// Security Assessment

11.17.2025 - 12.08.2025

Strong DAO Smart Contracts

Buck

HALBORN

Strong DAO Smart Contracts - Buck

Prepared by: HALBORN

Last Updated 12/12/2025

Date of Engagement: November 17th, 2025 - December 8th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
12	1	0	0	3	8

TABLE OF CONTENTS

1. Introduction

Buck DAO engaged Halborn to conduct a security assessment on their smart contracts beginning on November 17th, 2025 and ending on December 8th, 2025. The security assessment was scoped to the smart contracts provided in the strong-smart-contracts-internal Github repository, provided to the Halborn team. Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided with 16 days for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possess deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the Strong protocol smart contracts.
- Ensure that smart contract of Strong protocol functions operate as intended.

In summary, Halborn identified several areas for improvement to reduce the likelihood and impact of security risks, which were mostly addressed by the Buck protocol team. The main recommendations were:

- Redesign the reward distribution system to ensure that the total rewards paid out to all users never exceeds the value of funds distributed to the protocol.
- Ensure that users who hold tokens for the same amount of time receive the same rewards, regardless of how frequently they interact with the system.
- Prevent the system from accepting outdated or unreliable price data by enforcing minimum freshness and quality thresholds that cannot be accidentally disabled.
- Require that collateral valuations always move forward in time and cannot be overwritten with older data points.

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture and purpose of the `Buck` protocol.
- Manual code review and walkthrough of the `Buck` in-scope contracts.
- Manual assessment of critical Solidity variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis and fuzzing of security for scoped contracts and imported functions (Slither & Medusa).
- Local deployment and testing with Foundry

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: strong-smart-contracts-internal

(b) Assessed Commit ID: 2cfcd5f

(c) Items in scope:

- src/collateral/CollateralAttestation.sol
- src/kyc/KYCRegistry.sol
- src/liquidity/LiquidityReserve.sol
- src/liquidity/LiquidityWindow.sol
- src/oracle/OracleAdapter.sol
- src/policy/PolicyManager.sol
- src/rewards/RewardsEngine.sol
- src/token/STRX.sol

Out-of-Scope: Third-party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- b3f877f
- 700b7db
- a34eed2
- 9d0bda8
- 56e3fc1
- a7e0e5a

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	3	8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PHANTOM UNIT ACCOUNTING CAUSES OVER-MINTING AND REWARD MISALLOCATION	CRITICAL	SOLVED - 12/10/2025
REVOKED KYC USERS CAN IMMEDIATELY RE-REGISTER WITH SAME MERKLE PROOF	LOW	SOLVED - 12/10/2025
MULTIPLE UNUSED STATE VARIABLES AND CONSTANTS ACROSS CONTRACTS	LOW	SOLVED - 12/10/2025
VIEW FUNCTION MISMATCH WITH CLAIM BEHAVIOR	LOW	SOLVED - 12/10/2025
SYSTEM CONTRACTS NOT EXCLUDED FROM REWARDS DURING DEPLOYMENT	INFORMATIONAL	ACKNOWLEDGED - 12/10/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MULTIPLE SAME-TIMESTAMP DISTRIBUTIONS CAUSE REWARD LOSS	INFORMATIONAL	SOLVED - 12/10/2025
MISSING EPOCH VALIDATION ALLOWS PAST AND OVERLAPPING EPOCHS	INFORMATIONAL	ACKNOWLEDGED - 12/10/2025
ORACLEADAPTER ALLOWS DISABLED STALENESS AND CONFIDENCE CHECKS	INFORMATIONAL	ACKNOWLEDGED - 12/10/2025
COLLATERALATTESTATION ALLOWS BACKWARD TIMESTAMP IN ATTESTATIONS	INFORMATIONAL	SOLVED - 12/10/2025
MISSING RELATIONSHIP VALIDATION BETWEEN STALENESS THRESHOLDS	INFORMATIONAL	SOLVED - 12/10/2025
MISSING DELAY ORDERING VALIDATION IN TIER CONFIGURATION	INFORMATIONAL	SOLVED - 12/10/2025
TIERED WITHDRAWAL SECURITY MODEL CAN BE BYPASSED VIA SPLITTING	INFORMATIONAL	ACKNOWLEDGED - 12/10/2025

7. FINDINGS & TECH DETAILS

7.1 PHANTOM UNIT ACCOUNTING CAUSES OVER-MINTING AND REWARD MISALLOCATION

// CRITICAL

Description

The reward calculation logic contains a fundamental accounting asymmetry between global unit settlement and per-account unit settlement. This single root cause manifests as multiple observable symptoms: protocol-level over-minting of STRX tokens, systematic wealth transfer from passive to active holders, exploitable extraction by late joiners, and amplification through Sybil patterns.

The `globalSettle()` function accumulates balance-time units across all epochs without boundary restrictions, while `settle()` caps individual account accrual to the current epoch. This creates phantom units, globally counted but individually unrealizable, which corrupt reward calculations and cause the protocol to mint more STRX than the value of USDC coupons distributed.

Root cause explanation

The reward system uses a "units" model where units represent balance × time. The intended invariant is:

```
sum(all_account_units) == global_units
```

However, this invariant is violated:

Step 1: Global settlement spans epochs

When `_globalSettle()` is called, it calculates units for the entire time period since the last settlement:

```
1032 | uint256 elapsed = uint256(effectiveEnd - lastSettlement);
1033 | uint256 globalUnits = elapsed * totalSupply;
```

If `lastSettlement` was in epoch N-2 and we're now in epoch N, this captures units across multiple epochs.

Step 2: Per-account settlement is epoch-bounded

When `_settle()` is called for, it explicitly caps the accrual window:

```
1142 uint64 accrualStart = state.lastAccrual;
1143
1144 // Apply epoch start boundary - can't accrue before current epoch started
1145 uint64 epochStart_ = epochStart;
1146 if (epochStart_ != 0 && accrualStart < epochStart_) {
1147     accrualStart = epochStart_;  // CAPS to current epoch start
1148 }
```

Step 3: The gap becomes phantom units

The difference between global units and realizable account units accumulates as `phantomLockedUnits`. These phantom units:

- Are included in the denominator when calculating `accRewardPerUnit`
- Are never fully realized by the accounts they were computed from
- Create "orphaned" reward capacity that active claimers can capture

Code Location

Global settlement (unbounded time):

```
988  function _globalSettle() internal {
989      uint64 currentTime = uint64(block.timestamp);
990      uint64 lastSettlement = lastGlobalSettlementTime;
991
992      // Uses unbounded time since last settlement
993      uint256 elapsed = uint256(effectiveEnd - lastSettlement);
994      uint256 globalUnits = elapsed * totalSupply;
995
996      // Adds ALL units to phantom pool
997      if (globalUnits > 0) {
998          settledLockedUnits += globalUnits;
999          phantomLockedUnits += globalUnits;  // These become "orphaned"
1000     }
1001 }
```

Per-account settlement (epoch-bounded):

```
901  function _settle(AccountState storage state, address account) internal {
902      uint64 accrualStart = state.lastAccrual;
903
```

```
904         // CRITICAL: Caps accrual to current epoch boundaries
905         if (epochStart != 0 && accrualStart < epochStart) {
906             accrualStart = epochStart;  // Loses all pre-epoch time
907         }
908 
909         uint64 accrualEnd = uint64(block.timestamp);
910         if (epochEnd != 0 && accrualEnd > epochEnd) {
911             accrualEnd = epochEnd;  // Caps at epoch end
912         }
913 
914         // Only accrues within the capped window
915         uint256 elapsed = uint256(accrualEnd - accrualStart);
916         uint256 newUnits = state.balance * elapsed;
917     }
```

This asymmetry creates "phantom units" units that exist in the global denominator used to calculate `accRewardPerUnit`, but which are never realized by individual accounts. The phantom units corrupt the reward distribution math, causing:

1. **Over-minting**: Total STRX claimed exceeds USDC distributed
2. **Active holder advantage**: Frequent claimers capture phantom unit rewards
3. **Passive holder loss**: Infrequent claimers lose cross-epoch rewards
4. **Late joiner extraction**: New entrants inherit phantom unit rewards
5. **Sybil amplification**: Multi-address strategies compound the effects
6. **Transfer-based manipulation**: Mid-epoch transfers misallocate rewards between sender/receiver

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import {BaseTest} from "test/utils/BaseTest.sol";
import {RewardsEngine} from "src/rewards/RewardsEngine.sol";
import {STRX} from "src/token/STRX.sol";
import {PolicyManager} from "src/policy/PolicyManager.sol";
import {OracleAdapter} from "src/oracle/OracleAdapter.sol";
import {LiquidityReserve} from "src/liquidity/LiquidityReserve.sol";
import {MockUSDC} from "src/mocks/MockUSDC.sol";
import {console} from "forge-std/Test.sol";

/**
 * @title PhantomUnitsOverMintPOC
 * @notice Minimal proof-of-concept demonstrating the phantom units over-minting bug
 *
 * BUG SUMMARY:
 * - Active claimers extract more than their fair share of rewards
```

```
 * - Passive holders lose rewards to active claimers
 * - Total STRX minted exceeds the USDC value distributed (over-minting)
 *
 * ROOT CAUSE:
 * - _globalSettle() accumulates units for ALL holders across epochs
 * - _settle() caps individual accrual to the CURRENT epoch only
 * - The difference becomes "phantom units" - counted globally but never claimed
 * - Active claimers capture rewards allocated to phantom units
 */
contract PhantomUnitsOverMintPOC is BaseTest {
    RewardsEngine public rewards;
    STRX public strx;
    PolicyManager public policy;
    OracleAdapter public oracle;
    LiquidityReserve public reserve;
    MockUSDC public usdc;

    address constant TIMELOCK = address(0x1000);
    address constant TREASURY = address(0x2000);
    address constant LIQUIDITY_WINDOW = address(0x3000);
    address constant DISTRIBUTOR = address(0x4000);
    address constant ALICE = address(0xA1);
    address constant BOB = address(0xB0);

    uint256 constant EPOCH_30_DAYS = 30 days;

    function setUp() public {
        usdc = new MockUSDC();
        vm.startPrank(TIMELOCK);

        strx = deploySTRX(TIMELOCK);
        policy = deployPolicyManager(TIMELOCK);
        oracle = new OracleAdapter(TIMELOCK);
        reserve = deployLiquidityReserve(TIMELOCK, address(usdc), address(0), TREASURY);

        // Production parameters: 30-min anti-snipe, 1 STRX min claim
        rewards = deployRewardsEngine(TIMELOCK, DISTRIBUTOR, 1800, 1e18, false);

        strx.configureModules(
            LIQUIDITY_WINDOW, address(reserve), TREASURY, address(policy), address(0), address(rewards)
        );

        rewards.setToken(address(strx));
        rewards.setPolicyManager(address(policy));
        rewards.setReserveAddresses(address(reserve), address(usdc));
        rewards.setMaxTokensToMintPerEpoch(type(uint256).max);

        // CAP price = $1.00 (1 STRX = 1 USDC)
        oracle.setFallbackPrice(1e18);
        policy.setDistributionOracle(address(oracle));

        // No skim for clean math
        PolicyManager.BandConfig memory config = policy.getBandConfig(PolicyManager.Band.Green);
        config.distributionSkimBps = 0;
        policy.setBandConfig(PolicyManager.Band.Green, config);
```

```
        policy.reportSystemSnapshot(
            PolicyManager.SystemSnapshot({
                reserveRatioBps: 10000,
                equityBufferBps: 500,
                oracleStaleSeconds: 0,
                totalSupply: 0,
                navPerToken: 1e18,
                reserveBalance: 10_000_000e6,
                collateralRatio: 1e18
            })
        );

        vm.stopPrank();

        // Fund distributor
        usdc.mint(DISTRIBUTOR, 10_000_000e6);
        vm.prank(DISTRIBUTOR);
        usdc.approve(address(rewards), type(uint256).max);
    }

    /**
     * @notice Demonstrates over-minting: total claimed STRX > total distributed USDC value
     *
     * Scenario:
     * - Alice and Bob each hold 500,000 STRX (50% each)
     * - 4 monthly epochs, $100,000 USDC distributed per epoch ($400,000 total)
     * - Alice claims every epoch (active), Bob claims once at end (passive)
     *
     * Expected: Each holder receives $200,000 worth of STRX (400,000 STRX total)
     * Actual: Total minted exceeds 400,000 STRX (over-minting)
     */
    function test_OverMinting_PhantomUnits() public {
        uint64 baseTime = uint64(block.timestamp);

        // Both holders start with equal 500K STRX
        vm.startPrank(LIQUIDITY_WINDOW);
        strx.mint(ALICE, 500_000e18);
        strx.mint(BOB, 500_000e18);
        vm.stopPrank();

        uint256 aliceTotal = 0;
        uint256 totalDistributed = 0;

        // 4 monthly epochs
        for (uint64 epoch = 1; epoch <= 4; epoch++) {
            uint64 epochStart = baseTime + (epoch - 1) * uint64(EPOCH_30_DAYS);
            uint64 epochEnd = epochStart + uint64(EPOCH_30_DAYS);

            vm.prank(TIMELOCK);
            rewards.configureEpoch(epoch, epochStart, epochEnd);

            vm.warp(epochStart + 15 days); // Mid-epoch

            // Distribute $100,000 USDC
            vm.prank(DISTRIBUTOR);
            rewards.distribute(100_000e6);
```

```
            totalDistributed += 100_000e6;

            // Alice claims every epoch
            vm.prank(ALICE);
            aliceTotal += rewards.claim(ALICE);
        }

        // Bob claims once at end
        vm.prank(BOB);
        uint256 bobTotal = rewards.claim(BOB);

        // Results
        uint256 totalClaimed = aliceTotal + bobTotal;
        uint256 expectedTotal = totalDistributed * 1e12; // At $1 CAP, 1 USDC = 1 STRX

        console.log("=== OVER-MINTING BUG PROOF ===");
        console.log("Total USDC distributed: $%d", totalDistributed / 1e6);
        console.log("Expected STRX minted:   %d", expectedTotal / 1e18);
        console.log("Actual STRX minted:     %d", totalClaimed / 1e18);
        console.log("");
        console.log("Alice (active):  %d STRX", aliceTotal / 1e18);
        console.log("Bob (passive):   %d STRX", bobTotal / 1e18);

        uint256 overMint = totalClaimed - expectedTotal;
        uint256 overMintPct = (overMint * 100) / expectedTotal;
        console.log("");
        console.log("OVER-MINTED: %d STRX (+%d%%)", overMint / 1e18, overMintPct);

        // Assert over-minting occurred
        assertGt(totalClaimed, expectedTotal, "BUG: More STRX minted than USDC distributed");
    }
}
```

POC test output:

```
Ran 1 test for test/rewards/PhantomUnitsOverMintPOC.t.sol:PhantomUnitsOverMintPOC
[PASS] test_OverMinting_PhantomUnits() (gas: 913584)
Logs:
  === OVER-MINTING BUG PROOF ===
  Total USDC distributed: $400000
  Expected STRX minted:   400000
  Actual STRX minted:     542475

  Alice (active):  426601 STRX
  Bob (passive):   115873 STRX

  OVER-MINTED: 142475 STRX (+35%)

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.71ms (3.31ms CPU time)

Ran 1 test suite in 365.56ms (13.71ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

The test demonstrates that with $400,000 USDC distributed across 4 epochs, the system mints 542,475 STRX instead of the expected 400,000 STRX, a 35% over-mint. This proves the phantom units accounting bug where active users (Alice) can extract significantly more rewards than the protocol received.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:H (9.4)

Recommendation

Consider replacing units with a monotonically increasing `globalRewardIndex`. Each account stores `userRewardIndex` updated on claim. `Rewards = balance × (globalIndex − userIndex)`. This makes claim frequency irrelevant to total entitlement.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by implementing a comprehensive time-weighted unit accounting system with late-entry protection and proportional forfeiture rules.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/b3f877f74f74ca26022b3b8a95c18720def3b120

7.2 REVOKED KYC USERS CAN IMMEDIATELY RE-REGISTER WITH SAME MERKLE PROOF

// LOW

Description

The KYC registry's revocation mechanism only toggles an "allowed" flag and does not maintain a persistent revocation state. A revoked address can immediately call `registerWithProof()` again using the same Merkle proof and epoch, effectively undoing its own revocation in the same block if it races or bundles transactions.

The `revoke()` helper sets `allowed[account]` to false but does not write to any blacklist or "revoked" mapping. The registration function checks only that `!_allowed[msg.sender]` and that the Merkle proof is valid for the current root and epoch. Since revocation does not change the root or epoch and only flips `_allowed` to false, the revoked user satisfies the preconditions for registration immediately after revocation and can re-enter with the same proof.

During incident response or compliance actions, operators may believe they have effectively removed a user from the allowed set while that user can immediately regain access, undermining the effectiveness of revocation and potentially enabling continued misuse until a new Merkle root is computed and published.

Code Location

In `KYCRegistry.sol` : Revocation only clears the `allowed` flag, so a revoked address can immediately satisfy the preconditions for `registerWithProof` using the same proof.

```
125 function _revoke(address account) internal {
126     if (_allowed[account]) {
127         _allowed[account] = false;
128         emit KYCUpdated(account, false, currentEpoch);
129     }
130 }
131
132 function registerWithProof(bytes32[] calldata proof, uint64 epoch) external whenNotPaused {
133     require(merkleRoot != bytes32(0), "KYCRegistry: root not set");
134     require(epoch == currentEpoch, "KYCRegistry: stale epoch");
135     require(!_allowed[msg.sender], "KYCRegistry: already allowed");
136
137     bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
138     // verify proof against current merkleRoot...
139 }
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:M/I:M/D:N/Y:N (3.1)

Recommendation

Consider introducing a persistent revocation mapping that is checked in `registerWithProof()`, with a separate administrative path (for example, via `forceAllow()`) to clear revocation when needed so that revoked accounts cannot silently re-register using old proofs.

Remediation Comment

SOLVED: The **Buck team** solved this issue by introducing a persistent denylist mapping that blocks revoked users from re-registering.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/700b7dbdc4ca0475b6077a677408f0874838c2f5

7.3 MULTIPLE UNUSED STATE VARIABLES AND CONSTANTS ACROSS CONTRACTS

// LOW

Description

Several state variables and constants are declared and configured but never read or used in the codebase. This represents dead code that may mislead administrators, and indicates incomplete feature implementation.

Code Location

1. LiquidityWindow.sol - `maxOracleStaleness` (state variable)
- Declared at line 125, set at line 260
- The variable is set via `configureOracle()` but never read for any logic
- The only code calling `OracleAdapter.isHealthy()` is in `PolicyManager.sol`, which uses its own hardcoded constants `STRESSED_ORACLE_STALENESS`, `HEALTHY_ORACLE_STALENESS`)

2. PolicyManager.sol - `distributionOracle` (state variable)
- Declared at line 164, set at line 376
- The variable is set via `setDistributionOracle()` but never queried anywhere in the codebase

3. PolicyManager.sol - `HOURS_PER_DAY` (constant)
- Declared at line 56
- Declared but never used anywhere in the contract

4. RewardsEngine.sol - `lastConfirmedReserveBalance` (state variable)
- Declared at line 152, set at lines 281-282
- The variable is only used for event emission in `adjustReserveBaseline()`, never for any logic

5. STRX.sol - `MAX_SWAP_FEE_BPS` (constant)
- Declared at line 55
- Declared but never used - swap fees are not validated against this cap

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Remove unused code or implement the intended functionality:

1. `maxOracleStaleness` : Either remove it from LiquidityWindow or use it in oracle health checks
2. `distributionOracle` : Either remove or implement the distribution oracle logic it was intended for
3. `HOURS_PER_DAY` : Remove unused constant
4. `lastConfirmedReserveBalance` : Either remove or use for reserve balance validation
5. `MAX_SWAP_FEE_BPS` : Either remove or add fee validation: `require(feeBps <= MAX_SWAP_FEE_BPS)`

Remediation Comment

SOLVED: The **Buck team** solved the issue in the specified commit by removing unused state variables and constants from the codebase, along with their associated setter functions and events.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/a34eed24e32c455a18e4b4600e5fd6af7e82f343

7.4 VIEW FUNCTION MISMATCH WITH CLAIM BEHAVIOR

// LOW

Description

The `getAccountFullState()` view function can return claimable amounts that differ from what `claim()` will actually pay out, causing user confusion and integration issues.

The view function uses a different code path than the claim function

- `getAccountFullState` function uses: `_calculatePendingUnits()`, `_splitPendingUnits()`, `_shouldUnlockUnits()`
- `Claim` uses: `_settle()`, `_unlockAccountUnits()`, `_addUnits()`

These paths handle edge cases differently, particularly around:

- Passive holder unit splitting at `lastGlobalSettlement`
- Phantom unit realization timing
- Epoch boundary handling

Code Location

In the `RewardsEngine::getAccountFullState` function:

```
820 | // View path (getAccountFullState):
821 | if (lastGlobalSettlement > 0 && distributionCount > 0 && state.lastAccrual < lastGlobalSettlement) {
822 |     (pendingUnlocked, pendingLocked) = _splitPendingUnits(state, account, lastGlobalSettlement);
823 | } else {
824 |     pendingLocked = _calculatePendingUnits(state, account);
825 | }
826 |
827 | // Claim path (_settle) uses different logic with epoch capping
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Refactor to use a single shared settlement calculation for both view and mutation paths, ensuring consistent results. Add property-based tests asserting `getAccountFullState(account).claimable == claim(account)` across all edge cases.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by completely rewriting the RewardsEngine contract with a unified "Epoch-Bound Synthetix Index" architecture that eliminates the view/claim divergence.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/b3f877f74f74ca26022b3b8a95c18720def3b120

7.5 SYSTEM CONTRACTS NOT EXCLUDED FROM REWARDS DURING DEPLOYMENT

// INFORMATIONAL

Description

System contracts such as the Treasury, LiquidityReserve, and DEX pair can hold STRX balances and participate in the rewards mechanism unless explicitly excluded. If these contracts are not excluded at deployment time, they may accrue a significant fraction of rewards, diluting payouts to end users and obscuring reward accounting.

The rewards system computes global reward accrual based on total STRX supply, and rewards are allocated proportionally to balance-time units. System contracts naturally accumulate non-trivial STRX balances due to fee flows and liquidity positions, but there is no automatic exclusion for these addresses. Unless deployment scripts or governance actions explicitly call `setAccountExcluded()` for each system contract, their balances count as eligible for rewards even though these addresses are not economic end users.

Code Location

In `RewardsEngine.sol` : System and user accounts share the same exclusion flag, so system contracts must be explicitly excluded to avoid accruing rewards.

```
359 function setAccountExcluded(address account, bool isExcluded) external onlyRole(ADMIN_ROLE) {
360     AccountState storage state = _accounts[account];
361     _settle(state, account);
362 
363     uint64 timestamp = uint64(block.timestamp);
364     state.excluded = isExcluded;
365 }
```

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:M (1.3)

Recommendation

Consider standardizing deployment and upgrade procedures to explicitly exclude all known system addresses from rewards, and optionally add defensive logic that auto-excludes contracts or specific roles that are never intended to receive user-facing rewards.

Remediation Comment

ACKNOWLEDGED: The Buck team acknowledged this finding regarding the auto-exclusion of system contracts from accruing rewards.

7.6 MULTIPLE SAME-TIMESTAMP DISTRIBUTIONS CAUSE REWARD LOSS

// INFORMATIONAL

Description

When `distribute()` is called multiple times with no time elapsed between calls, subsequent distributions are diverted entirely to the `dust` accumulator instead of being allocated to token holders.

The `distribute()` function calculates new units as `newUnits = settledUnlockedUnits – unitsAccounted`. When no time has passed since the last distribution, `settledUnlockedUnits` hasn't increased, so `newUnits == 0`. In this case, the entire reward amount (including the coupon and any existing dust) is assigned to `dust` rather than increasing `accRewardPerUnit`.

Code Location

In the `RewardEngine.sol::distribute` function:

```
603 uint256 totalReward = tokensFromCoupon + dust;
604 uint256 newUnits = settledUnlockedUnits - unitsAccounted;
605
606 if (newUnits == 0) {
607     dust = totalReward;  // Entire reward goes to dust
608     emit DistributionDeclared(currentEpochId, 0, accRewardPerUnit, dust, grossAPYBps, netAPYBps);
609     return (0, skimAmount);
610 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:M (1.3)

Recommendation

Consider adding a revert condition in the `distribute()` logic if `newUnits == 0 && totalReward > 0` to make the failure explicit.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by redesigning the RewardsEngine contract to enforce exactly one distribution per epoch, making the zero-time dust trap vulnerability impossible to trigger.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/b3f877f74f74ca26022b3b8a95c18720def3b120

7.7 MISSING EPOCH VALIDATION ALLOWS PAST AND OVERLAPPING EPOCHS

// INFORMATIONAL

Description

The `configureEpoch()` function in `RewardsEngine` contract does not validate that epoch timestamps are in the future or that epochs do not overlap with existing ones.

Code Location

In `RewardsEngine.configureEpoch()`

```
346 function configureEpoch(uint64 epochId, uint64 epochStart_, uint64 epochEnd_)
347     external
348     onlyRole(ADMIN_ROLE)
349 {
350     if (epochEnd_ <= epochStart_) revert InvalidConfig();
351     if (epochId <= currentEpochId) revert InvalidConfig();
352     currentEpochId = epochId;
353     epochStart = epochStart_;
354     epochEnd = epochEnd_;
355     emit EpochConfigured(epochId, epochStart_, epochEnd_);
356 }
```

The function only validates:

1. epochEnd > epochStart
2. epochId > currentEpochId

It does not validate:

1. That epochStart is in the future (or at least current time)
2. That epochEnd is in the future
3. That new epoch does not overlap with previous epoch

Administrators can configure epochs entirely in the past or epochs that overlap with existing ones. This could lead to:

- Unexpected reward distribution behavior

- Confusion in epoch accounting
- Edge cases in anti-snipe cutoff calculations

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Add validation to ensure `epochStart_` is in the future and does not overlap with the previous epoch's end time.

Remediation Comment

ACKNOWLEDGED: The **Buck team** acknowledged this finding, citing the issue as an operational risk.

7.8 ORACLEADAPTER ALLOWS DISABLED STALENESS AND CONFIDENCE CHECKS

// INFORMATIONAL

Description

The `OracleAdapter` contract accepts `staleAfter` and `maxConf` parameters of zero when configuring the Pyth oracle, which results in staleness and confidence checks being effectively disabled. In combination with Pyth's pull-based update model, this can cause the protocol to accept stale or highly uncertain prices without any explicit signal that protection has been turned off.

The `_tryPyth()` helper only performs staleness and confidence checks when `pythStaleAfter` and `pythMaxConf` are non-zero, and `configurePyth()` does not enforce non-zero or bounded values for these fields. If either parameter is configured as zero, whether by omission or design, the corresponding guardrail is skipped, and downstream code receives prices that may be far from current market consensus or based on highly uncertain feeds.

Code Location

In `OracleAdapter.sol` : Pyth configuration and read-path treat zero `staleAfter` / `maxConf` as "checks disabled," silently omitting staleness and confidence validation.

```
68  function configurePyth(address pyth, bytes32 priceId, uint256 staleAfter, uint256 maxConf)
69      external
70      onlyOwner
71  {
72      if (pyth == address(0) || priceId == bytes32(0)) revert ZeroAddress();
73      pythContract = pyth;
74      pythPriceId = priceId;
75      pythStaleAfter = staleAfter; // 0 disables staleness check
76      pythMaxConf = maxConf;       // 0 disables confidence check
77      emit PythConfigured(pyth, priceId, staleAfter, maxConf);
78  }
79
80  function _tryPyth() internal view returns (uint256 price, uint256 updatedAt) {
81      if (pythContract == address(0) || pythPriceId == bytes32(0)) return (0, 0);
82
83      IPyth.Price memory p = IPyth(pythContract).getPriceUnsafe(pythPriceId);
84      if (p.price <= 0) return (0, 0);
85      if (pythStaleAfter != 0 && block.timestamp > p.publishTime + pythStaleAfter) {
86          return (0, 0);
87      }
88      if (pythMaxConf != 0 && p.conf > 0) {
89          uint256 scaledConf = _scalePythConfidence(p);
90          if (scaledConf == 0 || scaledConf > pythMaxConf) {
91              return (0, 0);
```

```
        }
    }

    uint256 scaled = _scalePythPrice(p);
    if (scaled == 0) return (0, 0);
    return (scaled, p.publishTime);
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Consider enforcing reasonable non-zero ranges for `staleAfter` and `maxConf` in `configurePyth()`, and introducing explicit “bypass” or “emergency mode” flags if there is a legitimate need to temporarily relax these protections, so that disabling checks is deliberate, auditable, and cannot occur silently via zero values.

Remediation Comment

ACKNOWLEDGED: The **Buck team** acknowledged this finding regarding the OracleAdapter contract allowing disabled staleness and confidence checks.

7.9 COLLATERALATTESTATION ALLOWS BACKWARD TIMESTAMP IN ATTESTATIONS

// INFORMATIONAL

Description

The `publishAttestation()` function in `CollateralAttestation` ensures that attestations are not too old relative to the current block time, but does not enforce that the newly supplied attestation timestamp is greater than or equal to the previous measurement time. This allows attestations with older timestamps to overwrite newer ones, effectively permitting time reversal in the attested collateral state.

When an attestor submits a new valuation and timestamp, the contract validates that `block.timestamp − attestedTimestamp` is within a staleness window but does not compare `attestedTimestamp` against `attestationMeasurementTime`. As a result, an attestor can submit a valuation with a timestamp that is earlier than the last recorded measurement, so long as it is still within the global staleness bound relative to the current block.

Code Location

In `CollateralAttestation.sol` : The attestation path enforces a global staleness window but never compares the new measurement time against the prior one, permitting backward timestamps

```
191 function publishAttestation(uint256 _V, uint256 _HC, uint256 _attestedTimestamp)
192     external
193     onlyRole(ATTESTOR_ROLE)
194 {
195     if (_HC > PRECISION) revert InvalidHaircut();
196     if (_HC == 0) revert InvalidHaircut();
197
198     // Calculate new CR from provided parameters before storing
199     // Ensures staleness check uses the correct threshold (stressed vs healthy)
200     uint256 newCR = _calculateCR(_V, _HC);
201     uint256 maxStaleness = newCR >= PRECISION ? healthyStaleness : stressedStaleness;
202
203     if (block.timestamp - _attestedTimestamp > maxStaleness) {
204         revert StaleAttestationSubmission(_attestedTimestamp, block.timestamp, maxStaleness);
205     }
206
207     V = _V;
208     HC = _HC;
209     lastAttestationTime = block.timestamp;
210     attestationMeasurementTime = _attestedTimestamp;
211 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Consider enforcing that each new attestation timestamp is strictly greater than the previous `attestationMeasurementTime` and not in the future, so that the attested collateral timeline is strictly monotonic and cannot move backward in time.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by adding a monotonic timestamp check that requires each new attestation's measurement time to be strictly greater than the previous one.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/9d0bda87fffa3d06a76b6cf06ca18885e0d78aeb

7.10 MISSING RELATIONSHIP VALIDATION BETWEEN STALENESS THRESHOLDS

// INFORMATIONAL

Description

`CollateralAttestation` maintains separate staleness thresholds for healthy and stressed states but does not enforce any relationship between them during initialization or dynamic updates. This allows configurations where the stressed threshold permits older data than the healthy threshold, inverting the intended security model.

The contract defines `healthyStaleness` and `stressedStaleness` with comments indicating that the stressed threshold should be shorter (i.e., fresher data required when the system is undercollateralized). However the `setStalenessThresholds()` does not verify that `healthyStaleness` is greater than or equal to `stressedStaleness`. As a result, an administrator can accidentally configure `stressedStaleness` to be larger, allowing older data in precisely the scenarios where fresh data is most important.

Code Location

In `CollateralAttestation.sol` : Threshold-update logic requires non-zero values but does not enforce that the healthy threshold is at least as strict as the stressed threshold.

```
305 function setStalenessThresholds(uint256 _healthyStaleness, uint256 _stressedStaleness)
306     external
307     onlyRole(ADMIN_ROLE)
308 {
309     if (_healthyStaleness == 0) revert InvalidStalenessThreshold();
310     if (_stressedStaleness == 0) revert InvalidStalenessThreshold();
311
312     healthyStaleness = _healthyStaleness;
313     stressedStaleness = _stressedStaleness;
314
315     emit StalenessThresholdsUpdated(_healthyStaleness, _stressedStaleness);
316 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Consider enforcing an invariant that `healthyStaleness` is greater than or equal to `stressedStaleness` when thresholds are updated, reverting configurations that violate this relationship.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by adding a check-in `setStalenessThresholds` function that requires `stressedStaleness <= healthyStaleness`.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/56e3fc169599204fa769b8876adf770cb09aad27

7.11 MISSING DELAY ORDERING VALIDATION IN TIER CONFIGURATION

// INFORMATIONAL

Description

The `_validateTierConfig()` function in `LiquidityReserve` verifies that tier amount thresholds are ordered but does not verify that associated delay values are monotonically non-decreasing. This allows misconfigurations where large withdrawals are granted shorter (or zero) delays than small withdrawals, undermining the intent of the tiered withdrawal model.

Tier configurations specify both `maxAmountBps` and `delaySeconds` for immediate, delayed, and slow tiers. Validation currently ensures that amount thresholds are increasing and that the slow tier covers the full 100% range, but it does not enforce any ordering on delay durations. As a result, governance or administrative actions can configure tiers where, for example, the slow tier has a delay of zero seconds while the immediate tier has a non-zero delay.

Code Location

In `LiquidityReserve.sol` : Tier configuration validates amount thresholds but does not enforce any monotonic ordering on delay durations.

```
225 function configureTiers(
226     TierConfig calldata immediate,
227     TierConfig calldata delayed,
228     TierConfig calldata slow
229 ) external onlyRole(ADMIN_ROLE) {
230     _validateTierConfig(immediate, delayed, slow);
231     tierImmediate = immediate;
232     tierDelayed = delayed;
233     tierSlow = slow;
234 }
235 
236 function _validateTierConfig(
237     TierConfig calldata immediate,
238     TierConfig calldata delayed,
239     TierConfig calldata slow
240 ) internal pure {
241     if (immediate.maxAmountBps > delayed.maxAmountBps) revert InvalidConfig();
242     // additional checks on amountBps, but no validation of delaySeconds ordering
243 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Consider extending `_validateTierConfig()` to enforce that `delaySeconds` for the delayed tier is at least as large as for the immediate tier, and that `delaySeconds` for the slow tier is at least as large as for the delayed tier, rejecting any configuration that violates this ordering.

Remediation Comment

SOLVED: The **Buck team** solved this finding in the specified commit by adding two delay ordering checks to `_validateTierConfig` function that enforce `immediate.delaySeconds <= delayed.delaySeconds <= slow.delaySeconds`.

Remediation Hash

https://github.com/stronginc/strong-smart-contracts-internal/commit/a7e0e5a6ad0f83a183611391e3085710c5feaf48

7.12 TIERED WITHDRAWAL SECURITY MODEL CAN BE BYPASSED VIA SPLITTING

// INFORMATIONAL

Description

The tiered withdrawal system evaluates each withdrawal in isolation based on the current reserve balance and has no notion of cumulative withdrawals over a period. A compromised treasurer can drain a large fraction of the reserve by issuing many small withdrawals that each fall under the immediate tier threshold, effectively bypassing the intended delay for large withdrawals.

The `_determineTier()` function computes the percentage of a single withdrawal relative to the current reserve and maps it to immediate, delayed, or slow tiers with fixed delays. There is no tracking of how much has been withdrawn in the current block or over any rolling time window, nor any per-block or per-epoch aggregate limit. An attacker controlling the treasurer role can repeatedly request and execute small, immediate-tier withdrawals in rapid succession, exploiting the absence of cumulative enforcement.

Code Location

In `LiquidityReserve.sol` : Each withdrawal independently computes its tier based on the current balance share, with no awareness of prior withdrawals in the same block or window.

```
411 function _enqueueWithdrawal(address to, uint256 amount, address requestedBy) internal {
412     // CHECKS: Determine tier and delay based on amount
413     (uint8 tier, uint64 releaseAt) = _determineTier(amount);
414
415     // EFFECTS: Add withdrawal to queue
416     _withdrawals.push(
417         WithdrawalRequest({
418             to: to,
419             amount: amount,
420             releaseAt: releaseAt,
421             tier: tier,
422             requestedBy: requestedBy
423         })
424     );
425
426     // INTERACTIONS: Emit event (no external call, but follows pattern)
427     emit WithdrawalRequested(_withdrawals.length - 1, to, amount, releaseAt, tier, requestedBy);
428 }
429
430 function _determineTier(uint256 amount) internal view returns (uint8 tier, uint64 releaseAt) {
431     uint256 balance = asset.balanceOf(address(this));
432     uint256 amountBps =
433         balance > 0 ? Math.mulDiv(amount, BPS_DENOMINATOR, balance) : BPS_DENOMINATOR;
434
```

```
435     uint64 nowTs = uint64(block.timestamp);
436
437     if (amountBps <= tierImmediate.maxAmountBps) {
438         return (0, nowTs + tierImmediate.delaySeconds);
439     }
440     if (amountBps <= tierDelayed.maxAmountBps) {
441         return (1, nowTs + tierDelayed.delaySeconds);
442     }
443     return (2, nowTs + tierSlow.delaySeconds);
444 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Consider augmenting the tier logic with cumulative tracking (for example, total withdrawals per time window) or per-block caps, and using the cumulative amount when determining which tier's delay to apply, so that splitting a large withdrawal into many smaller pieces does not avoid the intended security delay.

Remediation Comment

ACKNOWLEDGED: The **Buck team** acknowledged this finding regarding the tiered withdrawal security model that can be bypassed via splitting.
