Critical Issues Verification Report

Norah Token Ecosystem Smart Contracts

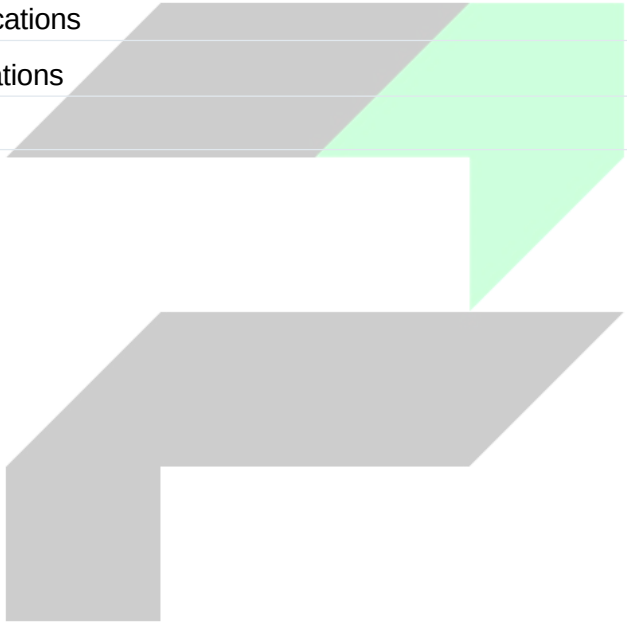
VERIFICATION STATUS

ALL CRITICAL ISSUES RESOLVED

Date:	December 22, 2025
Report Version:	1.0
Scope:	Critical Issues (C-01 through C-05)

Table of Contents

1. Executive Summary 3
2. Verification Methodology 3
3. Issue-by-Issue Verification 4
    3.1 [C-01] Broken Bridging Logic in BridgeApplication 4
    3.2 [C-02] Transaction Mapping Collision in BridgeReserve 5
    3.3 [C-03] Double-Claiming of Rewards 6
    3.4 [C-04] Price Consensus Manipulation 7
    3.5 [C-05] Broken Daily Limit Logic (Solana) 8
4. Summary of Verifications 9
5. Additional Observations 9
6. Conclusion 10



1. Executive Summary

This report verifies that all five critical vulnerabilities identified in the original audit report have been addressed in the current codebase. Each critical issue has been examined against the actual contract implementations to confirm proper remediation.

Issue ID	Description	Status
C-01	Broken Bridging Logic in BridgeApplication	FIXED
C-02	Transaction Mapping Collision in BridgeReserve	FIXED
C-03	Double-Claiming of Rewards in QuarterlyRevenueObligation	FIXED
C-04	Fake Price Consensus in OracleIntegration	FIXED
C-05	Broken Daily Limit Logic in NorahToken (Solana)	FIXED

2. Verification Methodology

For each critical issue, the following verification process was applied:

1. Review of the original vulnerability description and impact
2. Examination of the recommended fix
3. Code analysis of the current implementation
4. Verification that the fix properly addresses the vulnerability
5. Assessment of any remaining risks or edge cases

3. Issue-by-Issue Verification

3.1 [C-01] Broken Bridging Logic in BridgeApplication ✓ FIXED

Location: `contracts/BridgeApplication.sol`

Original Issue:

- The processBridge function conflated incoming and outgoing bridge directionality
- Incoming bridge transactions were validated against bridgeRequests mapping (stores outgoing requests)
- This created circular logic where bridging tokens effectively burned and re-minted them on the same chain

Recommended Fix:

- Separate logic for incoming and outgoing bridges
- Create a new executeIncomingBridge function that does not check bridgeRequests

Verification Findings:

- **Separate Functions Implemented:** Lines 192-237: processOutgoingBridge() and Lines 239-264: executeIncomingBridge()
- **Incoming Bridge Logic:** executeIncomingBridge() does NOT check bridgeRequests mapping
- **Outgoing Bridge Logic:** processOutgoingBridge() correctly validates against bridgeRequests mapping
- **Transaction Tracking:** Both functions use processedTransactions mapping with different keys

Code Evidence:

```
// Line 239-264: executeIncomingBridge - Does NOT check bridgeRequests function
executeIncomingBridge( address user, uint256 amount, uint256 fromChainId, string memory txHash )
external onlyValidator nonReentrant whenNotPaused { // ... validations ... // No bridgeRequests
check - directly processes incoming bridge mintManager.createBridgeMintRequest(user, amount,
fromChainId, txHash); }
```

Verdict: FIXED — The bridging logic has been properly separated. Incoming bridges no longer require matching outgoing requests.

3.2 [C-02] Transaction Mapping Collision in BridgeReserve FIXED

Location: `contracts/BridgeReserve.sol`

Original Issue:

- Contract used a global bridgeTransactions mapping keyed by nonce only
- Nonces were generated per-user via userNonces[msg.sender]++
- Multiple users would generate identical nonces causing data overwrites

Recommended Fix:

- Use keccak256(abi.encodePacked(msg.sender, nonce)) as the key, OR

- Implement a global nonce counter, OR
- Use nested mappings

Verification Findings:

- **Nested Mapping Implemented:** mapping(address => mapping(uint256 => BridgeTransaction))
- **Per-User Nonce Tracking:** Each user maintains their own nonce counter
- **Storage Access Pattern:** All accesses use nested structure [user][nonce]

Code Evidence:

```
// Line 39: Nested mapping prevents collisions mapping(address => mapping(uint256 =>
BridgeTransaction)) public bridgeTransactions; // Line 134: Storage uses nested structure
bridgeTransactions[msg.sender][nonce] = BridgeTransaction({ user: msg.sender, amount: amount, //
... });
```

Verdict: FIXED — Nested mapping structure ensures each user's transactions are stored in separate nonce spaces.

3.3 [C-03] Double-Claiming of Rewards in QuarterlyRevenueObligation FIXED

Location: `contracts/QuarterlyRevenueObligation.sol`

Original Issue:

- claimDistribution function determined user share based on current token balance at claim time
- Users could claim rewards, transfer tokens to another account, and claim again
- This allowed complete fund drainage through repeated claims

Recommended Fix:

- Implement historical balance snapshots using ERC20Votes.getPastVotes or ERC20Snapshot
- Record snapshot block in calculateDistribution and use it in claimDistribution

Verification Findings:

- **Snapshot Block Tracking:** distributionSnapshotBlock mapping records block numbers
- **Snapshot Creation:** Snapshot taken when distribution is calculated, not claimed
- **Historical Balance Lookup:** Uses norahToken.getPastVotes(msg.sender, snapshotBlock)
- **Token Contract Support:** NorahToken extends ERC20Votes providing getPastVotes()

Code Evidence:

```
// Line 260-262: Snapshot taken during calculation distributionSnapshotBlock[quarter] =
block.number; distributionSnapshotTimestamp[quarter] = block.timestamp;
distributionSnapshotTaken[quarter] = true; // Line 300: Historical balance lookup uint256
historicalBalance = norahToken.getPastVotes(msg.sender, snapshotBlock);
```

Verdict: FIXED — Historical snapshots using ERC20Votes prevent double-claiming attacks.

3.4 [C-04] Fake Price Consensus in OracleIntegration FIXED

Location: `contracts/OracleIntegration.sol`

Original Issue:

- Contract stored only one price entry per symbol globally
- calculateConsensusPrice iterated over oracles but read the same global value
- System returned the last submitted price, enabling single-oracle manipulation

Recommended Fix:

- Change storage to mapping(address => mapping(string => PriceData))
- Aggregate prices properly in consensus calculation

Verification Findings:

- **Per-Oracle Storage:** mapping(address => mapping(string => PriceData))
- **Price Update Logic:** Prices stored per oracle using msg.sender as key

- **Consensus Calculation:** Iterates oracles, reads per-oracle prices, calculates true average
- **Validation Logic:** Validates each oracle's data before inclusion

Code Evidence:

```
// Line 42: Per-oracle storage mapping(address => mapping(string => PriceData)) public
oraclePrices; // Line 324: Reading per-oracle prices in consensus PriceData memory data =
oraclePrices[oracle][symbol]; // Line 338: Calculating true consensus int256 consensusPrice =
totalPrice / int256(validOracles);
```

> **Verdict:** FIXED — Per-oracle storage enables proper consensus calculation resistant to single-oracle manipulation.

3.5 [C-05] Broken Daily Limit Logic in NorahToken (Solana) FIXED

Location: `solana-programs/norah-token/src/lib.rs`

Original Issue:

- Daily limit check used raw Unix timestamp (seconds) instead of day index
- Since timestamps update every second, limit reset per block instead of per day
- A compromised robot authority could mint up to daily_limit per block

Recommended Fix:

- Calculate day index: let current_day = Clock::get()?.unix_timestamp / 86400;

Verification Findings:

- **Day Index Calculation:** const SECONDS_PER_DAY: i64 = 86400; current_day = current_time / SECONDS_PER_DAY
- **Daily Limit Check:** Compares day indices, not raw timestamps
- **Daily Limit Reset:** Resets only when last_mint_day != current_day
- **Consistent Implementation:** Both robot_mint and execute_mint use same calculation

Code Evidence:

```
// Line 52-54: Day index calculation const SECONDS_PER_DAY: i64 = 86400; let current_day = current_time / SECONDS_PER_DAY; let last_mint_day = ctx.accounts.authority.last_mint_date / SECONDS_PER_DAY; // Line 85-88: Reset only on day change if last_mint_day != current_day { authority.daily_minted = 0; authority.last_mint_date = current_time; }
```

Verdict: FIXED — Daily limit logic now correctly uses day indices, preventing unlimited minting within a single day.

4. Summary of Verifications

Issue ID	Status	Verification Result
C-01	FIXED	Bridging logic properly separated into incoming/outgoing functions
C-02	FIXED	Nested mapping prevents transaction collision
C-03	FIXED	Historical snapshots prevent double-claiming
C-04	FIXED	Per-oracle storage enables proper consensus calculation
C-05	FIXED	Day index calculation enforces true daily limits

5. Additional Observations

Positive Findings:

- **Code Quality:** All fixes are implemented cleanly and follow recommended patterns
- **Backward Compatibility:** OracleIntegration maintains backward compatibility while fixing the consensus issue
- **Comprehensive Fixes:** Each fix addresses the root cause, not just symptoms
- **Documentation:** Code comments indicate awareness of the fixes (e.g., "FIX C-05" in Solana code)

6. Conclusion

FINAL VERDICT

ALL CRITICAL ISSUES VERIFIED AS FIXED

All five critical vulnerabilities have been successfully resolved. The implementations follow the recommended fixes and address the root causes of each issue. The codebase demonstrates proper understanding of the vulnerabilities and appropriate remediation strategies.