Buck Labs: Smart Contracts Security Review

Auditors

R0bert, Lead Security Researcher

Sujith Somraaj, Lead Security Researcher

Chinmay Farkya, Security Researcher

Report prepared by: Lucas Goiriz

January 5, 2026

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Buck Labs: Smart Contracts according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Buck Labs engaged with Spearbit to review the strong-smart-contracts-internal protocol. In this period of time a total of **39** issues were found.

Summary

Project Name	Buck Labs
Repository	strong-smart-contracts-internal
Commit	bbc8789c
Type of Project	DeFi, Stablecoin
Audit Timeline	Dec 18th to Jan 1st

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	6	4	2
Low Risk	8	7	1
Gas Optimizations	6	5	1
Informational	16	14	2
Total	**39**	**33**	**6**

4.1 Scope

The security review had the following components in scope for strong-smart-contracts-internal on commit hash bbc8789c:

```
src
├── access
│   └── AccessRegistry.sol
├── collateral
│   └── CollateralAttestation.sol
├── liquidity
│   ├── LiquidityReserve.sol
│   └── LiquidityWindow.sol
├── oracle
│   └── OracleAdapter.sol
├── policy
│   └── PolicyManager.sol
├── rewards
│   └── RewardsEngine.sol
└── token
    └── STRX.sol
```

5 Findings

5.1 High Risk

5.1.1 ABI struct mismatch in band config

Severity: High Risk

Context: LiquidityWindow.sol#L33-L41

Description: `LiquidityWindow` defines a local `IPolicyManager.BandConfig` struct that does not match the layout used by `PolicyManager`. This compiles because function selectors depend only on the function name and inputs, but the return value is ABI-decoded using the caller's struct layout. The result is that `LiquidityWindow` decodes the wrong fields from `PolicyManager.getBandConfig`.

In `LiquidityWindow` the interface is:

```
struct BandConfig {
    uint16 mintFeeBps;
    uint16 refundFeeBps;
    uint16 halfSpreadBps;
    uint16 alphaBps;
    uint16 floorBps;
    uint16 dexBuyFeeBps;
    uint16 dexSellFeeBps;
}
```

In `PolicyManager` the actual struct is:

```
struct BandConfig {
    uint16 halfSpreadBps;
    uint16 mintFeeBps;
    uint16 refundFeeBps;
    uint32 oracleStaleSeconds;
    uint16 deviationThresholdBps;
    uint16 alphaBps;
    uint16 floorBps;
    uint16 distributionSkimBps;
    CapSettings caps;
}
```

`LiquidityWindow.requestRefund` fetches the config and uses `floorBps` to compute the reserve floor:

```
IPolicyManager.BandConfig memory bandConfig =
    IPolicyManager(policyManager).getBandConfig(params.currentBand);
uint256 floor = _calculateFloor(bandConfig);
```

```
uint256 floorAmount18 = (totalSupply * config.floorBps) / BPS_DENOMINATOR;
```

Because `LiquidityWindow` expects `floorBps` as the fifth item, it decodes the fifth field of `PolicyManager.BandConfig`, which is `deviationThresholdBps`. With the default settings, `deviationThresholdBps` is 25/50/100 while `floorBps` is 100, so the computed floor is too small in Green and Yellow.

Impact: refunds can drain reserves down to `deviationThresholdBps` instead of the configured `floorBps`, reducing the intended liquidity floor and weakening reserve protection.

Recommendation: Consider updating `LiquidityWindow` to use the exact same `BandConfig` definition as `PolicyManager`, ideally by importing a shared interface. If only the floor is needed, replace the struct return with a dedicated getter, for example:

```
function getBandFloorBps(Band band) external view returns (uint16);
```

Buck Labs: Fixed in commit 5d79958.

Spearbit: Fix verified. Commit 5d79958 fixes the ABI struct mismatch by removing the local `BandConfig` decode in `LiquidityWindow` and using a dedicated floor getter instead:

- `src/liquidity/LiquidityWindow.sol now` calls `getBandFloorBps` and passes the returned `uint16` into `_calculateFloor`, so it no longer decodes the wrong field from `PolicyManager.BandConfig`.
- `src/policy/PolicyManager.sol` adds `getBandFloorBps`, matching the recommendation for a dedicated getter.

5.1.2 Eligible supply is not reduced on late entry

Severity: High Risk

Context: RewardsEngine.sol#L1257-L1274

Description: `RewardsEngine` integrates global eligible units using `currentEligibleSupply`, which is intended to represent the sum of balances that are currently eligible to earn. In the late-entry path of `_handleInflow`, the account is marked ineligible but the prior eligible balance is not removed from `currentEligibleSupply`. If the account already had an eligible balance before the late inflow, the global integrator keeps counting that balance even though the account will no longer accrue units.

```
bool isLateEntry = (checkpointStart > 0 && now_ >= checkpointStart && now_ < epochEnd);
if (isLateEntry) {
    s.eligible = false;
    s.lastAccrualTime = now_;
} else {
    if (!s.excluded) {
        s.eligible = true;
        currentEligibleSupply += amount;
    }
}
```

```
if (elapsed > 0 && currentEligibleSupply > 0) {
    globalEligibleUnits += currentEligibleSupply * elapsed;
}
```

This breaks the intended relationship between the per-account accruals and the global denominator used at distribution time. The contract then computes `deltaIndex` using a denominator that includes balances that are no longer eligible, so it dilutes rewards for eligible accounts and decouples `globalEligibleUnits` from the sum of actual eligible units.

Impact: rewards can be systematically diluted and distribution accounting becomes inconsistent because the denominator includes balances that are no longer earning.

Recommendation: When an account becomes ineligible during `_handleInflow`, subtract its eligible balance from `currentEligibleSupply` before flipping the flag. If eligibility can be partial per-balance, track an explicit eligible balance per account and update `currentEligibleSupply` using that value.

Buck Labs: Fixed in commit 559b098.

Spearbit: Fix verified. Commit 559b098 fixes the issue by removing the late-entry disqualification that inflated `currentEligibleSupply` and replacing it with late-inflow accounting, so the denominator only reflects earning balance.

- `src/rewards/RewardsEngine.sol` adds `lateInflow/lateInflowEpoch`, treats late inflows as non-earning and keeps `currentEligibleSupply` unchanged for those tokens; outflows now reduce eligible supply only for the earning portion, so the global denominator stays consistent.
- The accrual paths are updated to use `earningBalance` (balance minus late inflow), preventing dilution from ineligible units.

5.1.3 `totalExcludedSupply` drifts from reality

Severity: High Risk

Context: RewardsEngine.sol#L498-L551

Description: `RewardsEngine` tracks `totalExcludedSupply` as the sum of balances for excluded accounts, but it only updates this value inside the `setAccountExcluded()` and `setBreakageSink()` functions. Excluded accounts can still receive and send STRX through normal transfers and reward mints and those balance changes go through the token hook without adjusting `totalExcludedSupply`. As a result, `totalExcludedSupply` drifts from the real excluded balances over time.

```
function setAccountExcluded(address account, bool isExcluded) external onlyRole(ADMIN_ROLE) {
    // ...
    if (isExcluded) {
        totalExcludedSupply += s.balance;
        // ...
    } else {
        if (totalExcludedSupply >= s.balance) {
            totalExcludedSupply -= s.balance;
        } else {
            totalExcludedSupply = 0;
        }
    }
}
```

```
function onBalanceChange(address from, address to, uint256 amount) external onlyToken {
    if (from != address(0) && from != address(this)) {
        _handleOutflow(from, amount);
    }
    if (to != address(0) && to != address(this)) {
        _handleInflow(to, amount);
    }
}
```

Epoch configuration relies on this value to set the starting eligible supply:

```
currentEligibleSupply = IERC20(token_).totalSupply() - totalExcludedSupply;
```

If `totalExcludedSupply` is stale, `currentEligibleSupply` becomes incorrect. Because excluded balances can increase via reward mints and transfers, this causes a persistent mismatch between the actual excluded supply and what the system uses for denominators during distributions. In extreme cases where excluded balances shrink materially and the tracked value stays high, the subtraction can underflow and revert, blocking epoch configuration.

Impact: reward allocation is computed with an incorrect denominator and epoch configuration can revert if `totalExcludedSupply` exceeds total supply due to drift.

Recommendation: Consider updating `totalExcludedSupply` on every balance change for excluded accounts. On the token hook path, subtract `amount` when an excluded account is the sender and add `amount` when an excluded account is the recipient, including mint and burn flows. If that is not safe, replace `totalExcludedSupply` with a more robust mechanism that can not drift, such as tracking excluded balances per account and summing them on-demand.

Buck Labs: Fixed in commit 8105ffb.

Spearbit: Fix verified. Commit 8105ffb fixes the issue by updating `totalExcludedSupply` on every excluded account balance change. `src/rewards/RewardsEngine.sol` now subtracts `amount` from `totalExcludedSupply` in `_handleOutflow` when `s.excluded` and adds amount in `_handleInflow` when `s.excluded`. This keeps `currentEligibleSupply = totalSupply - totalExcludedSupply` accurate and prevents the drift/underflow path described.

5.2 Medium Risk

5.2.1 Late-entry disqualification is griefable

Severity: Medium Risk

Context: RewardsEngine.sol#L1263-L1265

Description: `RewardsEngine` marks an account ineligible for the remainder of the epoch on any inflow after `checkpointStart`. The hook is called for all transfers and all mints and the hook does not have context to distinguish a true late entry from a dust transfer or a reward mint. This means a third party can send a tiny amount of STRX to a victim during the checkpoint window and disqualify the victim for the entire epoch, even if the victim held their balance since epoch start. Claims also mint STRX to the claimant and therefore trigger the same inflow path, so claiming rewards during the checkpoint window can self-disqualify an account for the current epoch.

```
bool isLateEntry = (checkpointStart > 0 && now_ >= checkpointStart && now_ < epochEnd);
if (isLateEntry) {
    s.eligible = false;
    s.lastAccrualTime = now_;
}
```

```
super._update(from, to, value);
_notifyRewards(from, to, value);
```

Because the eligibility flip is unconditional on the source of the inflow, any transfer or mint after `checkpointStart` can be used to force ineligibility and the receiver can not prevent or undo it within the same epoch.

Impact: an attacker can deny rewards to targeted accounts for the entire epoch and users can unintentionally disqualify themselves by claiming during the checkpoint window.

Recommendation: Consider restricting late-entry disqualification to sources that represent intentional entry, such as primary mints from `LiquidityWindow` or DEX buys identified by a trusted pair address, and ignore incidental transfers and reward mints. If the hook cannot safely infer intent from `(from, to, amount)`, extend the hook signature to include a reason or source flag so the `RewardsEngine` can apply the late-entry rule only for entry actions.

Buck Labs: Fixed in commit 559b098.

Spearbit: Fix verified. Commit 559b098 fixes the issue by removing late-entry disqualification and replacing it with late-inflow accounting, so dust transfers and reward mints no longer nuke eligibility. `RewardsEngine` now tracks `lateInflow/lateInflowEpoch` and computes `earningBalance = balance - lateInflow`, so any inflow during the checkpoint just becomes non-earning for the current epoch rather than flipping eligible to false. This directly eliminates the griefing/self-disqualification path described while still enforcing the ”must hold through checkpoint” rule.

5.2.2 Reward debt not reset on exclusion

Severity: Medium Risk

Context: RewardsEngine.sol#L522

Description: `RewardsEngine` uses `rewardDebt` as the baseline that pairs with `unitsAccrued` for O(1) claims, as described in the claim formula below. When `unitsAccrued` is cleared without also clearing `rewardDebt`, the baseline no longer matches the units, which causes current-epoch rewards to be permanently offset until enough new units accrue to exceed the stale debt.

```
// claim = pendingRewards + (unitsAccrued * accIndex - rewardDebt)
amount = s.pendingRewards;
if (s.unitsAccrued > 0 && accRewardPerUnit > 0) {
    uint256 currentEpochReward = Math.mulDiv(s.unitsAccrued, accRewardPerUnit, ACC_PRECISION);
    if (currentEpochReward > s.rewardDebt) {
        amount += currentEpochReward - s.rewardDebt;
```

```
    }
}
```

Two admin paths clear `unitsAccrued` but do not reset `rewardDebt`. In `setAccountExcluded`, an account that is excluded has its `unitsAccrued` reset while `rewardDebt` is left unchanged. If that account is later re-included in the same epoch (and not late-entry disqualified), it starts accruing fresh units on top of a stale `rewardDebt` value.

```
// setAccountExcluded: exclusion path
s.unitsAccrued = 0;
s.eligible = false;
```

Similarly, `setBreakageSink` excludes the sink and clears `unitsAccrued` but leaves `rewardDebt` untouched, which creates the same accounting mismatch for that account.

```
// setBreakageSink
s.excluded = true;
s.eligible = false;
s.unitsAccrued = 0;
```

Impact: accounts that are excluded and then re-included within the same epoch can be underpaid for the remainder of that epoch, with rewards effectively suppressed until the stale `rewardDebt` is overcome.

Recommendation: Whenever `unitsAccrued` is reset as part of exclusion changes, also reset `rewardDebt` to keep the accounting invariant intact. Consider whether `pendingRewards` should be preserved or cleared based on policy, but ensure `unitsAccrued` and `rewardDebt` remain consistent.

Buck Labs: Fixed in commit 96a314c.

Spearbit: Fix verified. Commit 96a314c fixes the issue. `RewardsEngine` now zeroes `rewardDebt` alongside `unitsAccrued` in both `setAccountExcluded` and `setBreakageSink`, preserving the `unitsAccrued/rewardDebt` invariant and preventing the underpayment scenario described.

5.2.3 Ineligible accounts can mint synthetic breakage units via post-checkpoint outflows and self-transfers

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: If an eligible user transfers STRX tokens after the checkpoint ends, they technically forfeit their rewards until the epoch ends. This is managed by two global variables: `futureBreakageUnits` and `totalBreakageAllTime`.

After `checkpointStart`, late entrants are explicitly marked as `eligible = false` and do not accrue any reward units for the remainder of the epoch.

However, if such an ineligible account performs an outflow after checkpointEnd (including a self-transfer), the contract executes the following logic:

```
uint256 futureUnits = amount * remainingSeconds;
futureBreakageUnits += futureUnits;
totalBreakageAllTime += futureUnits;
```

This leads to the creation of a synthetic breakage unit, even though the user is initially ineligible for rewards because:

- The account has no future earnings to forfeit.
- No eligible units are being removed to offset the newly added breakage units.

This issue can be easily exploited through self-transfers. As a result, the rewards for honest, eligible participants could be diluted by inflating the totalUnits variables, used as a denominator in calculating elligible rewards.

Proof of Concept: As demonstrated in the provided test:

- User2 enters after `checkpointStart` and is correctly marked ineligible.
- After `checkpointEnd`, repeated self-transfers by user2 cause `totalBreakageAllTime` to increase monotonically.

```
function test_audit() external {
  vm.startPrank(user1);
  usdc.transfer(user2, 100e6);
  usdc.approve(address(liquidityWindow), 100e6);

  liquidityWindow.requestMint(address(user1), 100e6, 0, 0);

  vm.warp(block.timestamp + 12 days);

  /// CHECKPOINT STARTED = USER2 IS INELIGIBLE
  vm.startPrank(user2);
  usdc.approve(address(liquidityWindow), 100e6);

  liquidityWindow.requestMint(address(user2), 100e6, 0, 0);

  rewardsEngine._accounts(user1);
  rewardsEngine._accounts(user2);

  rewardsEngine.totalBreakageAllTime();
  rewardsEngine.futureBreakageUnits();

  vm.warp(block.timestamp + 16 days);

  for(uint256 i; i < 4; i++) {
    strx.transfer(user2, strx.balanceOf(user2));
   }
   rewardsEngine.totalBreakageAllTime();
   rewardsEngine.futureBreakageUnits();
}
```

Recommendation: Modify the post-checkpoint outflow logic to ensure that only eligible accounts can generate breakage. Also optionally guard the `onBalanceChange()` hook against self transfers.

Buck Labs: Fixed in commit 67007bb.

Spearbit: Fix verified. Commit 67007bb fixes the issue by adding a self-transfer short-circuit in `RewardsEngine` so no breakage accrues on no-op transfers, and it gates post-checkpoint breakage on `s.eligible` so ineligible accounts can't mint future breakage. With the existing `fromEarning`/late-inflow split, late entrants only have non-earning balance and therefore generate zero breakage.

5.2.4 Temporal collateral ratio inflation during reward distribution

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `distribute()` function in `RewardsEngine.sol` creates a temporal inconsistency in the collateral ratio (CR) calculation by depositing USDC into the reserve without immediately minting the corresponding STRX tokens.

This creates a synthetic CR inflation that persists until users progressively claim their rewards, potentially allowing operations based on an artificially healthy CR that doesn't reflect the protocol's actual solvency.

The collateral ratio is calculated as:

$$CR = \frac{R + HC \times V}{L}$$

Where:

- R = USDC balance in `liquidityReserve`.
- HC $\times$ V = haircut-adjusted off-chain collateral value.
- L = STRX total supply.

The rewards distribution flow is:

1. USDC is transferred to the reserve causing the `R` value to increase immediately.
2. The allocation of STRX tokens is determined by the CAP price, and no STRX minting takes place.
3. Users claim later: STRX is only minted when users call `claim()`, resulting in the increase of `L` increasing progressively as users claim.

The Temporal Window: Between steps 1 and 3, there exists a period (potentially hours/days) where:

- R has increased (USDC in reserve).
- L has NOT increased (STRX supply unchanged).
- CR appears artificially inflated.

This issue can cause downstream effects on the protocol's operational assumptions:

1. Incorrect Solvency Assessment: If CR is 0.99 (undercollateralized) and a reward distribution occurs, the temporary USDC influx could push CR above 1.0. The system appears healthy when it's actually undercollateralized.
2. Operational Decisions Based on False CR: `PolicyManager.getCAPPrice()` uses CR to determine pricing. Band transitions in `PolicyManager.refreshBand()` rely on CR. Distributions may be allowed/blocked incorrectly based on inflated CR. `enforceCROnClaim` checks use inflated CR for headroom calculations.

Proof of Concept: The following Proof of Concept explains how distributing rewards immediately changes the collateral ratio:

```
function test_audit() external {
    vm.startPrank(user1);
    usdc.transfer(user2, 100e6);
    usdc.approve(address(liquidityWindow), 100e6);

    liquidityWindow.requestMint(address(user1), 100e6, 0, 0);

    vm.startPrank(user1);
    usdc.approve(address(liquidityWindow), 100e6);

    vm.warp(block.timestamp + 200);
    console.log("-------------- TRANSFERRING TOKENS -----------------");
    console.log("user1 STRX balance before transfer:", strx.balanceOf(user1));
    // strx.transfer(user2, 50e18);
    console.log("user1 STRX balance after transfer:", strx.balanceOf(user1));
    console.log("user2 STRX balance after transfer:", strx.balanceOf(user2));
    usdc.transfer(distributor, 1000e6);

    console.log("===== BEFORE DISTRIBUTION =====");
    console.log("Collateral Ratio:", collateralAttestation.getCollateralRatio());

    vm.warp(block.timestamp + 30 days);
    vm.startPrank(distributor);
    usdc.approve(address(rewardsEngine), 1000e6);
    rewardsEngine.distribute(1000e6);

    console.log("===== AFTER DISTRIBUTION =====");
```

```
    console.log("Collateral Ratio:", collateralAttestation.getCollateralRatio());
}
```

The full proof of concept setup is shared with the project team.

Recommendation: Consider either one of the following fixes:

1. Immediate Minting to Treasury: Modify `RewardsEngine.distribute()` to mint STRX tokens immediately and hold them in the contract's treasury.
2. Virtual Supply Adjustment: Modify `CollateralAttestation.getCollateralRatio()` to account for pending reward obligations.

Buck Labs: Acknowledged. This is true and a valid concern. We've got a few protections in place, like `blockDistributeOnDepeg` and caps and operationally if we were ever right on something like .99 or .98 collateralization where this sort of thing would be possible, we would deploy a buffer of capital into the reserve before doing distribute to make it a clean overcollateralization.

Spearbit: Acknowledged.

5.2.5 `forceAllow()` can cause inclusion of a non-compliant address in the access list

Severity: Medium Risk

Context: AccessRegistry.sol#L123-L132

Description: From comments under the `STRX.sol:: _update()` function, it is clear that the `Denylist` could also be used to freeze accounts and prevent them from sending/receiving STRX tokens (USDT style blacklisting), and could be used independently of the allowlist (which is based on compliance).

But the logic for reinstating it using `forceAllow()` is problematic.

- Suppose a wallet address was not in the merkle tree (not compliant originally) and not in the allowlist.
- Then it gets denylisted simply due to being a sanctioned address, to prevent transfers etc...
- Later if this address is ever removed from the denylist, it will automatically be added to the allowlist, doesn't need to go via the usual registration route.
- It automatically becomes an allowlist member, without registering via a merkle proof, thus bypassing compliance checking.

```
function forceAllow(address account) external onlyOwner {
    if (_denylisted[account]) {
        _denylisted[account] = false;
        emit Denylisted(account, false);
    }
    if (!_allowed[account]) {
        _allowed[account] = true; // <<<
        emit AccessUpdated(account, true, currentRootId);
    }
```

This happens because `forceAllow()` assumes that the address that is being reinstated was already registered before, while it could have been just blacklisted from transfers.

Such an address can immediately start using `requestMint()` and `requestRefund()` facilities in `LiquidityWindow` even though they were not in the compliance list.

Recommendation: Consider splitting the `forceAllow()` logic into two functions: one to remove an address from denylist and other to force-include an address in the allowlist.

Buck Labs: Fixed in commit b3555cd.

Spearbit: Fix verified. Commit b3555cd fixes the issue by separating the `forceAllow()` logic into two functions.

5.2.6 CAP price uses maximum of oracle and collateral ratio instead of minimum

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `getCAPPrice()` function in `PolicyManager.sol` determines the collateral aware peg (CAP) price by using `Math.max(oraclePrice, cr)` during undercollateralized conditions (CR < 1.0).

In situations where the protocol is undercollateralized (CR < 1.0), the current code chooses the greater of the oracle price and the collateral ratio:

```
if (oracleHealthy) {
    // Oracle is healthy, use max(oracle, CR) per architecture
    (uint256 oraclePrice,) = IOracleAdapter(oracleAdapter).latestPrice();
    rawCAP = Math.max(oraclePrice, cr);
}
```

This leads the protocol to always present users with the ”best” price rather than a conservative estimate, potentially enabling value extraction if either data source becomes compromised or corrupted.

Recommendation: Replace `Math.max` with `Math.min` to use the more conservative price source:

```
if (oracleHealthy) {
    // Oracle is healthy, use min(oracle, CR) for conservative pricing
    (uint256 oraclePrice,) = IOracleAdapter(oracleAdapter).latestPrice();
    rawCAP = Math.min(oraclePrice, cr);
}
```

This ensures that even if one data source is compromised or erroneous, the protocol uses the lower (more conservative) price to protect reserves. The min-of-sources approach is a standard safety pattern in DeFi pricing oracles, as it prevents exploitation when any single price feed becomes unreliable.

Buck Labs: Acknowledged. There is a very good chance we switch to min instead of max with v2, but for this contract set and our initial launch, we're going to go with max. If oracle shows 0.95 but CR is 0.60, using 0.95 penalizes users beyond what the market believes the backing is worth. Or vice versa.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 CAP pricing trusts stale oracle if `strictMode` off

Severity: Low Risk

Context: PolicyManager.sol#L636-L637

Description: `PolicyManager`'s CAP pricing relies on `OracleAdapter.isHealthy(maxOracleStale)` to decide whether it can use the oracle price when CR is below 1.0. `OracleAdapter.isHealthy` short-circuits to `true` when `strictMode` is disabled, which means stale oracle prices are treated as healthy unless `strictMode` has already been toggled on via `syncOracleStrictMode` or the owner.

```
function isHealthy(uint256 maxStale) external view returns (bool) {
    if (!strictMode) {
        return true;
    }
    (uint256 price, uint256 updatedAt) = this.latestPrice();
    if (price == 0 || updatedAt == 0) return false;
    return block.timestamp <= updatedAt + maxStale;
}
```

```
uint256 maxOracleStale = cr < 1e18 ? STRESSED_ORACLE_STALENESS : HEALTHY_ORACLE_STALENESS;
bool oracleHealthy = IOracleAdapter(oracleAdapter).isHealthy(maxOracleStale);
```

```
if (oracleHealthy) {
    (uint256 oraclePrice,) = IOracleAdapter(oracleAdapter).latestPrice();
    rawCAP = Math.max(oraclePrice, cr);
}
```

If CR drops below 1.0 and `strictMode` has not yet been enabled, the oracle is always considered healthy and can feed stale or outdated prices into CAP. This can lift the refund price above what a fresh oracle would allow under stress, which is the exact situation strict mode is intended to protect against.

Impact: undercollateralized periods can use stale oracle prices to compute CAP, increasing refund prices and weakening solvency.

Recommendation: Enforce oracle freshness directly in `getCAPPrice()` when `cr < 1e18`, independent of `strictMode`. For example, require

```
updatedAt != 0
```

and

```
block.timestamp - updatedAt <= STRESSED_ORACLE_STALENESS
```

before using `oraclePrice`, otherwise fall back to `cr` only.

Buck Labs: Fixed in commit 9f920d5.

Spearbit: Fix verified. Commit 9f920d5 fixes the issue by enforcing oracle freshness directly in `getCAPPrice()` under stress, independent of `strictMode`. `PolicyManager` now calls `latestPrice()` directly and checks `updatedAt` against `STRESSED_ORACLE_STALENESS` before using the oracle price; otherwise it falls back to `cr`. This removes the stale-oracle bypass from `isHealthy(strictMode=false)`.

5.3.2 Refund cap ignores floor depletion

Severity: Low Risk

Context: PolicyManager.sol#L1211-L1219

Description: The refund-cap derivation computes a cap based on `alphaBps` and available reserve above the floor. When reserves are at or below the floor, the computed `capAmount` stays zero, but the function then overwrites the result with the configured `baseAggregate` refund cap:

```
uint256 capAmount = 0;
if (!isMint) {
    uint256 floorAmount = Math.mulDiv(config.floorBps, L, BPS_DENOMINATOR);
    if (reserveBalance > floorAmount) {
        uint256 alphaAmount = Math.mulDiv(config.alphaBps, L, BPS_DENOMINATOR);
        uint256 available = reserveBalance - floorAmount;
        capAmount = alphaAmount < available ? alphaAmount : available;
    }
}

if (capAmount != 0) {
    aggregateBps = Math.mulDiv(capAmount, BPS_DENOMINATOR, L);
} else {
    aggregateBps = 0;
}
```

```
if (baseAggregate != 0) {
    if (aggregateBps == 0 || baseAggregate < aggregateBps) {
        aggregateBps = baseAggregate;
    }
}
```

This means `aggregateBps` becomes `baseAggregate` even when there is no reserve buffer above the floor. If the intended behavior is to hard-stop refunds when reserves are at or below the floor, the cap never tightens to zero and the refund cap check continues to allow refunds at the configured base rate.

Impact: refunds are not capped to zero when reserves are at/below the floor, so the cap mechanism does not enforce a hard stop and relies on other safeguards to prevent draining.

Recommendation: If reserves at or below the floor should force a zero refund cap, explicitly return zero in that case and do not apply the `baseAggregate` fallback. A minimal fix is to only apply the `baseAggregate` minimum when `reserveBalance > floorAmount` for refunds.

Buck Labs: Fixed in commits 049accb and 2d91ea3.

Spearbit: Fix verified. Commits 049accb and 2d91ea3 fix the issue by hard-stopping refunds returning 0 when reserves are at/below the floor.

5.3.3 Reserve ratio cast can wrap bands

Severity: Low Risk

Context: PolicyManager.sol#L816-L818

Description: `PolicyManager` derives the reserve ratio and equity buffer in basis points and stores both in `uint16`. The values are computed with full precision, then down-cast without clamping. If the computed basis points exceed 65535, the `uint16` cast wraps modulo 2^{16}. This can turn a very healthy reserve ratio into an arbitrary smaller number and drive band selection off the wrong threshold.

```
uint16 reserveRatioBps =
    L == 0 ? 0 : uint16(Math.mulDiv(reserveBalance, BPS_DENOMINATOR, L));
```

Because band evaluation relies on `reserveRatioBps`, a wrap can misclassify the band. For example, if total supply is small and reserves are relatively high, `reserveRatioBps` can exceed 65535 and wrap to a value below the 5% or 2.5% thresholds, pushing the system into Yellow or Red despite being over-reserved.

Impact: incorrect band selection leads to wrong fees, spreads, caps, and governance signals in early or unusual reserve/supply conditions.

Recommendation: Clamp the computed basis points to `BPS_DENOMINATOR` (or at least to `type(uint16).max`) before casting, or store ratios in a wider type. A minimal fix is to compute into a `uint256` and cap at 10000 before assigning to `uint16`.

Buck Labs: Fixed in commit 3671887.

Spearbit: Fix verified. Commit 3671887 fixes the issue by clamping the computed ratios before the `uint16` cast. `PolicyManager` now computes `reserveRatioCalc/equityBufferCalc` as `uint256`, clamps each to `BPS_DENOMINATOR` and only then casts to `uint16`, preventing wrap and band misclassification.

5.3.4 Refund can round to zero and then revert

Severity: Low Risk

Context: LiquidityWindow.sol#L424-L464

Description: `LiquidityWindow.requestRefund` computes a gross USDC amount by scaling the 18-decimal price result down to 6 decimals. For very small STRX amounts, the scaled value can round down to zero. The function does not reject this locally when `minUsdcOut` is zero, so execution reaches `LiquidityReserve.queueWithdrawal` with amount equal to zero and reverts with `InvalidAmount`. The refund always fails in this case, but the error comes from the reserve and is confusing for callers. The impact is limited to unexpected revert behavior and poor UX for small refunds.

```
uint256 grossUsdc18 = (strxAmount * effectivePrice) / PRICE_SCALE;
uint256 grossUsdc = grossUsdc18 / USDC_SCALE_FACTOR;
```

```
// ...
ILiquidityReserve(liquidityReserve).queueWithdrawal(address(this), grossUsdc);
```

Recommendation: Add a guard after computing `grossUsdc` to revert locally when the amount is zero, or introduce a dedicated error such as `RefundTooSmall` to make the failure mode explicit.

Buck Labs: Fixed in commit 1362965.

Spearbit: Fix verified. Commit 1362965 fixes the issue by adding a zero-amount guard in `LiquidityWindow.requestRefund`, so the revert happens locally and clearly when the refund rounds to zero.

5.3.5 `PolicyManager.getAggregateRemainingCapacity` uses wrong caps

Severity: Low Risk

Context: PolicyManager.sol#L984-L1001

Description: `PolicyManager.getAggregateRemainingCapacity` computes mint and refund remaining capacity using simple BPS multipliers from the band config. For mints, a zero `mintAggregateBps` is treated elsewhere as unlimited, but here it produces a zero cap until `recordMint` initializes the cycle, so the reported remaining capacity is zero even though enforcement allows unlimited mints. For refunds, this function uses the base `refundAggregateBps` from config rather than the derived cap from `_computeRefundCap`, which incorporates reserve and floor logic, so the reported remaining capacity can disagree with actual enforcement. The impact is incorrect telemetry for keepers and front ends, which can lead to misleading UX and wrong automation decisions.

```
uint256 mintCapTokens = Math.mulDiv(snapshot.totalSupply, config.caps.mintAggregateBps,
↪  BPS_DENOMINATOR);
uint256 refundCapTokens = Math.mulDiv(snapshot.totalSupply, config.caps.refundAggregateBps,
↪  BPS_DENOMINATOR);
```

Recommendation: Treat a zero `mintAggregateBps` as unlimited in `getAggregateRemainingCapacity` by returning a sentinel such as `type(uint256).max` or a separate flag. For refunds, compute `refundAggregateBps` via `_computeRefundCap(snapshot)` before converting to tokens so the returned remaining capacity matches enforcement.

Buck Labs: Fixed in commit 897601c.

Spearbit: Fix verified. Commit 897601c fixes the issue by treating `mintAggregateBps == 0` as unlimited and using `_computeRefundCap` for refund capacity, so `getAggregateRemainingCapacity` now matches enforcement.

5.3.6 Inconsistent admin rights in queued withdrawals

Severity: Low Risk

Context: LiquidityReserve.sol#L315-L320

Description: In `LiquidityReserve.sol`, the `ADMIN_ROLE` can queue a request for USDC withdrawal, which can be either cancelled or executed later.

The current access control model requires the `ADMIN_ROLE` address to create such a withdrawal, and has the ability to cancel this withdrawal. But to execute it, the `TREASURER_ROLE` needs to step in.

```
function executeWithdrawal(uint256 id)
    external
    onlyRole(TREASURER_ROLE)
    nonReentrant
    whenNotPaused
{
```

As per current access model, the `ADMIN_ROLE` will be responsible to relay governance decisions. Not allowing the governance to finalise a withdrawal, while simultaneously allowing them to queue and cancel it, is inconsistent and breaks the process since a different role would need to be involved.

Recommendation: Consider allowing `ADMIN_ROLE` to call `executeWithdrawal()` as well.

Buck Labs: Fixed in commit 293f3397.

Spearbit: Fix verified. Commit 293f339 fixes the issue by allowing `ADMIN_ROLE` to execute withdrawals.

5.3.7 Potential loss of trading fees if liquidity moves to multiple dexes

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In a previous audit, it was mentioned that

> STRX trading fees are charged for only a single dexPair

and the team said they intended to

> add additional dexPair addresses if meaningful pools emerge

But right now, there is no way to expand dexPairs' list (only a single address can be used), the team will need to upgrade the STRX token contract and add code instead.

In any situation when there are multiple dex pools for $BUCK, the trades will incur inconsistent fee application, which can lead to loss of trading fees for Buck Labs, and liquidity moving to different pools ⇒ thus requiring constant reconfiguration for fee exemptions and protocol-operated arbitrage bot.

Recommendation: Consider adding logic that allows admins to add new dex pools to a list, where this swap fee is applied consistently.

Buck Labs: Fixed in commit 7a3aed56.

Spearbit: Fix verified. Commit 7a3aed56 fixes the issue by replacing the single `dexPair` with an `isDexPair` mapping and adding `addDexPair/removeDexPair`, so fees apply across multiple pools.

5.3.8 Spread calculation discrepancy in `mint()/refund()`

Severity: Low Risk

Context: LiquidityWindow.sol#L418-L427

Description: The spread calculation differs in mint and refund flows in `LiquidityWindow.sol`.

In `requestMint()`, first fees is subtracted from input USDC amount, then spread is applied to calculate actual deposit value (using effectivePrice).

Whereas in `requestRefund()`, it is opposite => spread is applied on total requested STRX amount, and later fees is calculated using final USDC amount (spread priced-in).

```
uint256 effectivePrice = _applySpread(params.capPrice, false, params.halfSpreadBps);
if (minEffectivePrice != 0 && effectivePrice < minEffectivePrice) {
    revert PriceTooLow(effectivePrice, minEffectivePrice);
}

// Calculate USDC amount in 18 decimals, then scale down to 6
uint256 grossUsdc18 = (strxAmount * effectivePrice) / PRICE_SCALE;
uint256 grossUsdc = grossUsdc18 / USDC_SCALE_FACTOR;
```

```
feeUsdc = _calculateFeeAmount(grossUsdc, params.refundFeeBps);
usdcOut = grossUsdc - feeUsdc;
```

As per the above logic from `requestRefund()`, the `grossUsdc` amount is calculated first, then `feeUsdc` is calculated later.

The `strxAmount` is total requested refund amount, and it includes the fee as well. This means the spread (`effectivePrice`) is being applied on total amount (actual refund order amount + fees), when calculating `grossUsdc`.

To put it in perspective, the spread is being applied on fee as well, which can potentially alter fee calculations slightly.

Recommendation: Consider using same math for applying spread in both mint and refund flows. The approach used in requestMint() is a better one, as spread being applied on fees does not seem logical.

Buck Labs: Acknowledged. We are going to accept this for v1 and address in v2, they are inverse flows + the math difference feels negligible to us to open it up right now.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 External self-call in `isHealthy`

Severity: Gas Optimization

Context: OracleAdapter.sol#L133

Description: The health check calls the contract through its own external interface to fetch the latest price. This triggers ABI encoding and an external call to self, which is more expensive than a direct internal call and is unnecessary because the data is already available within the contract.

```
(uint256 price, uint256 updatedAt) = this.latestPrice();
```

The behavior is correct, but the pattern adds avoidable gas overhead. Impact is limited to higher gas use for callers.

Recommendation: Expose the price function as a public view and call it directly, or add an internal helper such as `_latestPrice()` and use that from `isHealthy` to avoid the external self-call.

Buck Labs: Fixed in commit cef9102.

Spearbit: Fix verified. Commit cef9102 fixes the issue by adding an internal `_latestPrice()` helper and updating `isHealthy()` to call it instead of `this.latestPrice()`, removing the external self-call overhead.

5.4.2 Redundant intermediate variable in collateral calculations

Severity: Gas Optimization

Context: CollateralAttestation.sol#L236-L237, CollateralAttestation.sol#L284-L285, CollateralAttestation.sol#L378-L379

Description: Three instances in `CollateralAttestation.sol` use an unnecessary intermediate variable reserveUSDC before assigning to `R`, wasting gas on redundant memory operations.

```
/// current implementation
uint256 reserveUSDC = IERC20(usdc).balanceOf(liquidityReserve);
uint256 R = reserveUSDC; // or R = _scaleToEighteen(reserveUSDC, ...)
```

Recommendation: Eliminate the intermediate variableas follows:

```
uint256 R = IERC20(usdc).balanceOf(liquidityReserve);

uint256 R = _scaleToEighteen(
```

```
        IERC20(usdc).balanceOf(liquidityReserve),
        reserveAssetDecimals
);
```

Buck Labs: Fixed in commit d433c064.

Spearbit: Fix verified. Commit d433c06 fixes the issue by removing the redundant `reserveUSDC` intermediate variable in `CollateralAttestation.sol`.

5.4.3 Inconsistent error handling: `require` vs. custom errors in `AccessRegistry`

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: `AccessRegistry.sol` uses require statements for error handling, which is inconsistent with the rest of the codebase and consumes more gas than the custom error pattern used in other contracts.

Recommendation: Consider refactoring to use custom error to increase gas efficiency and be consistent with rest of the codebase.

Buck Labs: Acknowledged.

Spearbit: Acknowledged.

5.4.4 Enums don't need to be initialized to their default value

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: In `PolicyManager.sol`, the initialize() function sets the value of `_band` to `Band.Green`.

This is unnecessary because `_band` is an enum type and Green is the first entry in its definition, which means Green is anyways the default value of `_band`.

Recommendation: Consider removing this line to save some gas.

```
- _band = Band.Green;
```

Buck Labs: Fixed in commit 41c8c7da.

Spearbit: Fix verified. Commit 41c8c7d fixes the issue by removing the redundant `_band = Band.Green` assignment.

5.4.5 `ReentranyGuardTransient` can be used to save gas on all nonreentrant calls

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The following contracts use `ReentrancyGuardUpgradeable` from Openzeppelin:

- `STRX.sol`.
- `LiquidityReserve.sol`.
- `LiquidityWindow.sol`.

There is a more gas efficient library for blocking reentrancies: ReentrancyGuardTransient

Recommendation: Consider using `ReentrancyGuardTransient`.

Buck Labs: Fixed in commit 03050e33.

Spearbit: Fix verified. Commit 03050e3 fixes the issue by migrating `LiquidityReserve`, `LiquidityWindow` and BUCK to `ReentrancyGuardTransient` and adding the utility implementation.

5.4.6 Code cleanup in `LiquidityWindow.sol`

Severity: Gas Optimization

Context: LiquidityWindow.sol#L364, LiquidityWindow.sol#L377

Description: The following calculation is done 3 times in the `requestMint()` logic:

```
// ... = usdcAmount - feeUsdc;
```

The later two instances can be safely replaced with `netAmount`, as the result of `usdcAmount - feeUsdc` is already stored there.

Recommendation: Consider replacing these repetitive calculations with `netAmount`.

Buck Labs: Fixed in commit 4c005ea4.

Spearbit: Fix verified. Commit 4c005ea fixes the issue by reusing `netAmount` instead of recomputing `usdcAmount - feeUsdc`.

5.5 Informational

5.5.1 Epoch rollover overpays post-dist units

Severity: Informational

Context: RewardsEngine.sol#L1317-L1320

Description: `RewardsEngine` settles post-distribution accrual by adding units to `s.unitsAccrued` and baselining them with `rewardDebt`, which is meant to ensure those units earn zero for the rest of the epoch. This happens when an account is settled after a distribution in the same epoch.

```
s.unitsAccrued += postDistUnits;
s.rewardDebt += Math.mulDiv(postDistUnits, accRewardPerUnit, ACC_PRECISION);
```

When the account later crosses an epoch boundary, the prior epoch is finalized by multiplying the entire `s.unitsAccrued` by the prior epoch's `deltaIndex` without subtracting `rewardDebt`. That finalization path treats all accrued units as pre-distribution units, even if they were accumulated after the distribution.

```
if (report.deltaIndex > 0 && s.unitsAccrued > 0) {
    s.pendingRewards += Math.mulDiv(s.unitsAccrued, report.deltaIndex, ACC_PRECISION);
}
```

`deltaIndex` is computed at distribution time using a denominator that only includes units up to `distributionTime`. If a user accrues post-distribution units and does not settle again until the next epoch, those post-distribution units remain in `s.unitsAccrued` and are rewarded again at rollover, which credits rewards that were never part of the allocation. This creates a deterministic overpayment path for any account that accrues after distribution and only settles in the next epoch.

However, per the stated ops (monthly distribution after epoch end), `distributionTime` is capped to `epochEnd`, so there is no post-distribution window within the epoch. Under that timing, post-distribution units cannot exist, so this path is not exploitable in practice. The contract does not enforce that timing; if distributions are ever executed before epoch end (or if the next epoch is configured first so `distribute()` happens mid-epoch), then post-distribution units can accrue and the overpay becomes real.

Impact (if distribution happens mid-epoch): users can claim rewards on post-distribution time, so total claims for an epoch can exceed `tokensFromCoupon + dust`, inflating STRX beyond the intended allocation.

Recommendation: Ensure epoch rollover only finalizes units accrued before `distributionTime`. A minimal fix is to detect accounts that already crossed the distribution (for example by checking `s.lastAccrualTime >= report.distributionTime`) and skip or clear `s.unitsAccrued` before applying `deltaIndex` for the prior epoch. Alternatively, track a per-account flag or separate accumulator for post-distribution units so they are never included in the prior epoch finalization.

Buck Labs: Fixed in commit 3fa8c9a.

Spearbit: Fix verified. Commit 3fa8c9a fixes the issue by enforcing the operational constraints at the contract level, so the mid-epoch distribution path is no longer possible.

- `src/rewards/RewardsEngine.sol` now reverts if `distribute()` is called before `epochEnd (DistributionTooEarly)`, eliminating post-distribution accrual within the same epoch.
- `src/rewards/RewardsEngine.sol` also prevents `configureEpoch()` from being called if the current epoch hasn't been `distributed (MustDistributeBeforeNewEpoch)`, so you can't advance the epoch and then distribute mid-epoch.

5.5.2 Cap events/errors use misleading units

Severity: Informational

Context: PolicyManager.sol#L203

Description: `PolicyManager` originally emitted cap usage in basis points, but the implementation now tracks and emits actual token amounts. The event and error parameter names still indicate BPS, which can mislead indexers and monitoring tools that rely on names to interpret units.

```
event DailyLimitRecorded(CapType capType, uint256 newAmountBps);
error CapExceeded(CapType capType, uint256 requestedBps, uint256 remainingBps);
```

These fields are now populated with token amounts rather than basis points.

```
emit DailyLimitRecorded(CapType.MintAggregate, newAmount);
revert CapExceeded(CapType.RefundAggregate, amountTokens, remainingTokens);
```

Impact: off-chain consumers can misinterpret units, producing incorrect dashboards or alert thresholds.

Recommendation: Rename the event/error fields to reflect token amounts or introduce new events/errors with clear units and deprecate the old ones.

Buck Labs: Fixed in commit c70b3cf.

Spearbit: Fix verified. Commit c70b3cf fixes the issue by renaming the event/error fields to reflect token units. `PolicyManager` updates `DailyLimitRecorded` parameter name to `newAmountTokens` and renames `CapExceeded/TransactionTooLarge` parameters to `requestedTokens/remainingTokens/maxAllowedTokens`, matching the actual units emitted.

5.5.3 Unused code paths and params

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Several contracts contain unused helpers, events, errors, and configuration that are never read or exercised by on-chain logic. These components add surface area and can mislead operators into thinking the behaviors are enforced.

`PolicyManager` includes internal helpers that are not used by any path, as well as diagnostics that are declared but never emitted. The `CapHit` event and `NotAuthorized` error are never referenced and `InvalidAmount` is only used by an internal helper that is itself unused:

```
function _computeMintCap(SystemSnapshot memory snapshot) internal view returns (uint256) {
    BandConfig memory config = _resolveActiveConfig();
    return _deriveCaps(snapshot, config, true);
}

function _recordCounterTokens(
    RollingCounter storage counter,
```

```
    uint256 amountTokens,
    CapType capType,
    address user,
    uint64 cycle
) internal {
    if (counter.capCycle != cycle) {
        counter.capCycle = cycle;
        counter.amountTokens = 0;
        emit CapWindowReset(user, capType);
    }
}

function _validateAmountBps(uint256 amountBps) internal pure {
    if (amountBps > BPS_DENOMINATOR) revert InvalidAmount();
}
```

```
event CapHit(
    CapType capType, address indexed user, uint256 attemptedAmount, uint256 remainingCapacity
);
error NotAuthorized();
error InvalidAmount();
```

`LiquidityWindow` stores an `oracleAdapter` and exposes `configureOracle`, but no mint/refund path reads `oracleAdapter`. The `OracleUnhealthy` and `Unauthorized` errors and the `IOracleAdapter` interface are also unused.

```
address public oracleAdapter;

function configureOracle(address adapter) external onlyOwner {
    if (adapter == address(0)) revert ZeroAddress();
    oracleAdapter = adapter;
    emit OracleConfigured(adapter);
}
```

```
error OracleUnhealthy();
error Unauthorized();

interface IOracleAdapter {
    function latestPrice() external view returns (uint256 price, uint256 updatedAt);
    function isHealthy(uint256 maxStale) external view returns (bool);
    function setStrictMode(bool enabled) external;
}
```

`AccessRegistry` accepts a `rootId` parameter in `registerWithProof` but does not validate or use it, so the parameter has no effect:

```
function registerWithProof(bytes32[] calldata proof, uint64 /* rootId */) external
↪  whenNotPaused {
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
    require(MerkleProof.verifyCalldata(proof, merkleRoot, leaf), "AccessRegistry: invalid
    ↪  proof");
    _allowed[msg.sender] = true;
    emit AccessUpdated(msg.sender, true, currentRootId);
}
```

Impact: dead code and unused parameters increase maintenance burden and create ambiguity about which guardrails are actually enforced on-chain.

Recommendation: Remove unused helpers, events, errors and parameters, or wire them into the live logic so the configured surfaces reflect actual enforcement.

Buck Labs: Fixed in commit 9687766.

Spearbit: Fix verified. Commit 9687766 fixes the issue by removing the unused/dead code and parameters called out.

5.5.4 Redundant zero-address owner check

Severity: Informational

Context: AccessRegistry.sol#L55

Description: The constructor performs an explicit zero-address check on the initial owner before invoking `Ownable`'s constructor. `Ownable` already reverts when the owner is the zero address, so this duplicates the same validation logic:

```
require(initialOwner != address(0), "AccessRegistry: invalid owner");
```

Recommendation: Remove the explicit zero-address check and rely on Ownable's constructor validation or keep only one check in a single place.

Buck Labs: Fixed in commit 36c42a9.

Spearbit: Fix verified. Commit 36c42a9 fixes the issue by removing the redundant `require(initialOwner != address(0))` the `AccessRegistry`, leaving the `Ownable` constructor as the single check.

5.5.5 Dead code in multiply function

Severity: Informational

Context: OracleAdapter.sol#L184

Description: The code checks for multiplication overflow by dividing the result and comparing against the original value. In Solidity 0.8+, the multiplication reverts on overflow, so the check is never reached and can never return 0.

```
if (result / pow != value) return 0; // overflow check
```

This guard is effectively dead code and adds unnecessary operations. Impact is limited to minor gas and code clarity.

Recommendation: Remove the redundant overflow check, or if a non-reverting overflow behavior is intended, wrap the multiply in an unchecked block and document the sentinel return.

Buck Labs: Fixed in commit 6caf933.

Spearbit: Fix verified. Commit 6caf933 fixes the issue by removing the unreachable overflow check in the `OracleAdapter` contract.

5.5.6 Redundant liquidity guard in refund

Severity: Informational

Context: LiquidityWindow.sol#L436

Description: The refund path performs a liquidity availability check only when the USDC address is nonzero, but the same function later requires USDC to be configured before it can queue a withdrawal and transfer funds. This means the guard does not change the outcome when USDC is unset and only defers the revert until after extra computation and external calls. The `liquidityReserve` check in the condition is also redundant because that address is initialized once and is not updated during normal operation.

```
if (usdc != address(0) && liquidityReserve != address(0)) {
    uint256 reserveBalance = IERC20(usdc).balanceOf(liquidityReserve);
    // ...
    if (grossUsdc > availableLiquidity) {
```

```
        revert InsufficientLiquidity(grossUsdc, availableLiquidity);
    }
}

require(usdc != address(0), "USDC not configured");
```

This redundancy adds an unnecessary branch and extra work on the refund path without changing behavior.

Recommendation: Move the USDC configuration requirement to the top of the refund flow and remove the conditional guard around the liquidity check so the revert path is immediate and the logic is simpler.

Buck Labs: Fixed in commit bdc8e21.

Spearbit: Fix verified. Commit bdc8e21 fixes the issue by moving the USDC configuration check to the top of `requestRefund` and by removing the redundant conditional guard.

5.5.7 Redundant `nonReentrant` on `cancelWithdrawal`

Severity: Informational

Context: LiquidityReserve.sol#L351

Description: The `cancelWithdrawal` function is marked `nonReentrant` but only updates storage and emits an event. It does not perform external calls, so the reentrancy guard does not provide additional protection in this context.

```
function cancelWithdrawal(uint256 id) external onlyRole(ADMIN_ROLE) nonReentrant {
```

Recommendation: Remove the `nonReentrant` modifier from the `cancelWithdrawal` function.

Buck Labs: Fixed in commit f876cee.

Spearbit: Fix verified. Commit f876cee fixes the issue by removing the `nonReentrant` modifier from the `cancelWithdrawal` function in the `LiquidityReserve` contract.

5.5.8 Inconsistent token parameter naming in `LiquidityWindow.initialize()`

Severity: Informational

Context: LiquidityWindow.sol#L193

Description: In `LiquidityWindow.sol`, the `initialize()` function parameter is named `strc_` instead of `strx_`, creating inconsistency with the protocol's token naming convention (STRX) and the storage variable it populates.

Recommendation: Rename the parameter to `strx_` for consistency.

Buck Labs: Fixed in commit 068c6365.

Spearbit: Fix verified. Commit 068c636 fixes the issue by renaming the `initialize` parameter (now `buck_` as part of the STRX $\rightarrow$ BUCK rename), resolving the inconsistency.

5.5.9 Fallback oracle freshness critical in strict mode

Severity: Informational

Context: OracleAdapter.sol#L120

Description: When CR < 1.0 (strict mode), the system relies on oracle pricing for CAP calculations with a 15-minute staleness threshold.

If the primary oracle (Pyth) fails after operating for extended periods, the system immediately falls back to the secondary oracle. If the fallback oracle is stale (>15 minutes old), this creates an instant DoS condition where all mint/refund operations revert due to StaleOraclePrice errors.

Risk Scenario:

1. CR drops to 0.95 $\rightarrow$ Strict mode activated (15-min staleness limit).
2. Pyth functions normally for 24 hours.
3. Fallback oracle updated infrequently (every 24-48 hours).
4. Pyth suddenly fails.
5. System falls back to stale oracle $\rightarrow$ DoS (all operations revert).

Recommendation: Implement strict operational controls and update the fallback oracle even if Pyth is functioning perfectly to avoid sudden DoS of the protocol.

Buck Labs: Acknowledged. Operational plan established:

- Healthy mode (CR $\geq$ 1.0): Update fallback every 24-48 hours (within 72-hour staleness limit).
- Strict mode (CR $<$ 1.0): Update fallback every 10-12 minutes (within 15-minute staleness limit).
- Backend service maintains fallback proactively, not reactively after Pyth failures.

Spearbit: Acknowledged.

5.5.10 Redundant `decimals()` override in `STRX` Token

Severity: Informational

Context: STRX.sol#L175-L178

Description: The `STRX.sol` contract overrides `decimals()` to return a hard-coded value of 18, which is already the default value returned by OpenZeppelin's ERC20Upgradeable parent contract.

Recommendation: Remove the redundant override to reduce code surface and improve clarity.

Buck Labs: Fixed in commit 77ade334.

Spearbit: Fix verified.

5.5.11 Code cleanup

Severity: Informational

Context: PolicyManager.sol#L8, PolicyManager.sol#L30, PolicyManager.sol#L39, RewardsEngine.sol#L41

Description: Few minor code quality issues were identified in `PolicyManager` and `RewardsEngine` contracts:

1. Unused SafeCast Import: The SafeCast library is imported in `PolicyManager.sol` but never used in the contract.
2. Outdated TODO Comment in `PolicyManager.sol`.
3. Unused `isHealthyCollateral()` and `balanceOf()` function declarations in interface.

Recommendation: Consider removing the unused import, function declarations, and the outdated code documentation.

Buck Labs: Fixed in commits b88f4e6, 0c3e2f2, a24b262 and 41c8c7d

Spearbit: Fix verified. The issue was fixed by removing the unused `SafeCast` import, clean the outdated comment and drop the unused interface declarations (`isHealthyCollateral` and `balanceOf`).

5.5.12 Redundant `treasury` storage variable

Severity: Informational

Context: CollateralAttestation.sol#L83-L85

Description: The `treasury` state variable in `CollateralAttestation.sol` serves no functional purpose. The contract stores the treasury address but never uses it in any calculations or logic.

Recommendation: Remove redundant treasury-related code:

```
- address public treasury;

  function initialize(
      address admin,
      address attestor,
      address _strxToken,
      address _liquidityReserve,
      address _usdc,
-     address _treasury,
      uint8 _reserveAssetDecimals,
      uint256 _healthyStaleness,
      uint256 _stressedStaleness
  ) external initializer {
      // ... validation ...
-     // Treasury can be zero initially (backwards compatibility)

      strxToken = _strxToken;
      liquidityReserve = _liquidityReserve;
      usdc = _usdc;
-     treasury = _treasury;
      // ...
  }

- function setTreasury(address _treasury) external onlyRole(ADMIN_ROLE) {
-     address oldTreasury = treasury;
-     treasury = _treasury;
-     emit TreasuryUpdated(oldTreasury, _treasury);
- }

- event TreasuryUpdated(address indexed oldTreasury, address indexed newTreasury);
```

Buck Labs: Fixed in commit 1dfc8c2.

Spearbit: Fix verified. Commit 1dfc8c2 fixes the issue by removing the unused treasury state, initializer parameter, setter and event from `CollateralAttestation.sol`.

5.5.13 `renounceOwnership()` should be blocked to prevent accidentally losing contract ownership

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The following contracts have a defined owner address, which acts as an admin for protocol operations:

- `STRX.sol`.
- `AccessRegistry.sol`.
- `LiquidityWindow.sol`.
- `OracleAdapter.sol`.

If `renounceOwnership()` is called by current owner by mistake, this can lead to permanent loss of ownership, which can create issues with protocol operations.

Recommendation: `renounceOwnership()` should be overriden and changed to always revert.

Buck Labs: Fixed in commit 36f7e21.

Spearbit: Fix verified. Commit 36f7e21 fixes the issue by adding `renounceOwnership()` overrides to `AccessRegistry`, `LiquidityWindow` and `OracleAdapter`.

5.5.14 Missing events in `LiquidityReserve::Initialize()` function

Severity: Informational

Context: LiquidityReserve.sol#L141-L145, LiquidityReserve.sol#L146-L151

Description: The `initialize()` function in `LiquidityReserve.sol` handles the configuration of some contract addresses, but related events are not emitted here.

The following events are missing for the Treasurer address:

```
emit TreasurerSet(treasurer_);
emit RecoverySinkSet(treasurer_, true);
```

The following events are missing for the Liquidity Window:

```
emit LiquidityWindowSet(liquidityWindow_);
emit RecoverySinkSet(liquidityWindow_, true);
```

Recommendation: Consider emitting these events for consistency.

Buck Labs: Fixed in commit a40c5dc3.

Spearbit: Fix verified. Commit a40c5dc fixes the issue by emitting `LiquidityWindowSet`, `TreasurerSet` and `RecoverySinkSet` during `LiquidityReserve.initialize`.

5.5.15 The `RootUpdated` event in AccessRegistry can be improved

Severity: Informational

Context: AccessRegistry.sol#L21, AccessRegistry.sol#L66-L68, AccessRegistry.sol#L96

Description: In `AccessRegistry.sol`, the `setRoot()` function is used to push a merkle root, that helps determine which addresses are allowed to interact with the protocol.

When a new root is published, the following event is emitted:

```
event RootUpdated(bytes32 indexed newRoot, uint64 indexed rootId);
```

Both attestor service and owner can publish new roots, but this event does not have a mention of who published a certain `rootID`.

Since the team wants to keep logs of who updated what, this event should also track who actually updated the new `rootID` (for offchain ops).

Recommendation: Add an ”attestor” address to this event definition, and log msg.sender while emitting this event in `setRoot()`.

Buck Labs: Fixed in commit 1bb6023e.

Spearbit: Fix verified. Commit 1bb6023e fixes the issue by adding `updatedBy` to the `RootUpdated` event and emitting `msg.sender`.

5.5.16 USDC market price is not considered in minting calculations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In a previous audit report, risks related to USDC depeg were discussed, but there are more impacts of a potential depeg, including over-minting of STRX coin.

In `requestMint()`, the USDC price is assumed to be == 1 USD, and used as is in determining collateral value.

The real USDC market price is not factored in. If USDC were to depeg, any users can come to the protocol and use their USDC (at a fixed face value of $1) and use it to mint STRX.

This can lead to a death spiral for STRX as the new collateral backing (USDC) is worth less than what we assume it to be.

Over-valuing collateral is always a bad idea, especially when we are minting another token against it.

Recommendation: Consider using USDC market price to calculate real collateral value.

Buck Labs: Acknowledged. We think it's too late for us to incorporate a USDC oracle and dynamic USDC pricing into v1. It's definitely a top priority for the v2 contract ecosystem. For now, we are going to accept this risk, and if USDC depegs we are going to pause minting and redemptions.

Spearbit: Acknowledged.