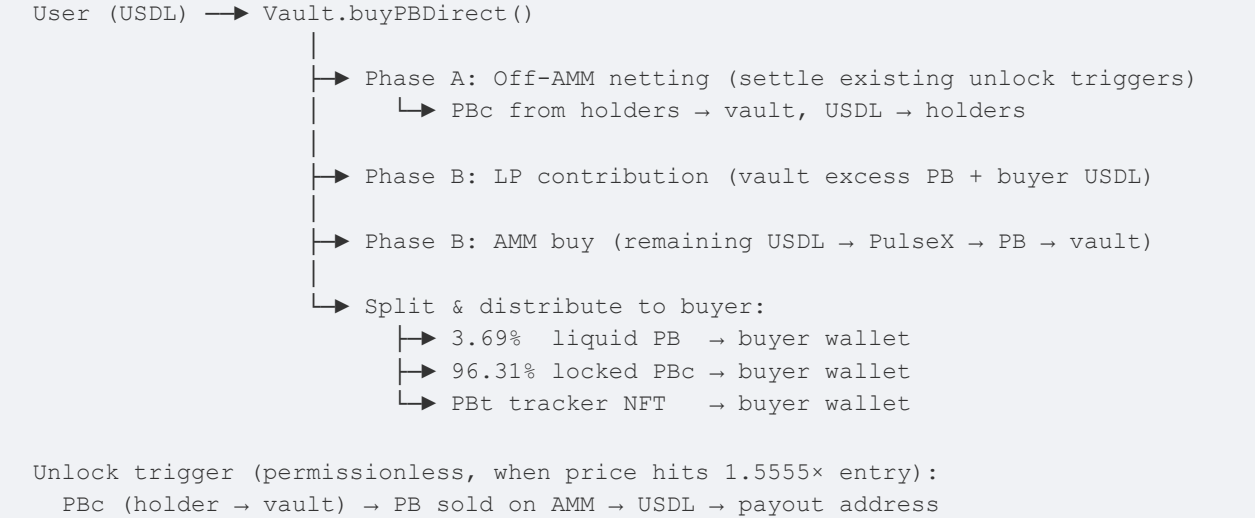
> *No critical exploit path was identified in the live protocol flow. The one HIGH finding is a deployment blocker for PBr/PBi (trivial fix). All MEDIUM findings have practical mitigations via the protocol's own design constraints.*

Architecture Overview

Confirmed Design Decisions

- **All buys route through Vault**
  - PB.sol enforces vault-only receipt from the PulseX pair. No direct AMM purchase is possible.
- **All buys split 3.69% / 96.31%**
  - Liquid PB + locked PBc on every purchase. No exceptions.
- **PBc is non-transferable between users**
  - Only moves user↔Vault via vault-mediated calls.
- **PBt (tracker NFT) is non-transferable**
  - Ownership changes only via vault-mediated recovery/inheritance.
- **No admin, no upgradability, no kill-switch**
  - After lockImmutableReferences(), deployer has zero privileges.
- **Spot price for unlock triggers**
  - Intentional since the vault is the only buyer on PulseX; pair price reflects real market state.
- **LP harvest via K-tracking**
  - Only fee growth is extracted. LP principal is never withdrawn.
- **Badge messages**
  - _isValidMessage whitelist (alphanumeric + space + dash + comma + period) ensures no JSON-unsafe characters.

Token Flow



Findings

H-1: Compilation blocker — undefined _escapeJsonString

Severity: HIGH | Status: **RESOLVED** | Location: *PBr.sol:195, PBi.sol:200*

Both badge contracts reference `_escapeJsonString(msg_)` in their `uri()` function, but this function is not defined anywhere in the contract, its imports, or inherited contracts (OpenZeppelin ERC1155). Hardhat compilation fails.

Additionally, the computed `escapedMsg` variable is never used — the raw `msg_` is inserted directly into the JSON output.

Risk: PBr and PBi cannot be compiled or deployed in their current state.

Mitigation: The `_isValidMessage` whitelist already restricts input to `[A-Za-z0-9 ,.-]`, which contains no JSON-special characters. Escaping is unnecessary.

Fix: Remove the dead line from both contracts: `- string memory escapedMsg = _escapeJsonString(msg_);`

M-1: Recovery/inheritance password visible in mempool

Severity: MEDIUM | Status: **Practical risk: LOW** | Location: *Vault.sol:_activateSuccession()*

When a recovery or inheritance address holder calls `activateRecovery()` or `activateInheritance()`, the plaintext password is passed as calldata and is visible in the mempool.

Why front-running is structurally impossible:

• msg.sender must match the registered recovery/inheritance address — a third party cannot satisfy this check regardless of knowing the password.

• The badge is non-transferable — there is no way to acquire someone else's badge.

• The password serves as a second factor; even with the recovery wallet's key compromised, the attacker needs the password too.

Optional improvement: A commit-reveal scheme would eliminate mempool exposure entirely, at the cost of requiring two transactions.

M-2: Unchecked USDL transferFrom return value in PresaleIOU

Severity: INFORMATIONAL | Status: **Already deployed, not modifiable** | Location: *PresaleIOU.sol:160*

The return value of `IERC20(USDL).transferFrom(msg.sender, PRESALE_TREASURY, totalCost)` is not checked. If USDL returned false instead of reverting, a buyer could acquire presale blocks without paying.

Non-issue because: PresaleIOU is already deployed on PulseChain mainnet and is immutable. USDL on PulseChain reverts on failure (does not return false). Vault.sol consistently checks return values, so this pattern does not carry forward.

L-1: No time-lock or cooling period on recovery/inheritance activation

Severity: LOW | Location: *Vault.sol:_activateSuccession()*

Recovery and inheritance activation is instant (single transaction). If an attacker obtains both the recovery address private key and the password, they can immediately redirect all future unlock payouts.

Design rationale: The protocol is designed to be fully autonomous with zero admin post-deployment. A time-lock would require either an admin role or a second on-chain interaction from the original holder (problematic for deceased holders).

The dual-sig + non-transferable badge model is sound. The two-factor design (wallet address + password) correctly prioritizes the succession scenario over the compromise scenario.

L-2: Unchecked PBc transferFrom return values in Vault

Severity: INFORMATIONAL | Status: **No practical risk** | Location: *Vault.sol:532, 702, 1132, 1173*

Four calls to `IERC20(PBC_TOKEN).transferFrom(...)` do not check the return value.

Why this is safe: PBc is a protocol-internal contract. Its transferFrom requires msg.sender == VAULT and to == VAULT. On insufficient balance, Solidity 0.8.24 arithmetic underflow reverts — it never silently returns false. PBc is immutable, so this cannot change.

Style note: Wrapping these calls to match the pattern used elsewhere in Vault.sol is a cosmetic improvement with zero security impact.

I-1: computeNextTriggerPrice precision loss from divide-before-multiply

Severity: INFORMATIONAL | Location: *Vault.sol:computeNextTriggerPrice()*

The expression `price = (price / UNLOCK_DIVISOR) * UNLOCK_MULTIPLIER` loses up to UNLOCK_DIVISOR - 1 (9,999) wei per iteration.

For realistic prices (> 0.001 USDL/PB = 1e15 wei) the relative error is negligible. The alternative (multiply first) risks overflow at extremely high prices/indices. The comment explicitly documents this design choice.

Assessment: Intentional and acceptable.

I-2: _sanitizeUnlockHints insertion sort is O(n²)

Severity: INFORMATIONAL | Location: *Vault.sol:_sanitizeUnlockHints()*

With MAX_UNLOCK_PER_BUY = 500, worst-case insertion sort performs ~125,000 comparisons. The frontend self-limits to ~250, and gas costs fall on the caller (self-griefing). Not exploitable against other users.

I-3: PulseXInterface stores TWAP values but never uses them for pricing

Severity: INFORMATIONAL | Location: *PulseXInterface.sol*

Cumulative price values are stored via `updatePrice()` but `getCurrentPrice()` and `getPriceHistory()` both return the live spot price from reserves. The cumulative values are observability-only and are never used for TWAP computation.

Confirmed intentional by developer. Spot price is correct since the vault is the sole buyer on PulseX, making the pair price an accurate reflection of market activity.

I-4: executeUnlock and harvestLPRewards are permissionless

Severity: INFORMATIONAL | Location: *Vault.sol:executeUnlock(), Vault.sol:harvestLPRewards()*

Anyone can call executeUnlock(pbtId) for any position, and anyone can call harvestLPRewards(). In both cases, funds always go to the correct addresses and callers cannot steal funds.

By design — the protocol is fully autonomous post-deployment. Permissionless unlocks ensure positions are settled deterministically when price triggers are met, consistent with the 'no admin, no pause, genuinely autonomous' design philosophy.

I-5: Internal AMM buy uses minPBOut = 1

Severity: INFORMATIONAL | Location: *Vault.sol:1291*

The internal AMM swap has effectively zero individual slippage protection: `_executeBuyOnPulseX(ammBuyAmount, 1)`

The outer `buyPBDirect()` enforces the user's slippage tolerance on total PB received. A sandwich attacker would need to buy PB from the pair — but PB.sol's pair guard blocks all non-vault recipients. Not exploitable.

I-6: PresaleIOU.distributeUnsoldToFOAD bypasses MAX_BLOCKS_PER_ADDRESS

Severity: INFORMATIONAL | Location: *PresaleIOU.sol:distributeUnsoldToFOAD()*

When distributing unsold blocks to founders, `blocksPurchased[recipient]` is incremented but not checked against MAX_BLOCKS_PER_ADDRESS (5). Founders receiving unsold blocks could exceed 5 total.

Assessment: Treasury-only function, intentional for founder distribution.

Design Strengths

S-1: Flash Loan Immunity via PB.sol Pair Guard

The check if (from == PAIR && to != VAULT) revert in PB.sol is the protocol's most important security property. It means:

- No one can buy PB directly from the PulseX pair — only the vault can receive PB from AMM swaps
- Price can only be pushed UP via buyPBDirect() — which commits real USDL and creates real positions
- Price can only be pushed DOWN via user sells — which decreases the attacker's own PB holdings
- Flash loan price manipulation is impossible — you cannot borrow USDL, inflate price, trigger unlocks, and repay in one tx

This single guard eliminates the most common DeFi attack vector.

S-2: Pre-Buy Price Snapshot for Trigger Computation

Using the pre-buy price (before the buyer's own AMM activity inflates it) ensures fair trigger computation. This prevents the buyer from artificially setting low trigger prices through their own market impact.

S-3: One-Time Lock Pattern Across All Contracts

Every contract uses the same immutable lock pattern. After lockVault(), the deployer has no further privileges. This is replicated in PB, PBc, PBt, PBr, PBi — all become fully autonomous.

S-4: Consistent Reentrancy Protection

All state-changing external functions use OpenZeppelin ReentrancyGuard. Internal functions inherit protection from their callers. No unprotected entry point was found.

S-5: Non-Transferable Tokens Prevent Secondary Market Manipulation

PBt, PBr, PBi, and PresaleIOU all override transfer functions to revert. PBc restricts transfers to vault-mediated flows. This prevents selling positions to avoid unlock obligations, trading badges to hijack recovery/inheritance, and speculative secondary markets on locked positions.

S-6: Sequential Netting with Live Sanitization

The netting engine sanitizes caller-provided unlock hints against live on-chain state: removes dead positions, deduplicates IDs, sorts by trigger price (ascending), and caps at MAX_UNLOCK_PER_BUY (500). This prevents duplicate settlements, stale hint exploitation, and gas-griefing beyond the cap.

S-7: Exact-Amount Approvals

All approve() calls use the exact amount needed for the current operation, never type(uint256).max. This minimizes exposure if the PulseX router were compromised.

S-8: LP Fee Extraction via K-Tracking Preserves Principal

The K-per-LP-token mechanism ensures only trading fee growth is extracted. LP principal is never withdrawn. The sqrt(reserve0 * reserve1) per LP token only increases from trading fees; proportional LP minting/burning doesn't change it.

Deployment Sequence — Verified

The one-time lock pattern creates a critical dependency chain during deployment. The deployment scripts (MAINdep.js through MAINdep7-12.js) handle this with comprehensive safety:

```
Phase 1-6:   MAINdep.js      — Deploy tokens, pair, PulseXInterface, Vault, lockVault + mint
Phase 7:     MAINdep7.js     — Close presale + transfer USDL to Vault
Phase 8:     MAINdep8.js     — Deploy LaunchConverter
Phase 8.5:   MAINdep8.5.js   — Deploy PBRemoveUserLP + VaultViews
Phase 9:     MAINdep9.js     — lockImmutableReferences (PERMANENT)
Phase 10:    MAINdep10.js    — Seed LP -> trading goes live
Phase 11:    MAINdep11.js    — setLaunchConverter on PresaleIOU (POINT OF NO RETURN)
Phase 12:    MAINdep12.js    — Execute IOU->PBt conversion batches
```

Safety Properties Verified in the Scripts

- **Network gate: Every script verifies pulsemainnet + chainId 369; throws on mismatch**
- **Phase ordering: Each script enforces minimum previous phase via verifyPhase() / state checks**
- **On-chain verification: verifyOnchainState() runs at the start of each phase — checks deployed bytecode, vault-lock status, pair reserves, price feed config, and cross-references against on-chain factory data**
- **Interactive confirmations: Critical operations require typing explicit keywords (OK, CLOSE, LOCK, LINK, DEPLOY) — not just Enter**
- **Deployer consistency: State file tracks deployer address; re-runs verify the same deployer is signing**
- **Post-action validation: Every lockVault, setVault, lockImmutableReferences call is followed by an on-chain read-back that throws if the result does not match**
- **Address cross-validation: validatePairAddress() re-derives the pair from PulseX factory and compares against state; validateCriticalAddresses() checks for zero-address or missing entries**
- **Resume support: All scripts detect already-completed phases and skip them; state file tracks phase number persistently**
- **Abort-and-redeploy window: Everything before Phase 9 (lockImmutableReferences) can be abandoned — deploy fresh contracts and restart**
- **Batch conversion resilience: MAINdep12 retries failed batches (up to 3 retries with 10s backoff), saves progress every 5 batches, and picks up from the last unconverted batch on re-run**

> *Verdict: The deployment sequence is well-engineered. The scripts eliminate the risk of out-of-order execution or wrong-address locking through enforced phase gates, on-chain verification, and explicit confirmation prompts.*

Conclusion

The Perpetual Bitcoin protocol demonstrates a strong security posture through its immutable, admin-free architecture. The PB.sol pair guard is a particularly effective defense against the most common DeFi attack vectors.

- **One deployment blocker existed (H-1: undefined _escapeJsonString in PBr/PBi) — now RESOLVED by removing the dead line from both contracts.**
- **Two medium-severity findings exist but are substantially mitigated: password mempool visibility requires the attacker to already control the designated recovery address (structurally non-exploitable by third parties), and the unchecked PresaleIOU transfer concerns an already-deployed mainnet contract using a USDL token that reverts on failure.**
- **No critical exploit path was identified in the live protocol flows (buying, netting, unlocking, recovery, inheritance, VLock, LP harvesting, LP removal).**

> *With H-1 resolved, the protocol is ready for deployment from a code-level security perspective.*

— End of Review —