Strong Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

Blckhv

Slavcheww

BengalCatBalu

December 19, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

STRC (“Stretch”) is a variable-rate, perpetual preferred stock issued by Strategy (formerly MicroStrategy) to raise capital to fund Bitcoin purchases. It pays monthly dividends with a variable rate adjusted to help keep the share price near its $100 par value.

Strong is an upgradeable smart-contract system that issues BUCK (formerly STRX) ERC-20 which is backed by on-chain USDC reserves and attested off-chain STRC collateral. Price stability is enforced via a collateral-aware peg (CAP) such that BUCK trades at $1 when the collateral ratio (CR) 1 and depegs deterministically when undercollateralized.

A sanctions-compliant primary market handles mint/refund at CAP ± policy spreads, routing all USDC to an on-chain reserve with enforced caps and floors. Secondary BUCK trading is permissionless; issuance and redemption remain compliance-gated. A central `PolicyManager` implements a solvency band state machine that dynamically controls pricing, fees, mint/refund limits, and oracle strictness. `CollateralAttestation` publishes haircut-adjusted STRC valuations on-chain and computes the CR. Yield is derived from STRC coupon USDC that enters the reserve and is reminted as BUCK via an index-based `RewardsEngine`.

5 Audit Scope

The scope was limited to:

```
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
```

6 Executive Summary

Over the course of 11 days, the Cyfrin team conducted an audit on the Strong smart contracts provided by Strong. In this period, a total of 51 issues were found.

This audit identified 1 critical finding, 7 high-risk findings, 9 medium-risk findings, and 22 low-risk issues, with the most severe concentrated in the `RewardsEngine` accounting mechanism and other solvency-sensitive protocol logic. The single critical issue demonstrated that rewards accounting could inflate user claims beyond the amount actually distributed, violating the core invariant that total claimed rewards must never exceed total distributed rewards. This flaw stemmed from desynchronized global unit tracking that allowed previously rewarded units to participate again in future distributions, creating a direct risk of under-collateralization. Addressing this and other issues required a fundamental redesign of the rewards architecture.

The high-risk findings that materially impacted protocol solvency and correctness. Key examples included inconsistent liability definitions across `PolicyManager` and `CollateralAttestation` that produced incorrect collateral ratios and band behavior when BUCK (formerly STRX) and/or USDC deviate from $1; an ineffective anti-sniping mechanism that could be bypassed through trivial transfers; inclusion of excluded or ineligible supply in reward calculations, systematically diluting eligible user rewards; race-condition-dependent reward outcomes caused by broken phantom locked-unit accounting; reward minting behavior that could reduce the collateral ratio during price volatility between distribution and claim; and mint/refund caps that could be bypassed through repeated small transactions due to zero-basis-point rounding. Collectively, these issues broke key invariants and could have resulted in incorrect incentives and insolvency risk.

The medium-severity findings primarily included CAP price invariant violations when spreads were applied during under-collateralized states, oracle freshness logic that could be abused to block mints and refunds, reward loss or misallocation around epoch boundaries, inconsistencies in treasury and reserve accounting, and sequencing issues where collateral checks were performed before balances were properly updated. While less severe in isolation, several of these findings could compound during stressed market conditions.

Considering the number of issues identified, the non-trivial changes required during mitigation (including a complete rewrite of the `RewardsEngine` contract), and the short turnaround time for reviewing the mitigation fixes, it is recommended that a follow-up audit be undertaken prior to deploying significant monetary capital to production.

Summary

Project Name	Strong
Repository	strong-smart-contracts-internal
Commit	2cfcd5febed1...
Audit Timeline	Nov 28th - Dec 12th
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	7
Medium Risk	9
Low Risk	22
Informational	7
Gas Optimizations	5
Total Issues	51

Summary of Findings

[C-1] STRX rewards inflation results in risk of undercollateralization as more can be claimed than is distributed	Resolved
[H-1] `PolicyManager` and `CollateralAttestation` use different liability metrics, creating band/pricing divergence	Resolved
[H-2] `RewardsEngine` anti-snipe protection is totally broken	Resolved
[H-3] Ineligible addresses are not excluded from `settledLockedUnits` calculations which locks and decreases rewards	Resolved
[H-4] Locked/unlocked units are distributed incorrectly due to broken `phantomLockedUnits` accounting	Resolved
[H-5] The STRX reward mechanism can push the system into an undercollateralized state or drastically change the protocol configuration	Resolved
[H-6] `RewardsEngine::_settle` incorrectly accumulates rewards when claimed past the epoch end timestamp	Resolved
[H-7] Multiple small mints/refunds can bypass caps due to rounding down to zero basis points	Resolved
[M-1] CAP price invariant can be violated when `CR < 1`	Resolved
[M-2] `blockFreshCheck()` modifier logic can be weaponized to block STRX mints and refunds	Resolved
[M-3] STRX trading fees are charged for only a single `dexPair`	Acknowledged
[M-4] `LiquidityWindow::requestRefund` treasurer fee avoids USDC withdrawal delay from `LiquidityReserve`	Resolved
[M-5] Collateral ratio is calculated with mismatched values in both USDC and USD terms	Acknowledged
[M-6] Missing band refresh in `RewardsEngine::distribute` results in incorrect `distributionSkimBps`	Resolved
[M-7] `RewardsEngine::distribute` checks CAP price before withdrawing treasurer funds	Resolved
[M-8] `LiquidityReserve` withdrawal queue may be manipulated to enforce maintaining a specific CR and R/L	Resolved

[M-9] Pausing of `RewardsEngine::distribute` also delays users claiming rewards	Resolved
[L-01] Increasing the `KYCRegistry` epoch does not invalidate old proofs	Resolved
[L-02] Multisig owner is unlikely able to maintain fresh oracle fallback prices when `CR < 1`	Resolved
[L-03] `LiquidityReserve::setLiquidityWindow` does not revoke the `DEPOSITOR_ROLE` from outdated addresses	Resolved
[L-04] `PolicyManager::refreshBand` fails to update derived caps	Resolved
[L-05] Tiered delay of large withdrawals can change between enqueuing and execution	Resolved
[L-06] Missing condition in `PolicyManager::syncOracleStrictMode` event monitoring logic	Acknowledged
[L-07] `OracleAdapter::setFallbackPrice` allows any arbitrary fallback price to be set	Acknowledged
[L-08] `OracleAdapter::_tryPyth` may reject high confidence prices due to overly restrictive validation	Resolved
[L-09] `PolicyManager::requiresGovernanceVote` use `_lastSnapshot` instead of current snapshot	Resolved
[L-10] STRX charges fees on providing liquidity in the `dexPair`	Acknowledged
[L-11] Rewards may be lost as `RewardsEngine::configureEpoch` cannot always be updated at the correct time	Resolved
[L-12] `LiquidityReserve` and treasury will accumulate STRX rewards they cannot claim	Acknowledged
[L-13] `CollateralAttestation::V` and Pyth oracle configuration should be initialized upon deployment	Resolved
[L-14] Users who send STRX directly to `RewardsEngine` will continue to receive yield on their balance	Resolved
[L-15] `OracleAdapter::_scalePythValue` exponent validation should mirror the canonical Pyth Solidity SDK	Resolved
[L-16] Oracle staleness validation is unnecessarily convoluted and conflicting	Resolved
[L-17] `RewardsEngine::setEarningParams` called during epoch may alter existing reward distribution	Resolved
[L-18] `RewardsEngine::setAccountExcluded` called within a STRX pause period will cause prior rewards to be lost	Resolved
[L-19] Rewards should be claimable once per distribution rather than once per epoch	Resolved
[L-20] Use of `LiquidityReserve` balance of USDC overstates solvency by ignoring pending withdrawals	Resolved
[L-21] Pyth confidence interval is not applied to pricing mints/refunds	Acknowledged
[L-22] Effective mint price calculated in `LiquidityWindow::requestMint` should be rounded up	Resolved

[I-1] `LiquidityWindow::setUSDC` should be removed in favor of setting `usdc` within `LiquidityWindow::initialize`	Acknowledged
[I-2] Unchained initializers should be called instead	Acknowledged
[I-3] Reconfiguration of STRX modules does not remove fee exemption of old addresses	Resolved
[I-4] Incorrect block time assumptions in `PolicyManager::_computeCurrentSnapshot`	Acknowledged
[I-5] `Band.Emergency` config cannot be enabled without `PolicyManager` contract upgrade	Resolved
[I-6] `LiquidityWindow::_getCAPPrice` is unused and can be removed	Resolved
[I-7] System addresses cannot bypass the KYC check within `LiquidityWindow`	Acknowledged
[G-1] Redundant zero address check in `STRX::burn` can be removed	Resolved
[G-2] Redundant `isRecoverySink` assignment logic should be removed from `LiquidityReserve::initialize`	Resolved
[G-3] `STRX::_update` conditional branches can be simplified	Resolved
[G-4] Unreachable code in `OracleAdapter::_scalePythValue` should be removed	Resolved
[G-5] `LiquidityWindow::requestMint` should combine separate calls to `LiquidityReserve::recordDeposit` for both fee and net amount	Resolved

7 Findings

7.1 Critical Risk

7.1.1 STRX rewards inflation results in risk of undercollateralization as more can be claimed than is distributed

Description: The `RewardsEngine` accounting model allows previously rewarded `unlockedUnits` to participate in subsequent `distribute()` calls. This happens due to a desynchronization between `settledUnlockedUnits` and `unitsAccounted`. Specifically, after a distribution, `accRewardPerUnit` is increased based on `newUnits = settledUnlockedUnits - unitsAccounted` before `unitsAccounted` is updated to `settledUnlockedUnits`.

However, additional units can later become part of `unlockedUnits` through:

1. Post-distribution `_settle()` calls.
2. Phantom unit realization via `_addUnits()`.
3. Transfers or mints after a distribution.

These newly materialized units are not consistently reflected in `unitsAccounted`, which causes already rewarded units to be included again in future reward index calculations. As a result, historical units can receive rewards multiple times across sequential distributions.

Impact: Users can claim significantly more STRX than the amount actually distributed via `RewardsEngine::distribute`, as can be observed in the below PoC in which a total distribution of 60 STRX resulted in a user claiming 108 STRX. This breaks the fundamental accounting invariant `totalClaimed  totalDistributed` and results in risk of under-collateralization.

Proof of Concept: The following test should be added to `RewardsEngineTest.t.sol`:

```
function testGlobalRewards() public {
    _configureTokenAndEpoch(1, 1_000, 1_000 + 30 days);
    vm.prank(TIMELOCK);
    rewards.setEarningParams(0, !DEFAULT_CLAIM_ONCE);
    vm.prank(TIMELOCK);
    rewards.setMinClaimTokens(0);

    vm.warp(1_000);
    token.mint(ALICE, 90 ether);
    token.mint(BOB, 10 ether);

    vm.warp(3_000);
    uint256 couponAmount = _couponForTokens(50 ether);
    token.transfer(ALICE, BOB, 10 ether); // trigger settle
    vm.prank(TIMELOCK);
    rewards.distribute(couponAmount); // Distribute 50 STRX
    token.mint(ALICE, 100 ether); // 180/20


    vm.warp(4_000);
    token.transfer(ALICE, BOB, 10 ether); // 170 / 30 + trigger settle
    console.log("Unlocked: ", rewards.getUnlockedUnits(ALICE));
    console.log("Locked:", rewards.getLockedUnits(ALICE));
    console.log("Acc reward per unit: ", rewards.accRewardPerUnit());
    console.log("Pending rewards: ", rewards.pendingRewards(ALICE));

    couponAmount = _couponForTokens(10 ether);
    vm.prank(TIMELOCK);
    rewards.distribute(couponAmount); // distribute extra 10 STRX
    console.log("Unlocked 2: ", rewards.getUnlockedUnits(ALICE));
    console.log("Locked 2:", rewards.getLockedUnits(ALICE));
    console.log("Acc reward per unit 2: ", rewards.accRewardPerUnit());
    console.log("Pending rewards 2: ", rewards.pendingRewards(ALICE));
```

```
    uint256 alice_balance_before = token.balanceOf(ALICE);
    vm.prank(ALICE);
    rewards.claim(ALICE);
    uint256 alice_balance_after = token.balanceOf(ALICE);
    console.log("ALICE claim rewards ", (alice_balance_after - alice_balance_before) / 1e18);
    assertEq(alice_balance_after - alice_balance_before, 108 ether); // WE DISTRIBUTE ONLY 60 STRX !!!
}
```

Recommended Mitigation: The mitigation is complex and requires a complete architectural overhaul to fix this completely. Consider adding a strict invariant check that `totalClaimed  totalDistributed`.

Strong: Fixed in commit b3f877f.

Cyfrin: Verified. Rewards are now distributed and claimed by an epoch-bound, index-based mechanism.

7.2 High Risk

7.2.1 `PolicyManager` and `CollateralAttestation` use different liability metrics, creating band/pricing divergence

Description: `CollateralAttestation::getCollateralRatio` and `LiquidityWindow::_calculateFloor` both incorrectly assume that STRX will being pegged to USDC at a 1:1 ratio in perpetuity; however, this is not guaranteed as the STRX price continuously fluctuates.

The collateral ratio is derived from the following formula:

`CR = (R + HC × V) / L`, where `R = USDC reserve`, `HC = Haircut Coefficient`, `V = USD Brokerage Mark`, `L = STRX Total Supply`.

Since CR is used as the price for STRX, it is only natural that STRX is not converted into USDC terms when calculating `L` as USD/STRX corresponds to the units of measurement obtained from the oracle. On the other hand, it is strange to quote the CR in USDC/STRX terms, as is currently the case. The problematic variable here is `L`, which should also be quoted in USDC terms.

```
function getCollateralRatio() public view returns (uint256) {
    uint256 L = IERC20(strxToken).totalSupply();
    if (L == 0) return type(uint256).max; // Infinite CR when no supply

    // Count ONLY reserve USDC as collateral (treasury is protocol profit)
    uint256 reserveUSDC = IERC20(usdc).balanceOf(liquidityReserve);
    uint256 R = reserveUSDC;

    // CR = (R + HC×V) / L
    // All in 18 decimals: R (scaled from reserveAssetDecimals), HC (18 decimals), V (18 decimals), L
    ↪  (18 decimals)
    uint256 scaledR = _scaleToEighteen(R, reserveAssetDecimals);
    uint256 haircutValue = Math.mulDiv(HC, V, PRECISION);
    uint256 numerator = scaledR + haircutValue;

    return Math.mulDiv(numerator, PRECISION, L);
}
```

On the contrary, `PolicyManager::_computeCurrentSnapshot` correctly prices the STRX by using the onchain price feed to quote the total dollar value of the liability:

```
// Query on-chain state directly (NO TRUST REQUIRED)
uint256 totalSupply = IStrcToken(strxToken).totalSupply();
// Scale USDC balance from 6 decimals to 18 decimals for compatibility with L (liabilities)
// This ensures accurate reserve ratio and floor calculations
uint256 reserveBalance = IERC20(usdc).balanceOf(liquidityReserve) * 1e12;
(uint256 navPerToken,) = IOracleAdapter(oracleAdapter).latestPrice();
uint256 lastOracleBlock = IOracleAdapter(oracleAdapter).getLastPriceUpdateBlock();

// Calculate L (total liability in dollars)
// L = totalSupply * NAV (scaled properly)
uint256 L =
    totalSupply == 0 || navPerToken == 0 ? 0 : Math.mulDiv(totalSupply, navPerToken, 1e18);
```

This effectively makes all the calculations use the same underlying asset denomination, which is the correct way of evaluating the collateralization and risk profile of the protocol.

Impact:

1. The current band may not accurately reflect the intended configuration for the actual CR.
2. The equity buffer, allocated in the snapshot, will deviate when compared with the incorrect collateral ratio depending on whether the oracle price is above or below $1. This could result in blocking refunds when the cap is near the daily limit.

3. Mint/refund caps will be incorrect, again when compared with the incorrect collateral ratio.

Scenario	PolicyManager's L	CollateralAttestation's L	Inconsistency
STRX = $1.00	1M tokens × $1 = $1M	1M tokens	Both agree (incidentally)
STRX = $0.50	1M tokens × $0.50 = $500k	1M tokens	Mismatch
STRX = $2.00	1M tokens × $2 = $2M	1M tokens	Mismatch

Recommended Mitigation: Both contracts use the same definition of `L` in reserve currency terms.

Strong: Fixed in commit bf2c4db. L is now pegged to $1 notional in `PolicyManager`. Reasoning is, if the CR drops to .5 and the token is now trading at fifty cents, we need the bands to get more conservative and defensive, not less. If we peg it to the CR price, the bands get greener and greener the further the price depegs. So, `PolicyManager` now uses L = totalSupply (at $1 notional).

Cyfrin: Verified. Bands/caps now use $1 assumption for conservative risk management during depegs.

7.2.2 `RewardsEngine` anti-snipe protection is totally broken

Description: The anti-snipe system implemented in `RewardsEngine` assumes that rewards should not be counted for STRX that were purchased within the time window `[epochEnd - antiSnipeCutoffSeconds, epochEnd]`.

`RewardsEngine::_settle` checks whether the timestamp of the last STRX inflow into the account falls within the cutoff window and, if it does, then rewards for those tokens should not be counted:

```
function _settle(AccountState storage state, address account) internal {
    ...
    uint64 lastInflow = state.lastInflow;

    // Anti-snipe check: if deposit was after cutoff, wait for next epoch
    uint64 cutoff = _currentCutoff();
    if (cutoff != 0 && lastInflow != 0 && lastInflow >= cutoff) {
        uint64 epochEndCheckpoint = epochEnd;
        if (epochEndCheckpoint > accrualStart) {
            accrualStart = epochEndCheckpoint;
        }
    }

    ...

    uint256 elapsed = uint256(effectiveEnd - accrualStart);
    uint256 newUnits = elapsed * balance;
    if (newUnits != 0) {
        _addUnits(state, account, newUnits);
    }

    state.lastAccrual = currentTime;
}
```

However, the current implementation of this system is completely broken due to how `state.lastInflow` is updated:

```
function _handleInflow(address account, uint256 amount, uint64 senderLastInflow) internal {
    if (amount == 0) return;

    AccountState storage state = _accounts[account];
    _settle(state, account);

    state.balance += amount;
```

```
    // Inherit the oldest lastInflow time to prevent transfer resets
    // Ensures tokens keep their original mint time for anti-snipe calculations
    if (state.lastInflow == 0) {
        // New account: use sender's time if available (transfer), else current time (mint)
        state.lastInflow = senderLastInflow != 0 ? senderLastInflow : uint64(block.timestamp);
    } else if (senderLastInflow != 0 && senderLastInflow < state.lastInflow) {
        // Recipient already has tokens: inherit sender's OLDER mint time
        // Example: Alice minted Day 0, Bob minted Day 5, Alice transfers to Bob
        //          Bob's lastInflow should be Day 0 (oldest)
        state.lastInflow = senderLastInflow;
    }
    // else: recipient's existing lastInflow is already older, keep it
}
```

Based on the above logic, the `lastInflow` of an account is determined in four possible ways:

1. It is the user's first time receiving tokens (`lastInflow = 0`) via mint. In this case, `lastInflow` becomes `block.timestamp`.
2. It is the user's first time receiving tokens (`lastInflow = 0`) via transfer. In this case, `lastInflow` becomes `senderLastInflow`.
3. It is not the user's first time receiving tokens (`lastInflow != 0`) via mint. In this case, `lastInflow` remains unchanged.
4. It is not the user's first time receiving tokens (`lastInflow != 0`) via transfer. In this case, `lastInflow = min(lastInflow, senderLastInflow)`.

Thus, if `lastInflow != 0` then minting new tokens or receiving new via transfer can only decrease `lastInflow`. As a result, the cutoff window check for newly acquired tokens will be bypassed based on the timestamp attributed to older tokens. It follows that any user who acquired tokens very early on can send 1 wei of STRX to another user to set the receiver's `lastInflow` to the same early timestamp, effectively allowing them to help other addresses pass the cutoff window check.

Proof of Concept: The following test should be added to `RewardsPendingViewsTest.t.sol`:

```
function testBrokenCutoff() public {
    vm.warp(1000);

    // Configure epoch with end time
    vm.prank(TIMELOCK);
    uint64 start = uint64(block.timestamp);
    uint64 end = start + 10000; // Long epoch
    rewards.configureEpoch(2, start, end);

    // Calculate cutoff time (30 minutes before end)
    uint64 cutoffTime = end - ANTI_SNIPE_CUTOFF;
    vm.warp(1);
    token.mint(ALICE, 100);
    // Alice buys tokens DURING cutoff window
    vm.warp(cutoffTime + 100); // 100 seconds into cutoff window

    token.mint(ALICE, 100e18);
    token.mint(BOB, 100e18);
    (,uint256 lockedUnitsAlice,,,,uint64 lastInflowAlice,,,) = rewards.getAccountFullState(ALICE);
    (,uint256 lockedUnitsBob,,,,uint64 lastInflowBob,,,) = rewards.getAccountFullState(BOB);
    console.log("Epoch Start", start);
    console.log("Current Cutoff Time", rewards.currentCutoffTime());
    console.log("Alice / Bob lastInflow", lastInflowAlice, lastInflowBob);
    console.log("Alice / Bob / difference lockedUnits", lockedUnitsAlice, lockedUnitsBob);
    console.log("");

    // Advance past the buffer
    vm.warp(cutoffTime + 100 + SETTLEMENT_BUFFER + 100);
```

```
    // Trigger settlement (replaces deleted poke())
    token.transfer(ALICE, BOB, 1);
    (, lockedUnitsAlice,,,,lastInflowAlice,,,) = rewards.getAccountFullState(ALICE);
    (, lockedUnitsBob,,,,lastInflowBob,,,) = rewards.getAccountFullState(BOB);

    console.log("Epoch Start", start);
    console.log("Current Cutoff Time", rewards.currentCutoffTime());
    console.log("Alice / Bob lastInflow", lastInflowAlice, lastInflowBob);
    console.log("Alice / Bob / difference lockedUnits", lockedUnitsAlice, lockedUnitsBob);
    console.log("");

    // Warp past epoch end (into next epoch)
    vm.warp(end + 100);

    // Trigger settlement (replaces deleted poke())
    token.transfer(ALICE, BOB, 1);

    (, lockedUnitsAlice,,,,lastInflowAlice,,,) = rewards.getAccountFullState(ALICE);
    (, lockedUnitsBob,,,,lastInflowBob,,,) = rewards.getAccountFullState(BOB);

    console.log("Epoch Start", start);
    console.log("Current Cutoff Time", rewards.currentCutoffTime());
    console.log("Alice / Bob lastInflow", lastInflowAlice, lastInflowBob);
    console.log("Alice / Bob / difference lockedUnits", lockedUnitsAlice, lockedUnitsBob);
    console.log("");
}
```

Also note that the only existing tests that in any way test the anti-snipe system are `testGetPendingLockedUnits_AntiSnipeCutoff_PausesAccrual` and `testGetPendingLockedUnits_AntiSnipeCutoff_BeforeCutoff_AccruesNormally` present in `RewardsPendingViewsTest`; however, these tests are also completely broken which means the entire system is effectively untested.

1. In `setUp()`, `deployRewardsEngine()` is called with `antiSnipeCutoffSeconds_ = 0`, which means the system is simply running a `RewardsEngine` instance with the anti-snipe mechanism disabled.
2. The test `testGetPendingLockedUnits_AntiSnipeCutoff_PausesAccrual` returns `pendingLockedUnits = 0` only because the transfer (`_settle()`) happens before `getPendingLockedUnits()` is called, whereas in `testGetPendingLockedUnits_AntiSnipeCutoff_BeforeCutoff_AccruesNormally` it is the opposite. This is the only reason the first test shows `pendingLockedUnits = 0` and the second one returns a non-zero value. These tests do not validate the cutoff system in any meaningful way.

Recommended Mitigation: The mitigation is complex and requires a redesign of the anti-snipe system. The cutoff seconds should be checked for the newly transferred balance which should be restricted from receiving rewards.

Strong: Fixed in commit b3f877f. The anti-snipe cutoff system described in this finding has been removed and replaced with a checkpoint-based eligibility system.

Cyfrin: Verified. A portion of forfeited rewards is calculated for purchases made after the epoch start but before the mid window checkpoint; otherwise, post-checkpoint purchases are made immediately ineligible.

7.2.3 Ineligible addresses are not excluded from `settledLockedUnits` calculations which locks and decreases rewards

Description: Addresses excluded through `RewardsEngine::setAccountExcluded` should not be counted towards the `settledLockedUnits` calculation of future distributions; however, `_globalSettle()` fails to remove the excluded balances from the `STRX::totalSupply` and calculates the `globalUnits` that will be unlocked based on the entire supply:

```
function _globalSettle() internal {
    ...
    uint256 totalSupply = IERC20(token_).totalSupply();
```

```
    if (totalSupply == 0) {
        lastGlobalSettlementTime = currentTime;
        return;
    }

    // Calculate units for the entire supply over the elapsed period
    // This accounts for ALL holders, including those who haven't interacted
    uint256 elapsed = uint256(effectiveEnd - lastSettlement);
    uint256 globalUnits = elapsed * totalSupply;

    if (globalUnits > 0) {
        // Only add units that aren't already in settledLockedUnits
        // Prevents double-counting while ensuring passive holders get rewards
        uint256 unitsToAdd = 0;
        if (globalUnits > settledLockedUnits) {
            unitsToAdd = globalUnits - settledLockedUnits;
        }

        // Add to global locked units (will be unlocked in distribute())
        // These are "phantom" units for passive holders only
        settledLockedUnits += unitsToAdd;
        phantomLockedUnits += unitsToAdd;
    }

    ...
}
```

This then influences the value of `settledUnlockedUnits` and ultimately `accRewardPerUnit` (the number of reward tokens per STRX) with the inflated `newUnits` being in the denominator. This decreases the `deltaIndex`, indicating a higher `totalSupply` of STRX while not all the holders are eligible, resulting in a lower `accRewardPerUnit`:

```
function distribute(uint256 couponUsdcAmount)
    external
    onlyRole(DISTRIBUTOR_ROLE)
    whenNotPaused
    returns (uint256 allocated, uint256 newDust)
{
    // Global settlement: Calculate units for ALL holders including passive ones
@>  _globalSettle(); // @audit - updates settledLockedUnits

    // First, unlock all locked units globally
    // Note: Individual account unlocking happens lazily when they interact with the contract
    uint256 lockedToUnlock = settledLockedUnits;
    if (lockedToUnlock > 0) {
        // Move all locked units to unlocked
@>      settledUnlockedUnits = settledUnlockedUnits + lockedToUnlock; // @audit-ok - unlocks all from
↪   prev distribution
        settledLockedUnits = 0;

        emit TotalUnitsUnlocked(currentEpochId, settledUnlockedUnits);
    }

    if (netCouponUsdc > type(uint256).max / USDC_TO_18) revert InvalidAmount();
    uint256 scaledCoupon = netCouponUsdc * USDC_TO_18;
    tokensFromCoupon = Math.mulDiv(scaledCoupon, PRICE_SCALE, capPrice);

@>  uint256 totalReward = tokensFromCoupon + dust;
    if (totalReward == 0) revert NothingToDistribute();

    uint256 newUnits = settledUnlockedUnits - unitsAccounted;

    {
```

```
@>      uint256 deltaIndex = Math.mulDiv(totalReward, ACC_PRECISION, newUnits); // @audit - STRX rewards
↪   / units (STRX::ts * time b/n distributions)
@>      allocated = Math.mulDiv(newUnits, deltaIndex, ACC_PRECISION); // @audit - units (STRX::ts * time
↪   b/n distributions) * deltaIndex
@>      accRewardPerUnit += deltaIndex;
    }

    unitsAccounted = settledUnlockedUnits;
    dust = totalReward - allocated;
    totalRewardsDeclared += allocated;

    // Update epoch minting counter
    if (maxTokensToMintPerEpoch > 0) {
        currentEpochTokensMinted += allocated;
    }

    lastDistributedEpochId = currentEpochId;
    lastDistributionTimestamp = uint64(block.timestamp);

    emit DistributionDeclared(currentEpochId, allocated, accRewardPerUnit, dust, grossAPYBps,
    ↪  netAPYBps);
    return (allocated, dust);
}
```

The `accRewardPerUnit` state is then used to calculate the STRX entitlement of users within 'RewardsEngine::claim. Since the calculation implicitly "reserves" rewards for excluded supply, which will never be claimable, non-excluded users receive smaller payouts than intended.

```
function claim(address recipient) external returns (uint256 amount) {
    if (recipient == address(0)) revert InvalidRecipient();

    AccountState storage state = _accounts[msg.sender];
    _settle(state, msg.sender);

    uint64 epochId = lastDistributedEpochId;
    if (epochId == 0) revert NoRewardsDeclared();

    if (claimOncePerEpoch && state.lastClaimEpoch == epochId) {
        revert ClaimTooSoon(epochId);
    }

    // Only use unlocked units for claims
    uint256 accumulated = Math.mulDiv(state.unlockedUnits, accRewardPerUnit, ACC_PRECISION);
    if (accumulated <= state.rewardDebt) revert ClaimTooSmall(0, minClaimTokens);

@>  amount = accumulated - state.rewardDebt;
    if (amount < minClaimTokens) revert ClaimTooSmall(amount, minClaimTokens);

    // Zero out unlocked units after claiming (locked units remain for next epoch)
    uint256 unitsToRemove = state.unlockedUnits;
    state.unlockedUnits = 0;
    state.rewardDebt = 0; // Reset reward debt since we're zeroing unlocked units
    state.lastClaimEpoch = epochId;

    // Update global totals to keep them accurate
    if (settledUnlockedUnits >= unitsToRemove) {
        settledUnlockedUnits -= unitsToRemove;
    } else {
        settledUnlockedUnits = 0; // Safety check
    }

    // Update unitsAccounted to reflect claimed units
    if (unitsAccounted >= unitsToRemove) {
```

```
        unitsAccounted -= unitsToRemove;
    } else {
        unitsAccounted = 0; // Safety check
    }

    totalRewardsClaimed += amount;

    _mintRewards(recipient, amount);
    emit RewardClaimed(msg.sender, recipient, amount, epochId);
}
```

Impact: Decreased rewards to eligible STRX holders due to deflated `deltaIndex`. USDC coupons forwarded as `totalRewards` also won't be entirely utilized and will only go towards increasing the collateral ratio of the STRX, effectively turning part of each coupon into a donation that overcollateralizes STRX instead of fully backing rewards.

Rewards will be deflated as long as there are excluded addresses. Given that STRX has DEX fees routed toward `LiquidityReserve` and the treasury, a large allocation of tokens into these contracts (while excluded) will significantly reduce the rewards of other holders.

Proof of Concept: The following test should be added to `RewardsEngineTest.t.sol`:

```
function testGlobalRewards() public {
    _configureTokenAndEpoch(1, 1_000, 1_000 + 30 days);
    vm.prank(TIMELOCK);
    rewards.setEarningParams(0, DEFAULT_CLAIM_ONCE);
    vm.prank(TIMELOCK);
    rewards.setMinClaimTokens(0);

    vm.warp(1_000);
    token.mint(ALICE, 90 ether);
    token.mint(BOB, 10 ether);

    vm.warp(2_000);
    vm.prank(TIMELOCK);
    rewards.setAccountExcluded(BOB, true); // Since there is no rewards bob should receive 0
    assertEq(rewards.getLockedUnits(BOB), 0);
    assertEq(rewards.getUnlockedUnits(BOB), 0);

    vm.warp(3_000);
    uint256 couponAmount = _couponForTokens(50 ether);
    vm.prank(TIMELOCK);
    rewards.distribute(couponAmount);
    console.log(rewards.accRewardPerUnit());

    uint256 alice_balance_before = token.balanceOf(ALICE);
    vm.prank(ALICE);
    rewards.claim(ALICE);
    uint256 alice_balance_after = token.balanceOf(ALICE);
    console.log(alice_balance_after - alice_balance_before);
    assertEq(alice_balance_after - alice_balance_before, 45 ether); // not 50 here
    assertEq(rewards.getLockedUnits(ALICE), 0);
    assertEq(rewards.getUnlockedUnits(ALICE), 0);
}
```

Recommended Mitigation:

1. Exclude the disabled STRX from `totalSupply` in `_globalSettle()`.
2. Extend `onBalanceChange()` to track the excluded address transfers and keep the storage consistent.

Strong: Fixed in commit b3f877f.

Cyfrin: Verified. Excluded accounts no longer contribute to eligible supply.

7.2.4 Locked/unlocked units are distributed incorrectly due to broken `phantomLockedUnits` accounting

Description: Per `Rewards::_unlockAccountUnits`, the units of a user are locked and become unlocked at the time of the next distribution:

```
function _unlockAccountUnits(AccountState storage state, address account) internal {
    // Only unlock units if this account hasn't been unlocked for the current distribution
    // This prevents new units that accumulate after distribution from being auto-unlocked
    if (distributionCount > 0 && state.lastUnlockEpoch < distributionCount) {
        // Check if locked units were added in a PREVIOUS distribution epoch
        // If lastDistributionEpoch < distributionCount, these units are eligible for unlock
        // This eliminates race condition - each account unlocks independently
        if (state.lockedUnits > 0 && state.lastDistributionEpoch < distributionCount) {
            uint256 unitsToUnlock = state.lockedUnits;
            state.unlockedUnits += unitsToUnlock;
            state.lockedUnits = 0;
            state.lastUnlockEpoch = distributionCount;

            // Global totals already updated in distribute(), just sync individual account
            // Don't update rewardDebt - newly unlocked units should claim rewards

            emit PointsUnlocked(account, unitsToUnlock, currentEpochId);
        }
    }
}
```

However, the logic also implements a concept of phantom locked units which keeps track of the same units but in the context of global settlement and is updated in the `_globalSettle()` function:

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

if (globalUnits > 0) {
    // Only add units that aren't already in settledLockedUnits
    // Prevents double-counting while ensuring passive holders get rewards
    uint256 unitsToAdd = 0;
    if (globalUnits > settledLockedUnits) {
        unitsToAdd = globalUnits - settledLockedUnits;
    }

    // Add to global locked units (will be unlocked in distribute())
    // These are "phantom" units for passive holders only
    settledLockedUnits += unitsToAdd;
    phantomLockedUnits += unitsToAdd;

    emit GlobalUnitsAccrued(currentEpochId, unitsToAdd, totalSupply, elapsed);
}
```

After `_globalSettle()` is executed, any subsequent initial call to `_settle()` for a user will allocate locked/unlocked units as follows:

```
function _addUnits(AccountState storage state, address account, uint256 newUnits) internal {
    // Handle phantom units from global settlement
    // Only apply to accounts that haven't settled since last global settlement
    uint64 lastGlobalSettlement = lastGlobalSettlementTime;
    bool hasSettledAfterGlobal = (lastGlobalSettlement > 0 && state.lastAccrual >=
    ↪  lastGlobalSettlement);

    if (phantomLockedUnits > 0 && !hasSettledAfterGlobal) {
        uint256 toRealize = phantomLockedUnits > newUnits ? newUnits : phantomLockedUnits;
        phantomLockedUnits -= toRealize;

        // Phantom units were already globally unlocked, so add directly to unlockedUnits
```

```
        // (not lockedUnits) to match the global state
        uint256 phantomUnits = toRealize;
        uint256 normalUnits = newUnits - toRealize;

        // Add phantom units directly to unlocked (they were already unlocked globally)
        if (phantomUnits > 0) {
            state.unlockedUnits += phantomUnits;
            // settledUnlockedUnits already updated in distribute()
        }

        // Add remaining normal units to locked
        if (normalUnits > 0) {
            state.lockedUnits += normalUnits;
            settledLockedUnits += normalUnits;
        }

        // Track distribution epoch for any locked units
        state.lastDistributionEpoch = distributionCount;
    } else {
        ...
    }
}
```

The current system for distributing `phantomLockedUnits` is completely broken in that it is subject to race conditions and distributes locked/unlocked units asymmetrically.

Consider the following scenario:

- User A has 90 tokens, and user B has 10 tokens.
- `EpochStart = 10`.
- At `block.timestamp = 20`, User A transfers 10 tokens to User B and triggers `_settle()` for A and B.
- Distribution occurs at `block.timestamp = 30`.
- At `block.timestamp = 40`, User A transfers another 10 tokens to User B and again triggers `_settle()` for A and B in strict order.

After this series of operations and regardless of the order, the distribution of units should be as follows:

- Total units of User A = (20 - 10) * 90 + 80 * (40 - 20).
- Locked units of User A = (40 - 30) * 80 = 800.
- Unlocked units of A = 90 * (20 - 10) + (30 - 10) * 80 = 1700.
- Total units of User B = (20 - 10) * 10 + 20 * (40 - 20).
- Locked units of User B = (40 - 30) * 20 = 200.
- Unlocked units of User B = 10 * (20 - 10) + (30 - 20) * 20 = 300.

However, the actual distribution will be as follows:

- Locked units of User A = 600; Unlocked units of User A = 1900.
- Locked units of User B = 400; Unlocked units of User B = 100.

At the same time, if `_settle()` had been called for User B before User A, the distribution would have been:

- Locked units of User A = 1000; Unlocked units of User A = 1500. Locked units of User B = 0; Unlocked units of User B = 500.

So it can be observed that in neither of the two cases is the distribution correct and it also depends on race conditions.

Impact: Incorrect distribution of locked/unlocked units allows some users to withdraw rewards earlier than others, since only unlocked units receive rewards. As a result, some users can delay the reward distribution of others while accelerating their own payout.

Proof of Concept: The following test should be added to `RewardsEngineTest.t.sol`:

```
function testGlobalRewards() public {
    _configureTokenAndEpoch(1, 1_000, 1_000 + 30 days);
    vm.prank(TIMELOCK);
    rewards.setEarningParams(0, DEFAULT_CLAIM_ONCE);
    vm.prank(TIMELOCK);
    rewards.setMinClaimTokens(0);

    vm.warp(1_000);
    token.mint(ALICE, 90 ether);
    token.mint(BOB, 10 ether);

    vm.warp(2_000);
    token.transfer(ALICE, BOB, 10 ether);// trigger settle // 80 / 20
    assertEq(rewards.getLockedUnits(ALICE), 90 ether * 1000);
    assertEq(rewards.getLockedUnits(BOB), 10 ether * 1000);
    assertEq(rewards.settledLockedUnits(), 100 ether * 1000);
    assertEq(rewards.settledUnlockedUnits(), 0);

    vm.warp(3_000);
    uint256 couponAmount = _couponForTokens(50 ether);
    vm.prank(TIMELOCK);
    rewards.distribute(couponAmount);
    assertEq(rewards.settledLockedUnits(), 0);
    assertEq(rewards.settledUnlockedUnits(), 100 ether * 2000);
    assertEq(rewards.unitsAccounted(), 100 ether * 2000);
   //assertEq(rewards.phantomLockedUnits(), 100 ether * 1000);

    vm.warp(4_000);
    bool from_alice_to_bob = true;
    if (from_alice_to_bob) {
        token.transfer(ALICE, BOB, 1); // + trigger settle
        assertEq(rewards.getLockedUnits(ALICE), 60000000000000000000000);
        assertEq(rewards.getUnlockedUnits(ALICE), 190000000000000000000000);
        assertEq(rewards.getLockedUnits(BOB), 40000000000000000000000);
        assertEq(rewards.getUnlockedUnits(BOB), 100000000000000000000000);
    } else {
        token.transfer(BOB, ALICE, 1); // + trigger settle
        assertEq(rewards.getLockedUnits(ALICE), 100000000000000000000000);
        assertEq(rewards.getUnlockedUnits(ALICE), 150000000000000000000000);
        assertEq(rewards.getLockedUnits(BOB), 0);
        assertEq(rewards.getUnlockedUnits(BOB), 500000000000000000000000);
    }
}
```

To verify that the distribution really depends on the order of `_settle()`, set the `from_alice_to_bob` variable to false.

Recommended Mitigation: The mitigation is complex and requires completely reworking the architecture of `RewardsEngine`.

Strong: Fixed in commit b3f877f.

Cyfrin: Verified. Both the sender and receiver are credited their pending rewards based on their pre-transfer balances. Post-transfer modification takes the timestamp from the `lastAccrualTime` which is updated on each account settlement.

7.2.5 The STRX reward mechanism can push the system into an undercollateralized state or drastically change the protocol configuration

Description: The current reward distribution mechanism has no protection against the fact that every mint of STRX rewards reduces the collateral ratio. This is made significantly worse by the fact that the actual price at which STRX is minted and the price at which it was obtained by the reward distributor can differ substantially.

Consider the following scenario. Suppose the system is currently in a state where `CR = 0.5`, meaning `1 STRX = 0.5 USDC`. At this moment, the reward distributor calls `RewardsEngine::distribute` with 500 USDC, allocating 1000 STRX for rewards. However, users will only be able to claim these rewards once the next epoch and next distribution occurs. Assume that by that time the price of STRX normalizes to 1 USDC. As a result, the system effectively added only 50 USDC to collateral for every 100 USDC added in STRX liability.

This creates a systematic value imbalance, directly deteriorates solvency, and can push the protocol into a structurally undercollateralized state even without any malicious behavior.

Impact: This introduces a structural solvency risk where normal reward claims can degrade CR and R/L even if the system was healthy at the time of distribution. In volatile conditions, this can push the protocol into undercollateralization, force band reconfiguration, or trigger price instability, without any malicious behavior.

Recommended Mitigation: The first step toward fixing this is to guarantee that the system does not distribute new rewards when the CAP price < 1.

However, this does not fully solve the problem, because retrospectively minting new tokens against an earlier coupon distribution can still put the system in a bad state. For example, the `V * HC` parameter might change and the system may become close to a CR < 1 state, or a large reward mint could sharply reduce the `R / L` metric and change the system's band configuration.

To protect against this, additional safeguards should be introduced to `RewardsEngine::claim`, such as an explicit check that the CR will not fall below 1 and the band configuration will remain unchanged after minting rewards.

Strong: Fixed in commits f7c7a3d, ddf5afd, and 053173b.

Cyfrin: Verified. Distributions are now blocked when CR is below 1 and a per-tx claim cap has been enforced.

7.2.6 `RewardsEngine::_settle` incorrectly accumulates rewards when claimed past the epoch end timestamp

Description: `RewardsEngine::_settle` implements reward accrual for distribution which, based on `RewardsEngine::distribute` are intended to pay out for the entire epoch duration. However, a portion of rewards claimed past the epoch end will be lost due to the last accrual timestamp being set as the current timestamp rather than the epoch end timestamp:

```
function _settle(AccountState storage state, address account) internal {
    ...

    uint64 effectiveEnd = currentTime;
    uint64 epochEnd_ = epochEnd;
    if (epochEnd_ != 0 && effectiveEnd > epochEnd_) {
@>      effectiveEnd = epochEnd_;
    }

    if (effectiveEnd <= accrualStart) {
        state.lastAccrual = currentTime;
        return;
    }

    uint256 elapsed = uint256(effectiveEnd - accrualStart);
    uint256 newUnits = elapsed * balance;
    if (newUnits != 0) {
        _addUnits(state, account, newUnits);
    }
```

```
@>  state.lastAccrual = currentTime;
}
```

Impact: STRX holders who claim after the end of an epoch will lose a portion of their rewards.

Proof of Concept: The following test should be added to `RewardsAccrualTest.t.sol`:

```
function test_MultipleEpochsAccrualAndClaiming() public {
    // Epoch 1 setup - allow time to accrue before settlement
    vm.warp(block.timestamp + SETTLEMENT_BUFFER + 1);

    // Trigger settlement to accrue units (replaces deleted poke())
    vm.prank(alice);
    strx.transfer(dave, 1e18);

    uint256 coupon = _couponForTokens(5_000e18);
    vm.prank(timelock);
    rewards.distribute(coupon);

    uint256 snapshot = vm.snapshot();

    uint256 epoch1Claimable = rewards.pendingRewards(alice);
    console.log("Epoch 1 - Alice claimable:", epoch1Claimable / 1e18);

    // Alice claims her first epoch rewards
    vm.prank(alice);
    uint256 epoch1Claimed = rewards.claim(alice);
    assertEq(epoch1Claimed, epoch1Claimable, "Should claim epoch 1 rewards");
    console.log("Epoch 1 - Alice claimed:", epoch1Claimed / 1e18);

    // Move to epoch 2 - Users continue to hold and accumulate new locked units
    vm.warp(block.timestamp + 1 days + SETTLEMENT_BUFFER + 1);

    // Trigger settlement to accumulate new locked units during the day
    vm.prank(alice);
    strx.transfer(dave, 1e18);

    // Check that Alice has new locked units
    uint256 aliceLockedUnits = rewards.getLockedUnits(alice);
    console.log("Epoch 2 - Alice locked units before distribution:", aliceLockedUnits);
    assertGt(aliceLockedUnits, 0, "Alice should have accumulated new locked units");

    // Second distribution - this will unlock the new locked units
    uint256 coupon2 = _couponForTokens(3_000e18);
    vm.prank(timelock);
    rewards.distribute(coupon2);

    uint256 epoch2Claimable = rewards.pendingRewards(alice);
    console.log("Epoch 2 - Alice claimable:", epoch2Claimable / 1e18);

    // Epoch 2 claimable should be from the newly distributed rewards
    assertGt(epoch2Claimable, 0, "Should have rewards from epoch 2");

    // Claim epoch 2 rewards
    vm.prank(alice);
    uint256 epoch2Claimed = rewards.claim(alice);
    assertEq(epoch2Claimed, epoch2Claimable, "Should claim epoch 2 rewards");

    console.log("Epoch 2 - Alice claimed:", epoch2Claimed / 1e18);
    console.log("Total claimed across epochs:", (epoch1Claimed + epoch2Claimed) / 1e18);

    vm.revertTo(snapshot);
```

```
    // Move to epoch 2 - Users continue to hold and accumulate new locked units
    vm.warp(block.timestamp + 1 days + SETTLEMENT_BUFFER + 1);

    // Claim epoch 1 rewards in epoch 2
    vm.prank(alice);
    epoch1Claimed = rewards.claim(alice);
    assertEq(epoch1Claimed, epoch1Claimable, "Should claim epoch 1 rewards");
    console.log("Epoch 1 - Alice claimed:", epoch1Claimed / 1e18);

    // Trigger settlement to accumulate new locked units during the day
    vm.prank(alice);
    strx.transfer(dave, 1e18);

    // Check that Alice has new locked units
    aliceLockedUnits = rewards.getLockedUnits(alice);
    console.log("Epoch 2 - Alice locked units before distribution:", aliceLockedUnits);
    assertGt(aliceLockedUnits, 0, "Alice should have accumulated new locked units");

    // Second distribution - this will unlock the new locked units
    coupon2 = _couponForTokens(3_000e18);
    vm.prank(timelock);
    rewards.distribute(coupon2);

    epoch2Claimable = rewards.pendingRewards(alice);
    console.log("Epoch 2 - Alice claimable:", epoch2Claimable / 1e18);

    // Epoch 2 claimable should be from the newly distributed rewards
    assertGt(epoch2Claimable, 0, "Should have rewards from epoch 2");

    // Claim epoch 2 rewards
    vm.prank(alice);
    epoch2Claimed = rewards.claim(alice);
    assertEq(epoch2Claimed, epoch2Claimable, "Should claim epoch 2 rewards");

    console.log("Epoch 2 - Alice claimed:", epoch2Claimed / 1e18);
    console.log("Total claimed across epochs:", (epoch1Claimed + epoch2Claimed) / 1e18);
}
```

Output:

```
[PASS] test_MultipleEpochsAccrualAndClaiming() (gas: 888491)
Logs:
  Epoch 1 - Alice claimable: 1428
  Epoch 1 - Alice claimed: 1428
  Epoch 2 - Alice locked units before distribution: 990781870428571183083740000
  Epoch 2 - Alice claimable: 471174
  Epoch 2 - Alice claimed: 471174
  Total claimed across epochs: 472603
  Epoch 1 - Alice claimed: 1428
  Epoch 2 - Alice locked units before distribution: 866923299000000000000000000
  Epoch 2 - Alice claimable: 412272
  Epoch 2 - Alice claimed: 412272
  Total claimed across epochs: 413701
```

Recommended Mitigation: The last accrual timestamp state updates should accurately reflect the intention to distribute full epoch rewards regardless on the actual claim timestamp.

Strong: Fixed in commit b3f877f.

Cyfrin: Verified. Rewards are now distributed and claimed by an epoch-bound, index-based mechanism.

7.2.7 Multiple small mints/refunds can bypass caps due to rounding down to zero basis points

Description: `LiquidityWindow` enforces per-epoch mint/refund limits using basis point caps:

```
function _amountToBps(uint256 amount, uint256 supply) internal pure returns (uint256) {
    if (amount == 0 || supply == 0) {
        return 0;
    }
    return (amount * BPS_DENOMINATOR) / supply;
}
```

This return value is then used by `PolicyEngine::checkMintCap` and `PolicyEngine::checkRefundCap` to enforce aggregate caps:

```
function requestMint(
    address recipient,
    uint256 usdcAmount,
    uint256 minStrxOut,
    uint256 maxEffectivePrice
)
    external
    nonReentrant
    whenNotPaused
    blockFreshCheck
    returns (uint256 strxOut, uint256 feeUsdc)
{
    ...
    uint256 amountBps = _amountToBps(strxOut, IStrcToken(strx).totalSupply());
    if (!IPolicyManager(policyManager).checkMintCap(msg.sender, amountBps)) {
        revert CapCheckFailed();
    }
    ...
    IPolicyManager(policyManager).recordMint(msg.sender, amountBps);
}
```

For sufficiently small amount relative to supply, the integer division `(amount * 10_000) / supply` rounds down to zero basis points, so the operation consumes no cap while the protocol still mints/refunds non-zero STRX.

```
function recordMint(address user, uint256 amountBps) external onlyRole(OPERATOR_ROLE) {
    uint64 dayEpoch = _currentEpoch();
    _recordCounter(_mintAggregate, amountBps, CapType.MintAggregate, address(0), dayEpoch);
}

function _recordCounter(
    RollingCounter storage counter,
    uint256 amountBps,
    CapType capType,
    address user,
    uint64 epoch
) internal {
    if (counter.epoch != epoch) {
        counter.epoch = epoch;
        counter.amountBps = 0;
        emit CapWindowReset(user, capType);
    }

    uint256 newAmount = uint256(counter.amountBps) + amountBps; // @audit add zero
    counter.amountBps = SafeCast.toUint128(newAmount);
    emit DailyLimitRecorded(capType, newAmount);
}
```

An attacker can exploit this by splitting a large mints/refunds into multiple smaller operations, each of which registers as 0 basis points and therefore does not count toward the daily caps. While the intended per-epoch refund cap

can be exceeded by chaining multiple zero basis point refunds, the supply is decreasing and so the cap shrinks. This behavior is however more problematic for mints as each operation increases the total supply. Caps are computed as `maxMint = totalSupply * mintAggregateBps / 10_000`, so as supply grows the absolute cap in STRX increases, meaning the attacker can mint an increasing amount of STRX in each epoch while still hiding a large portion of their activity behind zero basis point increments. Moreover, this allows other users to mint new STRX within the growing mint cap.

This behavior can be repeated every epoch, and the larger the STRX supply, the larger the absolute amount that can be minted in this manner. Although mint caps are initially configured as zero (effectively disabled), governance can and likely will enable them to enforce solvency/risk limits – at which point this bug becomes a real bypass of a core safety mechanism.

Impact: An attacker can repeatedly execute multiple small mint/refund operations that each register as zero basis points, thereby bypassing per-epoch caps and exceeding the intended aggregate mint/refund limits.

Proof of Concept: The following test should be added to `LiquidityWindowUSDCTest.t.sol`:

```
function testMintCapBypass() public {
    vm.startPrank(alice);
    uint256 mintUsdcAmount = 100000e6;
    usdc.approve(address(liquidityWindow), 150000e6);
    (uint256 strcReceived, uint256 mintFee) =
        liquidityWindow.requestMint(alice, mintUsdcAmount, 0, 0);
    vm.stopPrank();

    vm.startPrank(timelock);
    // Configure PolicyManager with unlimited caps for basic USDC flow testing
    PolicyManager.BandConfig memory config = policyManager.getBandConfig(PolicyManager.Band.Green);
    config.caps.mintAggregateBps = 10; // 10 / 10000 = 0.1% max
    config.caps.refundAggregateBps = 0; // 0 = unlimited refunds
    config.alphaBps = 10000; // 100% - allow unlimited refunds (alphaBps limits refund amount)
    policyManager.setBandConfig(PolicyManager.Band.Green, config);
    vm.stopPrank();

    vm.warp(block.timestamp + 1 days); // new epoch
    uint256 max_to_mint;
    uint256 totalMinted = 0;
    vm.startPrank(alice);
    mintUsdcAmount = 9e6;

    for (uint i = 0; i < 20; i++) {
        (strcReceived, ) = liquidityWindow.requestMint(alice, mintUsdcAmount, 0, 0);
        totalMinted += strcReceived;
        max_to_mint = strx.totalSupply() * config.caps.mintAggregateBps / 10000;
        console.log("MAX STRX TO MINT: ", max_to_mint / 1e18);
        console.log("Total Supply: ", strx.totalSupply() / 1e18);
        console.log("");
    }
    vm.stopPrank();
    console.log("Total Minted: ", totalMinted / 1e18);
}
```

In summary, for an initial cap `MAX STRX TO MINT 102`, 20 small mints of `9e6` USDC result in `TotalMinted 185 STRX (already > initial cap)`, increasing `totalSupply` from ~102k to ~103k and also increasing `max_to_mint` as `totalSupply` grows. All these mints pass cap checks because each small mint is rounded to 0 bps and does not consume the mint cap.

Output:

```
MAX STRX TO MINT:  102
  Total Supply:  102947

  MAX STRX TO MINT:  102
```

```
Total Supply:  102956

MAX STRX TO MINT:  102
Total Supply:  102966

MAX STRX TO MINT:  102
Total Supply:  102975

MAX STRX TO MINT:  102
Total Supply:  102984

MAX STRX TO MINT:  102
Total Supply:  102993

MAX STRX TO MINT:  103
Total Supply:  103003

MAX STRX TO MINT:  103
Total Supply:  103012

MAX STRX TO MINT:  103
Total Supply:  103021

MAX STRX TO MINT:  103
Total Supply:  103030

MAX STRX TO MINT:  103
Total Supply:  103040

MAX STRX TO MINT:  103
Total Supply:  103049

MAX STRX TO MINT:  103
Total Supply:  103058

MAX STRX TO MINT:  103
Total Supply:  103068

MAX STRX TO MINT:  103
Total Supply:  103077

MAX STRX TO MINT:  103
Total Supply:  103086

MAX STRX TO MINT:  103
Total Supply:  103095

MAX STRX TO MINT:  103
Total Supply:  103105

MAX STRX TO MINT:  103
Total Supply:  103114

MAX STRX TO MINT:  103
Total Supply:  103123

Total Minted:  185
```

Recommended Mitigation: Consider:

1. Tracking caps in absolute STRX terms rather than in basis points.
2. Introducing a minimum cap consumption for non-zero amounts. In `_amountToBps()`, enforce that any non-

zero amount consumes at least 1 bps:

```
uint256 bps = (amount * BPS_DENOMINATOR) / supply;
if (bps == 0 && amount != 0) bps = 1;
```

Or explicitly revert when amount is below a threshold that would round to 0 bps.

Strong: Fixed in commits dd3e758, b2b8f56 and 25d5461.

Cyfrin: Verified. Cap cycles are now snapshotted and enforced in absolute terms, aligned to midnight EST with admin ability to override the timezone offset.

7.3 Medium Risk

7.3.1 CAP price invariant can be violated when `CR < 1`

Description: The central CAP price invariant implies that `CAP < $1` whenever `CR < 1`:

$$\text{STRX} = \begin{cases} 1.00 \text{ USD}, & \text{if } CR \geq 1, \\ max(P_{\text{STRC}}, CR), & \text{if } CR < 1\ . \end{cases}$$

$$CR = \frac{R + (HC \cdot V)}{L}$$

This can be observed in `PolicyManager::getCAPPrice` where the returned price is intentionally capped at `1e18 - 1` when `CR < 1`:

```
function getCAPPrice() external view returns (uint256 price) {
    ...

    // If CR < 1.0, STRX = max(oraclePrice, CR), but capped below $1.00
    // Oracle price is already in 18 decimals (1e18 = $1.00)
    // Use the higher value to give users the better price when undercollateralized
    // But must ensure CAP < $1.00 when CR < 1.0 (invariant)

    // Pass staleness window to oracle health check based on CR state
    // Stressed mode (CR < 1.0): require 15min freshness
    // Healthy mode (CR >= 1.0): allow 72hr staleness
    uint256 rawCAP = cr; // Default to CR

    if (oracleAdapter != address(0)) {
        // Determine appropriate staleness window based on CR
        // When undercollateralized (CR < 1.0), oracle is critical → require 15min freshness
        // When healthy (CR >= 1.0), oracle not used → allow 72hr staleness
        uint256 maxOracleStale =
            cr < 1e18 ? STRESSED_ORACLE_STALENESS : HEALTHY_ORACLE_STALENESS;

        // Check oracle health with proper staleness window
        bool oracleHealthy = IOracleAdapter(oracleAdapter).isHealthy(maxOracleStale);

        if (oracleHealthy) {
            // Oracle is healthy, use max(oracle, CR) per architecture
            (uint256 oraclePrice,) = IOracleAdapter(oracleAdapter).latestPrice();
            rawCAP = Math.max(oraclePrice, cr);
        }
        // else: oracle unhealthy/stale, use CR only (already set above)
    }

    // If oracle or CR somehow >= $1.00, cap at $0.999999... (1e18 - 1)
    // This maintains the invariant: CAP < $1.00 when CR < 1.0
    return rawCAP >= 1e18 ? 1e18 - 1 : rawCAP;
}
```

However, this invariant is violated because the actual price used in `LiquidityWindow::requestMint` includes `halfSpreadBps` which is not taken into account in the pricing logic. This can be observed within `LiquidityWindow::_applySpread`:

```
function _applySpread(uint256 price, bool forMint, uint16 spreadBps)
    internal
    pure
    returns (uint256 effectivePrice)
{
    // Apply only the base half-spread (no additional haircut stacking)
    if (spreadBps == 0) {
```

```
        return price;
    }

    if (forMint) {
        // For mints, make token MORE expensive (user pays more)
@>      effectivePrice = (price * (BPS_DENOMINATOR + spreadBps)) / BPS_DENOMINATOR;
    } else {
        require(spreadBps <= BPS_DENOMINATOR, "spread-bounds");
        // For refunds, make token LESS valuable (user receives less)
@>      effectivePrice = (price * (BPS_DENOMINATOR - spreadBps)) / BPS_DENOMINATOR;
    }

    return effectivePrice;
}
```

Consider the following example:

- `capPrice` = 1e18 1 (0.9999999).
- `halfSpreadBps` = 10 (0.1%).
- `effectivePrice` used during minting will be 1.001.

Impact: When `CR < 1`, the CAP price invariant can be broken as it can be **greater than $1**.

Recommended Mitigation: Take the maximum possible `halfSpreadBps` into account when calculating the maximum allowable price.

Strong: Fixed in commit 47615fe. Clamped final mint price, when CR < 1, to < $1 (`1e18  1`).

Cyfrin: Verified. The invariant is now enforced in the edge cases and takes priority over maintaining the exact spread.

7.3.2 `blockFreshCheck()` modifier logic can be weaponized to block STRX mints and refunds

Description: The `blockFreshCheck()` modifier defined in `LiqudityWindow` queries `OracleAdapter::lastPriceUpdateBlock` which internally updates `_lastPriceUpdateBlock` on every call when the Pyth oracle has a newer `publishTime`:

```
modifier blockFreshCheck() {
    if (!blockFreshBypass && oracleAdapter != address(0) && blockFreshWindow > 0) {
        uint256 lastOracleBlock = IOracleAdapter(oracleAdapter).lastPriceUpdateBlock();
        if (lastOracleBlock > 0 && block.number <= lastOracleBlock + blockFreshWindow) {
            revert PriceNotFresh();
        }
    }
    _;
}

function lastPriceUpdateBlock() external returns (uint256) {
    // Check if Pyth has new price data
    if (pythContract != address(0) && pythPriceId != bytes32(0)) {
        IPyth.Price memory p = IPyth(pythContract).getPriceUnsafe(pythPriceId);

        // If Pyth has published since our last check, update block tracker
        if (p.publishTime > _lastPythPublishTime) {
            _lastPythPublishTime = p.publishTime;
            _lastPriceUpdateBlock = block.number;
        }
    }

    return _lastPriceUpdateBlock;
}
```

Note that Pyth itself publishes new **offchain** updates extremely frequently, up to as often as 400ms as claimed in the documentation. This is far below the average block times on most chains, allowing an attack to be performed relatively easily.

Since Pyth price updates can be pushed permissionlessly, a new Pyth price can be repeatedly pushed **onchain** by adversary by satisfying the following requirements:

1. Retrieve a valid signed update data from the official Pyth feed (e.g. via the Hermes or other off-chain API).
2. Call `Pyth::getUpdateFee` to calculate the fee charged by Pyth to update the price.
3. Call `Pyth::updatePriceFeeds` to update the price, paying the fee calculated in the previous step.

The `lastPriceUpdateBlock()` function can then be called which sets `lastOracleBlock` to the current `block.number`. Because the condition is `block.number <= lastOracleBlock + blockFreshWindow`, calls to `requestMint()` and `requestRefund()` will revert for all blocks in `[lastOracleBlock, lastOracleBlock + blockFreshWindow]`. With `blockFreshWindow` between 1 and 10, an attacker can keep the system in this reverting window by updating frequently enough, effectively turning the freshness check into a sustained DoS for mints and refunds.

Impact: Any address can repeatedly prevent `requestMint()` and `requestRefund()` from executing, blocking new mints and redemptions for as long as the attacker keeps pushing price updates and triggering the oracle adapter.

Recommended Mitigation:

1. If the intention is to reject stale prices instead of fresh ones, invert the check. For example:

```
if (!blockFreshBypass && oracleAdapter != address(0) && maxStaleBlocks > 0) {
  uint256 lastOracleBlock = IOracleAdapter(oracleAdapter).lastPriceUpdateBlock();
  if (lastOracleBlock == 0 || block.number > lastOracleBlock + maxStaleBlocks) {
      revert PriceStale();
  }
}
```

Considering the presence of `OracleAdapter::setFallbackPrice` which also updates `_lastPriceUpdateBlock`, it appears that the original intention was to prevent stale fallback prices from being accepted, so this recommendation must be the only one considered.

2. Decouple reads from writes in the oracle adapter:
   - Make `lastPriceUpdateBlock()` a pure getter function that does not modify state.
   - Update `_lastPriceUpdateBlock` only inside a dedicated oracle update function that is called by a trusted keeper or the protocol itself when ingesting new Pyth data.
3. Optionally, key freshness to `publishTime` instead of `block.number` (for example, `block.timestamp - lastPublishTime <= maxStaleSeconds`) while keeping the update function permissioned to protocol-controlled keepers rather than arbitrary users.

Strong: Fixed in commits 8cb6d39 and 674aee8.

Cyfrin: Verified. Block freshness validation has been removed. CAP freshness remains enforced via `PolicyManager::isHealthy` based on CR state but has been modified to use timestamps instead of block numbers.

7.3.3 STRX trading fees are charged for only a single `dexPair`

Description: `STRX::_calculateFee` determines the applicable fee for a transfer based on the STRX/USDC Uniswap V2 swap direction and fee exemptions:

```
function _calculateFee(address from, address to, uint256 amount)
    internal
    view
    returns (uint256)
{
    if (dexPair == address(0)) return 0;
```

```
    if (policyManager == address(0)) return 0; // No fees if no PolicyManager configured

    // Query PolicyManager for current DEX fees
    (uint16 buyFee, uint16 sellFee) = IPolicyManager(policyManager).getDexFees();

    if (from == dexPair) {
        if (_isFeeExempt(to)) return 0;
        return (amount * buyFee) / BPS_DENOMINATOR;
    }
    if (to == dexPair) {
        if (_isFeeExempt(from)) return 0;
        return (amount * sellFee) / BPS_DENOMINATOR;
    }
    return 0;
}
```

However, the protocol does not restrict token movement in any other Uniswap V2 pairs, nor in Uniswap V3 or V4.

Impact: Fees applicable to swaps performed through the STRX/USDC Uniswap V2 pair can be circumvented by either swapping through intermediate V2 pools or STRX/USDC pairs on Uniswap V3 or V4.

Recommended Mitigation: Consider modifying STRX to be a full fee-on-transfer token. Alternatively, consider restrict the set of pools with which STRX can interact to a predefined list, although this may be challenging given the permissionless nature of pair creation.

Strong: Acknowledged. This one is a great catch and had influence on our new architecture's method of capturing value. We don't want to make this token a fee-on-transfer because we think it will break the token's composability/attractiveness to get incorporated into other DeFi systems.

With our new model of rewards, our fees are shifting towards taking a bigger cut of the yield + breakage, less buy/sell/mint/redeem fees. So the DEX fees matter much less than they used to. At least to start, the V2 pair we set up will have the deepest liquidity which should hopefully attract the most buyers, even if there is a small DEX fee on the pool.

We don't want to remove `dexFees` from the code just so we have the option. We can add additional `dexPair` addresses if meaningful alternative V2-ish pools emerge.

Cyfrin: Acknowledged.

7.3.4 `LiquidityWindow::requestRefund` treasurer fee avoids USDC withdrawal delay from `LiquidityReserve`

Description: Based on the `README.md` and developer comments of `LiquidityReserve::queueWithdrawal`, treasurer withdrawals of USDC from the `LiquidityReserve` contract should always execute with a delay:

> `queueWithdrawal` auto-executes LiquidityWindow refunds or enqueues multi-hour withdrawals for treasury roles.

```
/// @dev LiquidityWindow calls for instant withdrawals (fast path for refunds)
/// @dev Treasurer/admin calls for tiered withdrawals (delayed based on amount)
function queueWithdrawal(address to, uint256 amount) external nonReentrant whenNotPaused {
    // CHECKS: Validate inputs
    if (to == address(0)) revert InvalidAddress();
    if (amount == 0) revert InvalidAmount();

    // LiquidityWindow fast path: instant withdrawal for refunds
    if (msg.sender == liquidityWindow && to != treasurer) {
        _instantWithdrawal(to, amount);
        return;
    }

   ...
    _enqueueWithdrawal(to, amount, msg.sender);
```

```
}
```

However, the treasurer implicitly bypasses this USDC withdrawal delay with every execution of the `LiquidityWindow::requestRefund` in which users redeem STRX for USDC. The output USDC is withdrawn from `LiquidityReserve` in the instant withdrawal path of the `LiquidityReserve::queueWithdrawal` call:

```
function requestRefund(
    address recipient,
    uint256 strxAmount,
    uint256 minUsdcOut,
    uint256 minEffectivePrice
)
    external
    nonReentrant
    whenNotPaused
    blockFreshCheck
    returns (uint256 usdcOut, uint256 feeUsdc)
{
    ...
    ILiquidityReserve(liquidityReserve).queueWithdrawal(address(this), grossUsdc);

    // Step 2: Now contract has USDC, route fees properly to Reserve/Treasury split
    _routeFees(feeUsdc);

    // Step 3: Send net amount to recipient
    IERC20(usdc).safeTransfer(recipient, usdcOut);

    ...
}
```

While the user's withdrawal is executed immediately, as expected, the output USDC is not sent to the user in its entirety – a portion is taken as a fee and distributed between `LiquidityReserve` and the `treasurer`. As a result, the portion received by the `treasurer` will effectively be instantly withdrawn from `LiquidityReserve`.

```
// Splits collected fees between reserve and treasury based on the configured percentage.
function _routeFees(uint256 feeUsdc) internal {
    if (feeUsdc == 0) return;
    if (usdc == address(0)) return; // Skip if USDC not configured

    // Split fees between Reserve and Treasury according to feeToReservePct
    uint256 toReserve = (feeUsdc * feeToReservePct) / BPS_DENOMINATOR;
    uint256 toTreasury = feeUsdc - toReserve;

    // Verify contract has sufficient USDC balance
    uint256 balance = IERC20(usdc).balanceOf(address(this));
    require(balance >= feeUsdc, "Insufficient USDC for fee routing");

    // Send Reserve portion to Reserve
    if (toReserve > 0) {
        IERC20(usdc).safeTransfer(liquidityReserve, toReserve);
        ILiquidityReserve(liquidityReserve).recordDeposit(toReserve);
    }

    // Send Treasury portion directly to Treasury
    if (toTreasury > 0) {
        IERC20(usdc).safeTransfer(treasury, toTreasury);
    }
}
```

Moreover, the current fee structure is broken by multiple issues associated with fee distribution and refunds:

1. On minting STRX, a portion of the `feeBps` is sent to the treasury while the remainder is sent with the net

USDC to the `LiquidityReserve`; however, the USDC allocated as a reserve fee is not separately tracked and can be erroneously withdrawn by redemptions.

2. On refunding STRX, a portion of the `feeBps` is sent directly to the treasury, while the remainder is returned to the `LiquidityReserve`. In reality, the reserve fee should either be separately tracked or forwarded to a dedicated brokerage address so it can be used in the offchain STRC purchases.

There is currently no such dedicated function to move reserve fee funds, meaning that `queueWithdrawal()` must be used to enqueue an admin withdrawal. Such a withdrawal is not permitted to be instantly executed and will be subject to a lockup ranging from 12-36 hours, but this is undesirable as the reserve fee funds are intended to be swept for buying STRC through the offchain brokerage account. The bonds issue yield once per month, so one mistimed late withdrawal could cost an entire months' worth of earnings.

Impact: Instead of explicitly withdrawing USDC from `LiquidityReserve` with a delay, the treasurer can obtain the same amount from one or more `LiquidityWindow::requestRefund` calls over a shorter period of time. Furthermore, fee distribution is incorrect and can result in loss of bond yield if the purchase deadline is missed.

Recommended Mitigation: The withdrawal mechanism should be redesigned such that:

1. Treasurer fees from STRX refunds are subject to a delay, if required. Note that the pending USDC to be withdrawn by queued refunds should be tracked and subtracted from CR computations; otherwise, burning STRX without discounting the corresponding USDC balance will falsely increase the collateral ratio.
2. Reserve fee allocation tracks USDC in the `LiquidityReserve` contract.
3. An optional new function is added for the brokerage account to withdraw the reserve fees.

Strong: Fixed in commit 23c6106. Resolved by enabling instant withdrawals for `TREASURER_ROLE`. The treasurer is a protocol-controlled wallet with operational requirements—bond purchase deadlines require immediate USDC transfers to brokerage, and queued withdrawals don't really serve us anymore. Because of this we don't want to put any restrictions on how much Treasurer can withdrawal from LR. Since this wallet is behind a 4/4 MPC Fireblocks approval wallet we think it's fine.

Cyfrin: Verified. Instant withdrawals are now enabled for the Treasurer.

7.3.5 Collateral ratio is calculated with mismatched values in both USDC and USD terms

Description: `CollateralAttester::V` is supplied in USD terms and scaled to 18 decimals. Given that 1 USDC is not strictly pegged to 1 USD, this will result in a mismatch between values in USD and scaled USDC terms when calculating the collateral ratio. The following logic implicitly assumes 1 USDC will always be redeemable for 1 USD and vice versa:

```
function getCollateralRatio() public view returns (uint256) {
    uint256 L = IERC20(strxToken).totalSupply();
    if (L == 0) return type(uint256).max; // Infinite CR when no supply

    // Count ONLY reserve USDC as collateral (treasury is protocol profit)
@>  uint256 reserveUSDC = IERC20(usdc).balanceOf(liquidityReserve);
    uint256 R = reserveUSDC;

    // CR = (R + HC×V) / L
    // All in 18 decimals: R (scaled from reserveAssetDecimals), HC (18 decimals), V (18 decimals), L
    ↪  (18 decimals)
    uint256 scaledR = _scaleToEighteen(R, reserveAssetDecimals);
@>  uint256 haircutValue = Math.mulDiv(HC, V, PRECISION);
    uint256 numerator = scaledR + haircutValue;

    return Math.mulDiv(numerator, PRECISION, L);
}
```

However, this is not strictly true and will become especially problematic if the price of USDC in USD were to depeg. While the USD value of the offchain reserves would not be subject to such a depeg event, onchain USDC reserves

would be affected. Therefore, to perform an accurate calculation of the collateral ratio, the USDC reserve should first be quoted in dollar terms.

The same issue is present in `PolicyManager::_computeCurrentSnapshot`. The result of this function should be a coefficient, i.e., a value without a unit of measurement; however, since `navPerToken` gives the price in USDC/STRX, the current unit of measurement for L/R is USDC/USD which implies a 1:1 peg:

```
// Query on-chain state directly (NO TRUST REQUIRED)
uint256 totalSupply = IStrcToken(strxToken).totalSupply();
// Scale USDC balance from 6 decimals to 18 decimals for compatibility with L (liabilities)
// This ensures accurate reserve ratio and floor calculations
uint256 reserveBalance = IERC20(usdc).balanceOf(liquidityReserve) * 1e12;
(uint256 navPerToken,) = IOracleAdapter(oracleAdapter).latestPrice();
uint256 lastOracleBlock = IOracleAdapter(oracleAdapter).getLastPriceUpdateBlock();

// Calculate L (total liability in dollars)
// L = totalSupply * NAV (scaled properly)
uint256 L =
    totalSupply == 0 || navPerToken == 0 ? 0 : Math.mulDiv(totalSupply, navPerToken, 1e18);

// Calculate R/L (reserve ratio)
uint16 reserveRatioBps =
    L == 0 ? 0 : uint16(Math.mulDiv(reserveBalance, BPS_DENOMINATOR, L));
```

Impact: The collateral ratio will be overstated in the event of a USDC depeg event.

Recommended Mitigation: Consider integrating a USDC/USD oracle to calculate the collateral ratio in dollar terms.

Strong: Agree and accepting. We assume USDCUSD when computing CR/R/L (V is USD-scaled, R is raw USDC), which could overstate CR in a USDC depeg.

We'll document the peg assumption now and plan a v2 fix to thread a USDC/USD feed through CollateralAttestation and PolicyManager.

Cyfrin: Acknowledged.

7.3.6 Missing band refresh in `RewardsEngine::distribute` results in incorrect `distributionSkimBps`

Description: `RewardsEngine::distribute` allocates USDC rewards. The value returned by `PolicyManager::getDistributionSkimBps` is the percentage of rewards directed to the treasurer, determined separately for each band defined in `PolicyManager`; however, `RewardsEngine::distribute` fails to call `PolicyEngine::refreshBand` at the beginning of its execution which means that `getDistributionSkimBps()` may execute with an outdated config:

```
// Distribution skim BPS (coupon skim to treasury/reserve).
function getDistributionSkimBps() external view returns (uint16) {
    BandConfig memory config = _resolveActiveConfig();
    return config.distributionSkimBps;
}
```

This function simply retrieves the config for the most recently set band, but this band may already be outdated. The inline comments of `refreshBand()` explicitly state that it should be called at the beginning of a transaction to ensure an up-to-date evaluation, and other functions in the protocol such as `LiquidityWindow::requestMint` also follow this pattern.

```
/// @notice Autonomous band refresh - recalculates band from live on-chain state
/// @dev Call this at the start of each transaction for accurate, gas-efficient band updates
/// @dev Uses cached _band for subsequent reads within the same transaction
/// @return newBand The updated band (GREEN/YELLOW/RED/EMERGENCY)
function refreshBand() external returns (Band newBand) {
    // Query live on-chain state (totalSupply, reserve balance, oracle price)
    SystemSnapshot memory snapshot = _computeCurrentSnapshot();
```

```
    // Skip band evaluation if system is uninitialized (zero supply = no liabilities yet)
    // Keep current band during bootstrap
    if (snapshot.totalSupply == 0) {
        return _band;
    }

    // Evaluate band from current reserve ratio
    (Band evaluated, string memory reason) = _evaluateBand(snapshot);

    // Update cached band if it changed
    if (evaluated != _band) {
        Band previous = _band;
        _band = evaluated;
        lastBandEvaluation = uint64(block.timestamp);
        emit BandChanged(previous, evaluated, reason);
    }

    return _band;
}
```

Impact: An incorrect value of `distributionSkimBps` may be used when executing `RewardsEngine::distribute`.

Recommended Mitigation: Call `refreshBand()` at the beginning of `RewardsEngine::distribute`.

Strong: Fixed in commit 8b94ef2.

Cyfrin: Verified. `PolicyManager::refreshBand` is now called within `RewardsEngine::distribute`.

7.3.7 `RewardsEngine::distribute` checks CAP price before withdrawing treasurer funds

Description: During the execution of the `RewardsEngine::distribute`, the current CAP price is fetched via a call to `PolicyManager::getCAPPrice` which itself calls `CollateralAttestation::getCollateralRatio` and uses the USDC balance of the `LiquidityReserve` directly:

```
function getCollateralRatio() public view returns (uint256) {
    uint256 L = IERC20(strxToken).totalSupply();
    if (L == 0) return type(uint256).max; // Infinite CR when no supply

    // Count ONLY reserve USDC as collateral (treasury is protocol profit)
    uint256 reserveUSDC = IERC20(usdc).balanceOf(liquidityReserve);
    uint256 R = reserveUSDC;

    // CR = (R + HC×V) / L
    // All in 18 decimals: R (scaled from reserveAssetDecimals), HC (18 decimals), V (18 decimals), L
    ↪  (18 decimals)
    uint256 scaledR = _scaleToEighteen(R, reserveAssetDecimals);
    uint256 haircutValue = Math.mulDiv(HC, V, PRECISION);
    uint256 numerator = scaledR + haircutValue;

    return Math.mulDiv(numerator, PRECISION, L);
}
```

As can be observed from the inline comments, it is important to account for only the USDC balance belonging to the `LiquidityReserve` and not to the treasurer. However, the call to `PolicyManager::getCAPPrice` is executed before the portion of USDC belonging to the treasurer is withdrawn from the `LiquidityReserve` through execution of `withdrawDistributionSkim()`.

```
 address policy = policyManager;
if (policy == address(0)) revert InvalidConfig();
IPolicyDistributionConfig config = IPolicyDistributionConfig(policy);

uint256 capPrice = config.getCAPPrice();
```

```
if (capPrice == 0) revert InvalidOraclePrice();

// Withdraw skim after CAP price validation but before amount calculations
// This ensures getCAPPrice() sees the reserve before skim, which is acceptable
// because the coupon has already been deposited (reserve is healthy)
if (skimUsdc > 0) {
    ILiquidityReserve(liquidityReserve).withdrawDistributionSkim(treasury, skimUsdc);
    emit DistributionSkimCollected(currentEpochId, skimUsdc, skimBps, treasury);
}
```

This results in a contradiction whereby calculation of the CR takes a portion of the treasurer's distributed USDC balance, held by the `LiquidityReserve`, into account.

Recommended Mitigation: Call `getCAPPrice()` after `withdrawDistributionSkim()`.

Strong: Fixed in commit a9c3acc. We now withdraw the skim before fetching CAP, so CR excludes treasury USDC as intended.

Cyfrin: Verified.

7.3.8 `LiquidityReserve` withdrawal queue may be manipulated to enforce maintaining a specific CR and R/L

Description: `LiquidityReserve` uses a single USDC balance to serve two fundamentally different withdrawal flows:

1. Instant withdrawals via `LiquidityWindow` for regular users (STRX is burned in exchange for USDC).
2. Queued withdrawals for trusted roles (treasurer/admin), processed with delays via tiered queues.

Both the collateral ration `CR` in `CollateralAttestation::getCollateralRatio` and reserve ratio `R/L` in `PolicyManager::_computeCurrentSnapshot` are computed directly from the `USDC` balance of `LiquidityReserve`. However, instant withdrawals do not reduce `CR` or `R/L`, because they simultaneously burn `STRX`. In contrast, queued withdrawals directly reduce `CR` and `R/L`, as they only decrease the reserve balance.

Because both flows share the same liquidity pool without isolation, a user can strategically perform instant withdrawals to temporarily block queued withdrawals, delaying the moment when `CR` and `R/L` are allowed to decrease. This creates a situation in which the reserve depletion can be artificially postponed onchain, while price-sensitive mechanisms still operate under outdated solvency assumptions.

Impact: A large holder can delay treasury withdrawals that would push CR below 1 and reprice STRX. This allows artificial maintenance of CR and CAP price. The risk scales with the total STRX supply and Uniswap V2 pair liquidity.

Recommended Mitigation: Consider implementing separate accounting for instant and queued withdrawals.

Strong: Fixed in commit 23c6106. Resolved by enabling instant withdrawals for `TREASURER_ROLE`. The treasurer is a protocol-controlled wallet with operational requirements—bond purchase deadlines require immediate USDC transfers to brokerage, and queued withdrawals don't really serve us anymore. Since this wallet is behind a 4/4 MPC Fireblocks approval wallet we think it's fine.

Cyfrin: Verified. The treasurer can now perform instant withdrawals.

7.3.9 Pausing of `RewardsEngine::distribute` also delays users claiming rewards

Description: `RewardsEngine::distribute` is guarded by the `whenNotPaused` modifier and can be paused by the admin; however, unlocking of rewards is strictly coupled to successful execution of distributions via `_unlockAccountUnits()` which only unlocks `lockedUnits` after `distributionCount` is incremented. As a result, when `distribute()` is paused, newly accumulated `lockedUnits` can not be unlocked until unpaused:

```
function _unlockAccountUnits(AccountState storage state, address account) internal {
    // Only unlock units if this account hasn't been unlocked for the current distribution
    // This prevents new units that accumulate after distribution from being auto-unlocked
```

```
@>  if (distributionCount > 0 && state.lastUnlockEpoch < distributionCount) {
        // Check if locked units were added in a PREVIOUS distribution epoch
        // If lastDistributionEpoch < distributionCount, these units are eligible for unlock
        // This eliminates race condition - each account unlocks independently
        if (state.lockedUnits > 0 && state.lastDistributionEpoch < distributionCount) {
            uint256 unitsToUnlock = state.lockedUnits;
            state.unlockedUnits += unitsToUnlock;
            state.lockedUnits = 0;
            state.lastUnlockEpoch = distributionCount;

            // Global totals already updated in distribute(), just sync individual account
            // Don't update rewardDebt - newly unlocked units should claim rewards

            emit PointsUnlocked(account, unitsToUnlock, currentEpochId);
        }
    }
}
```

Impact: Since rewards are accounted per epoch rather than per distribution, newly accrued units within the same epoch are also entitled to receive their share of earlier distributions from that same epoch. When `distribute()` is paused, `distributionCount` does not advance and locked units cannot be unlocked. As a result, users are unable to realize the rewards from previous distributions within the current epoch to which their newly accrued units are already entitled.

Recommended Mitigation: Consider decoupling the unlocking of rewards from the distribute execution path. At a minimum, introduce an alternative unlock mechanism that is not blocked by `whenNotPaused`, or allow `_unlockAccountUnits()` to progress based on epoch transitions instead of strictly relying on successful subsequent distributions.

Strong: Fixed in commit b3f877f. The coupling between pause and "unlock" belonged to the old locked/unlocked model of `RewardsEngine`. The new index-based implementation decouples claim from distribute; pausing distribute delays new allocations but does not delay claiming previously accrued entitlements.

Cyfrin: Verified. The `whenNotPaused` modifier has bee removed from `RewardsEngine::claim` and the new distribution mechanism allows claims to be made for previous epochs.

7.4 Low Risk

7.4.1 Increasing the `KYCRegistry` epoch does not invalidate old proofs

Description: `KYCRegistry` uses epoch-based proofs, however `epoch` is not included in the leaf hash which means that epoch rotation does not prevent replay. Old proofs are not automatically invalidated when the epoch is increased and so can be reused by simply passing the new epoch with same proof.

This breaks implicit assumptions documented by developer comments in both `KYCRegistry::registerWithProof` and `KYCRegistry::setRoot`:

```
// Epoch must strictly increase so users can't replay old proofs.
function setRoot(bytes32 newRoot, uint64 epoch) external onlyAttestor {
    ...
}

// Rejects stale epochs, double registrations, and invalid proofs to keep the set tight.
function registerWithProof(bytes32[] calldata proof, uint64 epoch) external whenNotPaused {
    ...
}
```

Impact: Increasing the epoch will not invalidate previously valid proofs. If the attestor increases the epoch but intentionally retains the same `merkleRoot` (e.g. with the intention of invalidating a set of stale proofs without recomputing the tree), users can still reuse their old proofs by simply passing the new epoch. Epoch-based replay protection is therefore non-functional.

Proof of Concept: The following test should be added to `KYCRegistryTest.t.sol` and run with `forge test --match-test testSameRootNewEpochAcceptsOldProof`:

```
function testSameRootNewEpochAcceptsOldProof() public {
    bytes32 leaf1 = keccak256(abi.encodePacked(USER1));
    bytes32 leaf2 = keccak256(abi.encodePacked(USER2));
    bytes32 root = _computeRoot(leaf1, leaf2);

    // Set root at epoch 1
    vm.prank(TIMELOCK);
    registry.setRoot(root, 1);

    // Build proof for USER1 against epoch 1 root (leaf2 is sibling)
    bytes32[] memory proof = new bytes32[](1);
    proof[0] = leaf2;

    // Rotate epoch but keep the same root (epoch increases)
    vm.prank(TIMELOCK);
    registry.setRoot(root, 2);

    // USER1 presents the same proof but uses the current epoch (2)
    vm.prank(USER1);
    registry.registerWithProof(proof, 2);

    assertTrue(registry.isAllowed(USER1));
}
```

Recommended Mitigation: While it is understood that the intention is to always update both the root and the epoch simultaneously, consider including the epoch in the leaf such that it can be used to invalidate stale proofs:

```
function registerWithProof(bytes32[] calldata proof, uint64 epoch) external whenNotPaused {
        require(merkleRoot != bytes32(0), "KYCRegistry: root not set");
        require(epoch == currentEpoch, "KYCRegistry: stale epoch");
        require(!_allowed[msg.sender], "KYCRegistry: already allowed");

-       bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
+       bytes32 leaf = keccak256(abi.encodePacked(msg.sender, epoch));
```

```
        require(MerkleProof.verifyCalldata(proof, merkleRoot, leaf), "KYCRegistry: invalid proof");

        _allowed[msg.sender] = true;
        emit KYCUpdated(msg.sender, true, epoch);
    }
```

Strong: Fixed in commits 47615fe and a8feadd.

Cyfrin: Verified. The documention has been improved, explicitly mentioning that epoch and merkle root must be updated together. The `epoch` parameter has also been removed altogether given that it currently serves no purpose. Transfers to/from disallowed addresses have additionally been blocked.

7.4.2 Multisig owner is unlikely able to maintain fresh oracle fallback prices when `CR < 1`

Description: At times when `CR < 1`, oracle prices must be updated at least every 15 minutes; otherwise, they will be rejected by the `OracleAdapter::isHealthy` check during the CAP price calculation within `PolicyManager::getCAPPrice`:

```
if (oracleAdapter != address(0)) {
    // Determine appropriate staleness window based on CR
    // When undercollateralized (CR < 1.0), oracle is critical → require 15min freshness
    // When healthy (CR >= 1.0), oracle not used → allow 72hr staleness
    uint256 maxOracleStale =
        cr < 1e18 ? STRESSED_ORACLE_STALENESS : HEALTHY_ORACLE_STALENESS;

    // Check oracle health with proper staleness window
    bool oracleHealthy = IOracleAdapter(oracleAdapter).isHealthy(maxOracleStale);

    if (oracleHealthy) {
        // Oracle is healthy, use max(oracle, CR) per architecture
        (uint256 oraclePrice,) = IOracleAdapter(oracleAdapter).latestPrice();
        rawCAP = Math.max(oraclePrice, cr);
    }
    // else: oracle unhealthy/stale, use CR only (already set above)
}
```

The oracle provides price data from two sources: Pyth prices and an owner-specified fallback price that is used in critical moments when the Pyth price is unavailable. According to the inline developer comments, the owner address is expected to be a multisig. When `CR < 1`, both prices must be updated regularly to keep the system in a correct and consistent state; however, it is very difficult for a multisig owner to update the fallback price every 15 minutes. In practice, it is highly unlikely that the multisig will be able to maintain such a high update frequency.

Since the CAP price is determined as the maximum between the CR-based price and the oracle price, the absence of a fresh oracle price at a given moment can cause a sudden and significant shift in the computed CAP price.

Impact: The fallback price will most likely be inaccurate during periods when `CR < 1`.

Recommended Mitigation: Consider introducing a new trusted role responsible for updating the fallback price in critical moments.

Strong: Fixed in commits 0efcc65 and 9efa6b8.

Cyfrin: Verified.

7.4.3 `LiquidityReserve::setLiquidityWindow` does not revoke the `DEPOSITOR_ROLE` from outdated addresses

Description: `LiquidityReserve::initialize` grants the `DEPOSITOR_ROLE` to `LiquidityWindow` which is needed to call `LiquidityReserve::recordDeposit`.

The stored `liquidityWindow` address can be changed by the owner with a call to `LiquidityReserve::setLiquidityWindow`; however, the `DEPOSITOR_ROLE` is not revoked.

```
// Repoints the module handling deposits/instant refunds; auto-grants depositor permissions.
function setLiquidityWindow(address newWindow) external onlyRole(ADMIN_ROLE) {
    if (newWindow == address(0)) revert InvalidAddress();
    liquidityWindow = newWindow;
    _grantRole(DEPOSITOR_ROLE, newWindow);
    isRecoverySink[newWindow] = true;
    emit LiquidityWindowSet(newWindow);
    emit RecoverySinkSet(newWindow, true);
}
```

Even if it were desired to have two active `LiquidityWindow` contracts, the `recordDeposit()` function has an additional check to ensure that only the currently set `liquidityWindow` can transfer the assets:

```
function recordDeposit(uint256 amount) external {
    if (!hasRole(DEPOSITOR_ROLE, msg.sender)) revert NotAuthorized();
    if (amount == 0) revert InvalidAmount();

    if (msg.sender != liquidityWindow) {
        asset.safeTransferFrom(msg.sender, address(this), amount);
    }
    emit DepositRecorded(msg.sender, amount);
}
```

Recommended Mitigation: Revoke the `DEPOSITOR_ROLE` from the previously configured address within `LiquidityReserve::setLiquidityWindow`.

Strong: Fixed in commit 1f0723a.

Cyfrin: Verified.

7.4.4 `PolicyManager::refreshBand` fails to update derived caps

Description: Unlike `PolicyManager::reportSystemSnapshot`, `PolicyManager::refreshBand` does not currently invoke `_refreshDerivedCaps()`. The expectation is for this function to be called at the beginning of each transaction for accurate band updates; however, this omission means that external view functions such as `PolicyManager::getDerivedCaps` and `PolicyManager::getAggregateRemainingCapacity` will return stale values that do not correctly correspond to the fresh snapshot data from which updated caps should have been derived.

Impact: Changes to capacity cap configurations associated with band updates will not be reflected by external view functions as intended.

Recommended Mitigation: Invoke `_refreshDerivedCaps()` within `PolicyManager::refreshBand` after the updated band has been evaluated.

Strong: Fixed in commit 8b94ef2.

Cyfrin: Verified. `_refreshDerivedCaps()` is now invoked within `PolicyManager::refreshBand`.

7.4.5 Tiered delay of large withdrawals can change between enqueuing and execution

Description: The `LiquidityReserve` uses a tiered withdrawal system that is calculated once at enqueue time based on the current reserve asset (USDC) balance:

```
function _determineTier(uint256 amount) internal view returns (uint8 tier, uint64 releaseAt) {
    uint256 balance = asset.balanceOf(address(this));
    uint256 amountBps =
        balance > 0 ? Math.mulDiv(amount, BPS_DENOMINATOR, balance) : BPS_DENOMINATOR;

    uint64 nowTs = uint64(block.timestamp);

    if (amountBps <= tierImmediate.maxAmountBps) {
        return (0, nowTs + tierImmediate.delaySeconds);
```

```
    }
    if (amountBps <= tierDelayed.maxAmountBps) {
        return (1, nowTs + tierDelayed.delaySeconds);
    }
    return (2, nowTs + tierSlow.delaySeconds);
}
```

The returned value is then stored in the withdrawal request and never recalculated at execution time. If multiple other legitimate large withdrawals drain the reserve before the withdrawal is executed, it will be possible for a queued withdrawal to be executed with an incorrect tier/delay.

Consider the following scenario:

1. The initial reserve balance is 10,000,000 USDC.
2. A user queues 150,000 USDC withdrawal: amountBps = 150 BPS (1.5%) → Tier 1, 12-hour delay.
3. Legitimate activity from other users drains the reserve balance to 500,000 USDC.
4. After 12 hours, the 150,000 USDC withdrawal is executed.
5. Actual impact: `150,000 / 500,000 = 30%` of remaining liquidity withdrawn with only a 12-hour delay.

Large withdrawals can therefore be queued (coordinated or otherwise) when the reserve is large but executed when it is depleted.

Impact: Tiered delay of large withdrawals can change between enqueuing and execution which may allow bad actors to bypass intended protections.

Recommended Mitigation: Consider recalculation of the tier at execution time to check whether it has changed. Alternatively, consider simplifying the withdrawal delay to a single flat value that affords sufficient time to react to potential bad actors.

Strong: Fixed in commits 23c6106 and c38394f.

Cyfrin: Verified. Treasury withdrawals are now instant and the tiered admin delay has been modified to a flat 24 hour period.

7.4.6 Missing condition in `PolicyManager::syncOracleStrictMode` event monitoring logic

Description: `PolicyManager::syncOracleStrictMode` emits different events for monitoring purposes depending on the previous value of the `_isInDeficit` state and whether oracle strict mode should be enabled based on the updated CR. The following conditional logic is present to handle the cases when the CR crosses the 1.0 threshold:

```
// Emit monitoring events when CR crosses 1.0 threshold
if (shouldBeStrict && !_isInDeficit) {
    // Entering undercollateralization: CR just crossed below 1.0
    uint256 deficit = 1e18 - cr; // How much below 1.0 we are
    emit CollateralDeficit(cr, deficit);
    _isInDeficit = true;
} else if (!shouldBeStrict && _isInDeficit) {
    // Exiting undercollateralization: CR just crossed back above 1.0
    emit RecollateralizationComplete(cr);
    _isInDeficit = false;
} else if (shouldBeStrict) {
    // Already in deficit, emit continuous monitoring event
    uint256 deficit = 1e18 - cr;
    emit CollateralDeficit(cr, deficit);
}
```

However, note the absence of the `if (!shouldBeStrict && !_isInDeficit)` condition. This can occur, for example, during the bootstrapping phase, assuming sufficient USDC reserves and correct price oracle configuration, since the `_isInDeficit` state will initialize as false by default. Similarly, once the system is operational, there will be no monitoring events emitted when `PolicyManager::syncOracleStrictMode` is called.

Given this represents `CR >= 1.0` and not previously marked as in deficit, i.e. the expected healthy steady state, it is arguably correct to omit this branch; however, it may still be preferable to emit such a "system is healthy" event to confirm both initial and sustained health state when the system has CR 1.0 and the oracle strict mode is synced.

Impact: `PolicyManager::syncOracleStrictMode` could be considered incomplete for monitoring purposes.

Recommended Mitigation: Consider explicitly handling the case where the system remains fully collateralized and was not previously in a deficit, emitting an event for monitoring purposes.

Strong: Acknowledged; will do in v2.

Cyfrin: Acknowledged.

7.4.7 `OracleAdapter::setFallbackPrice` allows any arbitrary fallback price to be set

Description: `OracleAdapter::setFallbackPrice` currently allows the owner to configure any arbitrary fallback price that will be used in the event the Pyth oracle fails. To mitigate centralization concerns and avoid sudden stepwise changes, it may be desirable to implement validation to enforce min/max bounds or some deviation check against the last known Pyth price.

Recommended Mitigation: Consider adding validation to `OracleAdapter::setFallbackPrice` to constrain the provided fallback price.

Strong: Acknowledged.

Cyfrin: Acknowledged.

7.4.8 `OracleAdapter::_tryPyth` may reject high confidence prices due to overly restrictive validation

Description: `OracleAdapter::_tryPyth` will return zero if the scaled confidence interval is zero or greater than the maximum allowed threshold confidence:

```
@>  if (pythMaxConf != 0 && p.conf > 0) {
        uint256 scaledConf = _scalePythConfidence(p);
@>      if (scaledConf == 0 || scaledConf > pythMaxConf) {
            return (0, 0);
        }
    }
```

However, this means that the price will be rejected if `p.conf > 0` but `scaledConf == 0` due to precision loss in `_scalePythValue()`. This can occur, for instance, when dividing very tight confidence by `exp < 0` and is in fact desirable as a tight confidence interval means there is high certainty in the reported price.

```
function _scalePythValue(uint256 value, int32 expo) private pure returns (uint256) {
    int256 exp = int256(expo) + 18;
    if (exp > 60 || exp < -60) return 0;

    if (exp >= 0) {
        ...
    } else {
        uint256 pow = 10 ** uint256(-exp);
        if (pow == 0) return 0;
@>      return value / pow;
    }
}
```

Impact: High confidence prices may be rejected due to overly restrictive validation.

Recommended Mitigation:

```
if (pythMaxConf != 0 && p.conf > 0) {
    uint256 scaledConf = _scalePythConfidence(p);
-   if (scaledConf == 0 || scaledConf > pythMaxConf) {
+   if (scaledConf > pythMaxConf) {
```

```
        return (0, 0);
    }
}
```

Strong: Fixed in commit d0fd6d4. `_tryPyth()` now accepts extremely tight confidence intervals that scale to zero and only rejects when `scaledConf > pythMaxConf`.

Cyfrin: Verified.

7.4.9 `PolicyManager::requiresGovernanceVote` use `_lastSnapshot` instead of current snapshot

Description: `PolicyManager::requiresGovernanceVote` is needed for the UI to signal an emergency state; however, it does not use the current snapshot but instead uses `_lastSnapshot` which will signal an incorrect state.

```
// Signals when reserve ratio has fallen into emergency territory (for UI/governance).
function requiresGovernanceVote() external view returns (bool) {
    return _lastSnapshot.reserveRatioBps <= _reserveThresholds.emergencyBps;
}
```

Recommended Mitigation: Compute the current snapshot before reporting state.

Strong: Fixed in commit 4235bd4.

Cyfrin: Verified.

7.4.10 STRX charges fees on providing liquidity in the `dexPair`

Description: The protocol charges a fee on transfers between the DEX pair and the user, as can be observed in `STRX::_calculateFee`:

```
function _calculateFee(address from, address to, uint256 amount)
    internal
    view
    returns (uint256)
{
    if (dexPair == address(0)) return 0;
    if (policyManager == address(0)) return 0; // No fees if no PolicyManager configured

    // Query PolicyManager for current DEX fees
    (uint16 buyFee, uint16 sellFee) = IPolicyManager(policyManager).getDexFees();

    if (from == dexPair) {
        if (_isFeeExempt(to)) return 0;
        return (amount * buyFee) / BPS_DENOMINATOR;
    }
    if (to == dexPair) {
        if (_isFeeExempt(from)) return 0;
        return (amount * sellFee) / BPS_DENOMINATOR;
    }
    return 0;
}
```

It is intended that fees will be charged when buying or selling STRX in the STRX/USDC Uniswap V2 pool; however, the protocol does not account for the fact that interactions with the pair occur not only during swap operations but also during liquidity modification. As a result, the current fee model will also charge fees from LP providers. Therefore, providing liquidity to such a pair may become unprofitable.

Sell taxes reduce LP fee earnings since LPs receive less USDC and volume is lower, while buys aren't taxed on USDC and therefore generate higher fee capture. This asymmetry is inherent to such tokens and should be expected by LPs.

Recommended Mitigation: Either avoid charging fees from LPs during liquidity modification or document this behavior clearly.

Strong: Acknowledged since we are moving to a low fee approach. We'll document it and be clear with our users if we turn on the DEX fees.

Cyfrin: Acknowledged.

7.4.11 Rewards may be lost as `RewardsEngine::configureEpoch` cannot always be updated at the correct time

Description: Rewards are accumulated for the entire interval between `epochStart` and `epochEnd`. However, once an epoch ends, a new one must be created via `RewardsEngine::configureEpoch`. Given that this is an admin-only function, there is no guarantee that it will be called at the exact moment `epochEnd` is reached, or even within the same block due to operational constraints.

Impact: All deposits made after the `epochEnd` but *before* the admin configures the new epoch will **not accrue any rewards**, even if the new `epochStart` is equal to the old `epochEnd`.

Proof of Concept: Consider the following scenario:

- `epochStart` = 1000
- `epochEnd` = 2000

Any deposit with a timestamp `>` `epochEnd`, while `epochStart` has not yet been updated, will move `state.lastAccrual` forward. Then, when the new epoch is configured, rewards will incorrectly start from this new `state.lastAccrual` even if earlier deposits should have been counted.

Time	Action	lastAccrual (before _settle())	lastAccrual (after _settle())	Balance (after action)	Rewards rec
1200	Mint 1(100 tokens)	0	1200	100	No
1950	Mint 2(50 tokens)	1200	1950	150	Yes (75,000
2050	Mint 3(50 tokens)	1950	2050	200	Yes (7,500 u
2100	Mint 4(20 tokens)	2050	2100	220	No (dead zo

If `epochStart` is later updated to `2000` after these mints occur, the rewards for the interval `[2000, 2100]` will not be accrued. Rewards will only begin accruing from `2100` onward, since that timestamp becomes the new `lastAccrual` during the next mint.

Recommended Mitigation: Consider changing the `epochStart` / `epochEnd` logic entirely. A better approach is to use a mapping and a struct to store multiple future epochs. This allows the protocol to pre-schedule upcoming epochs and switch between them inside `_settle()` based on `block.timestamp` and the current `epochEnd`.

Strong: Fixed in commit b3f877f. The new rewards engine caps all accrual to the epoch window via `_capped-Timestamp()`. Interactions after `epochEnd` don't advance time beyond `epochEnd`. When the next epoch is configured, `_settleAccount()` resets `lastAccrualTime` to the new `epochStart`. On the next interaction, it accrues from `epochStart` to "now," so time between epochs is included. Piecewise balance changes within that gap are compressed. If someone mints at `T > epochEnd` but before configure, the later settle may treat more of the interval as if the larger balance existed since `epochStart`, slightly over-crediting that account for the gap. This effect is bounded by how late `configureEpoch` is called and disappears if you configure on time.

Cyfrin: Verified.

7.4.12 `LiquidityReserve` and treasury will accumulate STRX rewards they cannot claim

Description: `LiquidityReserve` and `treasury` are not excluded by default from the `STRX` rewards in `RewardsEngine` which will accumulate but cannot be claimed. This will happen because `STRX` charges a DEX fee that is distributed on each token transfer that incurs them:

```
function _update(address from, address to, uint256 value) internal override {
    if (from == address(0)) {
        // mint - KYC is enforced in LiquidityWindow before calling this
        super._update(from, to, value);
        _notifyRewards(address(0), to, value);
        return;
    }

    if (to == address(0)) {
        // burn
        super._update(from, to, value);
        _notifyRewards(from, address(0), value);
        return;
    }

    uint256 feeAmount = _calculateFee(from, to, value);
    if (feeAmount > 0) {
        uint256 netAmount = value - feeAmount;
        super._update(from, to, netAmount);
@>      _distributeFees(from, feeAmount);
        // Notify RewardsEngine with net amount actually transferred
        _notifyRewards(from, to, netAmount);
    } else {
        super._update(from, to, value);
        _notifyRewards(from, to, value);
    }
}
```

```
function _distributeFees(address source, uint256 feeAmount) internal {
    // CHECKS: Calculate distribution and validate addresses upfront
    uint256 toReserve = (feeAmount * feeToReservePct) / BPS_DENOMINATOR;
    uint256 toTreasury = feeAmount - toReserve;

    // Validate addresses BEFORE any state changes
    if (toReserve > 0 && liquidityReserve == address(0)) revert InvalidAddress();
    if (toTreasury > 0 && treasury == address(0)) revert InvalidAddress();

    // EFFECTS: All balance updates together (atomic state changes)
    // Both transfers complete before any external calls
    if (toReserve > 0) {
        super._update(source, liquidityReserve, toReserve);
    }
    if (toTreasury > 0) {
        super._update(source, treasury, toTreasury);
    }

    // INTERACTIONS: All external calls after state is finalized
    // Even if RewardsHook reenters here, all balances are already updated
    //@audit-issue: liquidityReserve and treasury will accumulate rewards also, but they shouldn't and
    ↪  must be avoided
    if (toReserve > 0) {
@>      _notifyRewards(source, liquidityReserve, toReserve);
    }
    if (toTreasury > 0) {
@>      _notifyRewards(source, treasury, toTreasury);
    }
}
```

The problematic function is `_distributeFees()` because it calls `_notifyRewards()` for both addresses:

```
function _notifyRewards(address from, address to, uint256 amount) internal {
    if (rewardsHook == address(0)) return;
```

```
    IRewardsHook(rewardsHook).onBalanceChange(from, to, amount);
}
```

```
function onBalanceChange(address from, address to, uint256 amount) external onlyToken {
    // Skip processing for the RewardsEngine itself
    // The rewards contract should not accumulate units for its own balance
    if (from == address(this) || to == address(this)) {
        return;
    }

    // Capture sender's lastInflow BEFORE processing outflow
    // This ensures recipient inherits the correct mint time for anti-snipe
    uint64 senderLastInflow = 0;
    if (from != address(0)) {
        senderLastInflow = _accounts[from].lastInflow;
        _handleOutflow(from, amount);
    }

    if (to != address(0)) {
@>      _handleInflow(to, amount, senderLastInflow);
    }
}
```

For both addresses, `_handleInflow()` will be invoked, because they are the recipients of the `STRX` fee.

Note the existence of `RewardsEngine::setAccountExcluded` which is not currently used to exclude the above-mentioned contracts on initialization, which leaves a window for rewards to accumulate. Furthermore, there is no evidence in the deployment script for excluding the system contracts from incurring rewards.

Impact: Rewards will remain locked due to missing functionality in the `LiquidityReserve` and potentially also the treasury for claiming the `STRX` rewards accumulated from the DEX fee inflows.

Recommended Mitigation: Exclude all the system contracts from STRX rewards.

Strong: Acknowledged. Added this to our deployment steps to exclude system addresses from accrual operationally post-wiring.

Cyfrin: Acknowledged.

7.4.13 `CollateralAttestation::V` and Pyth oracle configuration should be initialized upon deployment

Description: `CollateralAttestation::HC` is initialized at deployment time whereas `V` is not. This means that the off-chain collateral contributes nothing to the CR until the attestor publishes the first attestation. Based on `isAttestationStale()`, this seems to be intentional and indeed `PolicyManager::getCAPPrice` explicitly handles the bootstrapping phase by assuming a CR of 1. However, `getCollateralRatio()` called within `syncOracleStrictMode()` has no such checks, meaning that the CR will be computed based on the USDC reserves and STRX supply only.

The impact appears to be limited to incorrect oracle `strictMode` sync and monitoring event emissions. If the Pyth oracle is not yet configured and no fallback price is set then `strictMode` is set on an oracle that cannot provide any price. This in turn means that `getCAPPrice()` falls back to CR only which will be zero until the first attestation is published. At this point, the system is in an inconsistent state where `strictMode` implies a healthy oracle is critical as CR is in a deficit but the oracle in fact cannot provide prices and monitoring events are emitted based on incomplete data.

Impact: Monitoring events may not be emitted as expected.

Recommended Mitigation: Similar to the other bootstrapping behavior, consider skipping the oracle sync if no attestation has been published or no oracle/fallback pricing has been configured. Alternatively, ensure that all critical components will be deployed and configured atomically, ideally setting the initial `V` and Pyth configuration within the initializer.

Strong: Fixed in commit 7fd146f.

Cyfrin: Verified.

7.4.14 Users who send STRX directly to `RewardsEngine` will continue to receive yield on their balance

Description: `RewardsEngine::onBalanceChange`, called inside `STRX::_update` (i.e., on every STRX token transfer), skips all processing when either the sender or recipient address is the `RewardsEngine` contract itself:

```
function onBalanceChange(address from, address to, uint256 amount) external onlyToken {
    // Skip processing for the RewardsEngine itself
    // The rewards contract should not accumulate units for its own balance
@>  if (from == address(this) || to == address(this)) {
        return;
    }

    // Capture sender's lastInflow BEFORE processing outflow
    // This ensures recipient inherits the correct mint time for anti-snipe
    uint64 senderLastInflow = 0;
    if (from != address(0)) {
        senderLastInflow = _accounts[from].lastInflow;
@>      _handleOutflow(from, amount);
    }

    if (to != address(0)) {
@>      _handleInflow(to, amount, senderLastInflow);
    }
}
```

However, the system maintains state for every address, including tracking the current STRX balance. Updates to the balance field occur inside `RewardsEngine::_handleOutflow` and `RewardsEngine::_handleInflow`:

```
function _handleOutflow(address account, uint256 amount) internal {
    if (amount == 0) return;

    AccountState storage state = _accounts[account];
    _settle(state, account);

    uint256 balance = state.balance;
    if (balance < amount) revert BalanceUnderflow();
    state.balance = balance - amount;
}

function _handleInflow(address account, uint256 amount, uint64 senderLastInflow) internal {
    if (amount == 0) return;

    AccountState storage state = _accounts[account];
    _settle(state, account);

    state.balance += amount;

    if (state.lastInflow == 0) {
        state.lastInflow = senderLastInflow != 0 ? senderLastInflow : uint64(block.timestamp);
    } else if (senderLastInflow != 0 && senderLastInflow < state.lastInflow) {
        state.lastInflow = senderLastInflow;
    }
}
```

When tokens are transferred directly to `RewardsEngine`, `_handleOutflow()` is not executed for the sender address. Similarly, when tokens are transferred from `RewardsEngine`, `_handleInflow()` is not executed for the recipient address (though this scenario is currently unreachable).

As a result, users who send their tokens to `RewardsEngine` (likely by accident) will continue receiving rewards based on an unchanged balance, even though their actual STRX balance has decreased.

Impact: Users who accidentally or otherwise send their STRX to `RewardsEngine` will receive rewards for an inflated balance.

Recommended Mitigation: Consider either calling the handler functions in both these cases.

Strong: Fixed in commit b732b08. We no longer return early when the `RewardsEngine` address is involved; we process the sender's side so their internal balance is decremented and accrual stops.

Cyfrin: Verified.

7.4.15 `OracleAdapter::_scalePythValue` exponent validation should mirror the canonical Pyth Solidity SDK

Description: Pyth stores prices as a 64 bit integer significand and an 32 bit integer exponent. `OracleAdapter::_scalePythValue` uses a threshold value of `+/- 60` when considering the exponent. However, in the canonical Pyth Solidity SDK `+/- 58` is used instead:

```
// Bounds check: prevent overflow/underflow with base 10 exponentiation
// Calculation: 10 ** n <= (2 ** 256 - 63) - 1
//              n <= log10((2 ** 193) - 1)
//              n <= 58.2
if (deltaExponent > 58 || deltaExponent < -58)
    revert PythErrors.ExponentOverflow();
```

While intermediate overflow is not possible based on the current implementation and upcasting behavior, it is still recommended to mirror this behavior.

Recommended Mitigation:

```
function _scalePythValue(uint256 value, int32 expo) private pure returns (uint256) {
    int256 exp = int256(expo) + 18;
-   if (exp > 60 || exp < -60) return 0;
+   if (exp > 58 || exp < -58) return 0;

    ...
}
```

Strong: Fixed in commit 4235bd4.

Cyfrin: Verified.

7.4.16 Oracle staleness validation is unnecessarily convoluted and conflicting

Description: The intention of `OracleAdapter::lastPriceUpdateBlock`, called within the `LiquidityWindow::blockFreshCheck` modifier, is to prevent mints and refunds from being made in the same block as a price oracle updates in strict mode. This function is assumed to be called from `LiquidityWindow` but can also be invoked by any arbitrary address. In fact, this behavior is necessary as the reverting path causes a self-DoS vector in which the `lastPriceUpdateBlock` state change will be rolled back whenever an update price has been posted. Thus the first transaction after any Pyth update will always revert, requiring `OracleAdapter::lastPriceUpdateBlock` to be manually invoked after which subsequent transactions for `blockFreshWindow` blocks will also be prevented. Rather than remaining with this forward looking behavior, it may be more preferable to revert transactions relying on price updates that were made more than a given number of block prior, inverting the semantics of the validation and using the currently unused `maxStaleBlocks` in place of `blockFreshWindow`.

In general, the implementation of price oracle staleness validation is unnecessarily convoluted and in certain places overlapping. For example, `OracleAdapter::_tryPyth` implements its own staleness check against `pythStaleAfter` but `isHealthy()` performs additional staleness validation based on the `maxStale` value passed in from `PolicyManager::getCAPPrice`. This can either be 15 minutes or 72 hours depending on the CR, so it will work fine when `CR >= 1.0` as 72 hours is very likely much larger than any normal oracle staleness period. However, when CR is below the threshold this can result in the scenario where `pythStaleAfter` is larger than `maxStale`.

The tests configure a `pythStaleAfter` value of 1 day, so in this scenario the Pyth price could pass `_tryPyth()` but fail `isHealthy()`.

```
function isHealthy(uint256 maxStale) external view returns (bool) {
    // When not in strict mode (CR  1.0), oracle health doesn't matter
    // CAP price = $1.00 regardless of oracle state
    if (!strictMode) {
        return true;
    }

    // In strict mode (CR < 1.0), oracle MUST be fresh for CAP pricing
    (uint256 price, uint256 updatedAt) = this.latestPrice();
    if (price == 0 || updatedAt == 0) return false;
    return block.timestamp <= updatedAt + maxStale;
}

function latestPrice() external view returns (uint256 price, uint256 updatedAt) {
    (price, updatedAt) = _tryPyth();
    if (price != 0) return (price, updatedAt);

    return (_fallbackPrice, _fallbackUpdatedAt);
}
```

Consider the following scenario:

- `CR < 1e18`.
- `pythStaleAfter = 1 day`.
- Assume the Pyth price was updated 20 hours ago.
- Assume fallback price was updated 10 minutes ago.
- `OracleAdapter::isHealthy` validation will succeed for the fallback price only, but during the execution of `OracleAdapter::latestPrice` the Pyth price will still be returned, even though that price would not pass the final `OracleAdapter::isHealthy` check.

Furthermore, `oracleStaleSeconds` present in the `PolicyManager` snapshot logic and `bandConfigs` (which is never validated) is also superfluous and instead it would be better to rely only on `HEALTHY/STRESSED_ORACLE_STALENESS` in all instances to resolve the ambiguity.

Impact: Multiple separate but overlapping implementations of oracle staleness validation is unnecessarily convoluted and conflicts with the intended behavior.

Recommended Mitigation: Rework and simplify this validation as suggested above such that all staleness checks are performed given a shared configuration.

Strong: Fixed in commits 8cb6d39 and 674aee8.

Cyfrin: Verified. Block freshness validation has been removed. CAP freshness remains enforced via `PolicyManager::isHealthy` based on CR state but has been modified to use timestamps instead of block numbers.

7.4.17 `RewardsEngine::setEarningParams` called during epoch may alter existing reward distribution

Description: `RewardsEngine::setEarningParams` allows the admin change `antiSnipeCutoffSeconds` at any time. This value is referenced from within `_settle()` via `_currentCutoff()` and applied to historical `lastInflow` timestamps.

```
function setEarningParams(uint32 antiSnipeCutoffSeconds_, bool claimOncePerEpoch_)
    external
    onlyRole(ADMIN_ROLE)
{
    antiSnipeCutoffSeconds = antiSnipeCutoffSeconds_;
    claimOncePerEpoch = claimOncePerEpoch_;
}
```

```
function _settle(AccountState storage state, address account) internal {
    ...
    uint64 lastInflow = state.lastInflow;
    uint64 cutoff = _currentCutoff();
    if (cutoff != 0 && lastInflow != 0 && lastInflow >= cutoff) {
        uint64 epochEndCheckpoint = epochEnd;
        if (epochEndCheckpoint > accrualStart) {
            accrualStart = epochEndCheckpoint;
        }
    }
    ...
}

function _currentCutoff() internal view returns (uint64) {
    uint64 epochEnd_ = epochEnd;
    uint32 cutoffWindow = antiSnipeCutoffSeconds;
    if (epochEnd_ == 0 || cutoffWindow == 0) return 0;
    if (epochEnd_ <= cutoffWindow) return 0;
    return epochEnd_ - cutoffWindow;
}
```

As such, if the admin increases the cutoff window during an active epoch, deposits that were originally outside the anti-snipe window can suddenly be treated as toxic flow and earn zero units for that epoch. If the window is decreased, deposits that were originally blocked can suddenly become eligible and accrue units, depending only on when `_settle()` is called relative to the configuration change.

Impact: Retroactively changing `antiSnipeCutoffSeconds` mid-epoch changes which deposits are eligible for rewards in that epoch. Users who deposited expecting to earn based on the old window can lose their entire epoch's accrual if the window is enlarged before they settle. Conversely, shrinking the window can make previously "sniped" deposits suddenly eligible and shift rewards toward those accounts. Two users with identical deposit times can end up with different rewards purely based on whether `_settle()` was executed before or after an admin configuration change, which breaks the "same behavior → same rewards" expectation and gives governance a lever to arbitrarily reallocate yield inside an ongoing epoch.

Recommended Mitigation: Make anti-snipe parameters epoch-scoped instead of global and time-varying. A simple approach is to snapshot `antiSnipeCutoffSeconds` when `configureEpoch` is called (for example, `epochCutoffSeconds[epochId]`) and use that fixed value in `_currentCutoff()` for the entire epoch. Alternatively, allow `setEarningParams()` to update `antiSnipeCutoffSeconds` only for the next epoch (store it as "pending" and apply it on execution of `configureEpoch()` or enforce that updates can only happen after epochEnd).

Strong: Fixed in commit b3f877f. The anti-snipe mechanism has been removed in favor of a new rewards contract with a checkpoint system instead.

Cyfrin: Verified.

7.4.18 `RewardsEngine::setAccountExcluded` called within a STRX pause period will cause prior rewards to be lost

Description: The call to `_mintRewardsSafe()` executed within `Rewards::setAccountExcluded` implies that minting to an account may fail due to the recipient lacking KYC. This can be inferred from the inline developer comments, as well as from the dedicated `RewardsKYCExclusionTest.t.sol` test file covering this scenario.

```
// Try to mint rewards - if it fails (e.g., KYC check), forfeit them
try this._mintRewardsSafe(account, claimableAmount) {
    // Success - rewards were minted
    totalRewardsClaimed += claimableAmount;
    emit RewardClaimed(
        account, account, claimableAmount, lastDistributedEpochId
    );
} catch {
    // Mint failed (likely KYC check) - forfeit rewards and update debt
```

```
    // This prevents the user from claiming these rewards later if they re-KYC
    state.rewardDebt = accumulated;
    emit RewardsForfeited(
        account, claimableAmount, "KYC check failed during exclusion"
    );
}
```

However, the absence of KYC is not the only reason why `STRX::mint` may revert. Considering the scenario in which STRX is in a paused state, any call to `_mintRewardsSafe()` will revert. In this situation, any call to `setAccountExcluded()` will cause the prior rewards of all users, even if they are KYC-approved, be lost. If such behavior is undesirable, the owner effectively loses the ability to call `setAccountExcluded()` altogether.

Recommended Mitigation: Consider explicitly handling different revert reasons on the `_mintRewardsSafe()` external call.

Strong: Fixed in commit b3f877f.

Cyfrin: Verified. The new exclusion logic avoids any attempt at minting STRX and simply only updates the struct responsible for reward accounting.

7.4.19 Rewards should be claimable once per distribution rather than once per epoch

Description: Currently, there can be multiple reward distributions within each epoch. Locked units are transferred into the unlocked state at the moment of the next `distribute()` function call, rather than at the moment of the epoch change. Converting units into the unlocked state automatically unlocks the rewards which can be seen in `_unlockAccountUnits()`:

```
function _unlockAccountUnits(AccountState storage state, address account) internal {
    // Only unlock units if this account hasn't been unlocked for the current distribution
    // This prevents new units that accumulate after distribution from being auto-unlocked
    if (distributionCount > 0 && state.lastUnlockEpoch < distributionCount) {
        // Check if locked units were added in a PREVIOUS distribution epoch
        // If lastDistributionEpoch < distributionCount, these units are eligible for unlock
        // This eliminates race condition - each account unlocks independently
        if (state.lockedUnits > 0 && state.lastDistributionEpoch < distributionCount) {
            uint256 unitsToUnlock = state.lockedUnits;
            state.unlockedUnits += unitsToUnlock;
            state.lockedUnits = 0;
            state.lastUnlockEpoch = distributionCount;

            // Global totals already updated in distribute(), just sync individual account
            // Don't update rewardDebt - newly unlocked units should claim rewards

            emit PointsUnlocked(account, unitsToUnlock, currentEpochId);
        }
    }
}
```

However, the reward claim logic is incorrectly implemented as once per epoch instead of once per distribution. This inconsistency may confuse users and by blocking rewards across multiple distributions until the epoch changes:

```
function claim(address recipient) external returns (uint256 amount) {
      if (recipient == address(0)) revert InvalidRecipient();

      AccountState storage state = _accounts[msg.sender];
      _settle(state, msg.sender);

      uint64 epochId = lastDistributedEpochId;
      if (epochId == 0) revert NoRewardsDeclared();

      if (claimOncePerEpoch && state.lastClaimEpoch == epochId) {
          revert ClaimTooSoon(epochId);
```

```
        }
    ...
}
```

Recommended Mitigation: Consider changing claims to be allowed once per distribution.

Strong: Fixed in commit b3f877f. The new implementation doesn't gate by epoch anymore; this is effectively resolved by the refactor.

Cyfrin: Verified. Both `pendingRewards` and `unitsAccrued` increase only after the epoch is resolved and are zeroed on claim.

7.4.20 Use of `LiquidityReserve` balance of USDC overstates solvency by ignoring pending withdrawals

Description: The collateral ratio (`CR = (R + HC × V) / L`) relies solely on the raw USDC balance of `LiquidityReserve` without accounting for queued withdrawal obligations. As a result, the reported reserve can significantly overstate the real amount of USDC that is economically available to back STRX redemptions.

An identical structural issue exists for R/L-based policy logic, where reserve-based limits and safety bands are derived from the same inflated balance. This allows the system to appear solvent and liquid while a large portion of the reserve is already allocated for future withdrawals.

This creates a systematic solvency misreporting risk, where both CR and R/L remain artificially high despite the reserve being partially locked into obligations that cannot be used to satisfy new refunds.

Recommended Mitigation: Replace the raw `liquidityReserve.balanceOf()` dependency with an effective reserve value, defined as `effectiveReserve = actualBalance  pendingWithdrawals`.

Strong: Fixed in commit 23c6106. Believe instant treasury withdrawals mitigate this. There are still delayed admin withdrawals but those are rare or never.

Cyfrin: Verified.

7.4.21 Pyth confidence interval is not applied to pricing mints/refunds

Description: According to Pyth best practices, protocols should use the confidence interval to derive a conservative price range (price ± conf), applying the lower bound for collateral valuation and the upper bound for debt valuation, to protect against oracle uncertainty during volatile or abnormal market conditions.

In the current implementation, `LiquidityWindow::requestMint` and `LiquidityWindow::requestRefund` simply only validate the confidence interval against a maximum acceptable threshold and apply `halfSpreadBps` which functions as a fee mechanism rather than as a protection against oracle inaccuracy. The oracle price itself is never adjusted using the confidence interval, meaning the protocol always operates on the raw midpoint price even when oracle uncertainty is elevated, which deviates from Pyth's recommended defensive pricing model.

Recommended Mitigation: When deriving the effective oracle price, apply an equivalent conservative adjustment strategy based on the confidence interval. This logic should be enforced in addition to the existing maximum confidence threshold, not as a replacement.

Strong: Acknowledged. We already cap max conf and enforce freshness under CR<1. Applying ±conf changes user UX and requires broader plumbing. Will consider a strictMode CI clamp in v2.

Cyfrin: Acknowledged.

7.4.22 Effective mint price calculated in `LiquidityWindow::requestMint` should be rounded up

Description: The effective mint price calculated within `LiquidityWindow::requestMint` is performed by invocation of `_applySpread()` which uses integer division rounding the price down after applying `halfSpreadBps`. Since `strxOut` is computed as `netAmount / effectivePrice`, this rounding may slightly increases the amount of STRX minted compared to the intended price.

```
function _applySpread(uint256 price, bool forMint, uint16 spreadBps)
    internal
    pure
    returns (uint256 effectivePrice)
{
    // Apply only the base half-spread (no additional haircut stacking)
    if (spreadBps == 0) {
        return price;
    }

    if (forMint) {
        // For mints, make token MORE expensive (user pays more)
        effectivePrice = (price * (BPS_DENOMINATOR + spreadBps)) / BPS_DENOMINATOR;
    } else {
        require(spreadBps <= BPS_DENOMINATOR, "spread-bounds");
        // For refunds, make token LESS valuable (user receives less)
        effectivePrice = (price * (BPS_DENOMINATOR - spreadBps)) / BPS_DENOMINATOR;
    }

    return effectivePrice;
}
```

Recommended Mitigation: Consider rounding the effective mint price up when applying the spread (e.g., via ceiling division) so that `strxOut` is computed against a conservatively higher price.

Strong: Fixed in commit d8ed3f0.

Cyfrin: Verified. The effective mint price is now rounded up.

7.5 Informational

7.5.1 `LiquidityWindow::setUSDC` should be removed in favor of setting `usdc` within `LiquidityWindow::initialize`

Description: `LiquidityWindow::setUSDC` allows the owner to configure the `usdc` address only once after initialization:

```
function setUSDC(address _usdc) external onlyOwner {
    require(_usdc != address(0), "Invalid USDC");
    require(usdc == address(0), "USDC already set");
    usdc = _usdc;
}
```

However, this can be removed in favor of setting `usdc` inside `LiquidityWindow::initialize`.

Strong: Acknowledged.

Cyfrin: Acknowledged.

7.5.2 Unchained initializers should be called instead

Description: While not an immediate issue in the current implementation, the direct use of initializer functions rather than their unchained equivalents should be avoided. `CollateralAttestation::initialize`, `LiquidityReserve::initialize`, `LiquidityWindow::initialize`, `PolicyManager::initialize`, `RewardsEngine::initialize`, and `STRX::intiialize` should be modified to avoid potential duplicate initialization in the future.

Recommended Mitigation: Consider using unchained initializers in `CollateralAttestation::initialize`, `LiquidityReserve::initialize`, `LiquidityWindow::initialize`, `PolicyManager::initialize`, `RewardsEngine::initialize`, and `STRX::intiialize`.

Strong: Accepted. Going to defer to a V2.

Cyfrin: Acknowledged.

7.5.3 Reconfiguration of STRX modules does not remove fee exemption of old addresses

Description: `STRX::configureModules` allows the owner to set the new `liquidityWindow`, `liquidityReserve`, and `treasury` addresses. This function also calls `_syncSystemFeeExemptions()` to exempt each of these addresses from fees; however, it fails to revoke this state for old addresses.

```
function _syncSystemFeeExemptions() internal {
    if (liquidityWindow != address(0)) {
        _setFeeExemptInternal(liquidityWindow, true);
    }
    if (liquidityReserve != address(0)) {
        _setFeeExemptInternal(liquidityReserve, true);
    }
    if (treasury != address(0)) {
        _setFeeExemptInternal(treasury, true);
    }
}
```

Recommended Mitigation: Consider revoking the fee exemption state of old addresses by invoking `_setFeeExemptInternal()` with `false`.

Strong: Fixed in commit b699240.

Cyfrin: Verified.

7.5.4 Incorrect block time assumptions in `PolicyManager::_computeCurrentSnapshot`

Description: It is intended for the protocol to be deployed on multiple chains; however, `oracleStaleSeconds` incorrectly assumes a constant block time of 12s on both L1 and L2. Most L2 chains have significantly lower block times and even on `Ethereum` mainnet the block time is not guaranteed to be constant.

```
function _computeCurrentSnapshot() internal view returns (SystemSnapshot memory snapshot) {
    ...
    // Calculate oracle staleness (blocks since last update * 12s avg block time)
    uint32 oracleStaleSeconds = uint32((block.number - lastOracleBlock) * 12);
}
```

Note that the `oracleStaleness` is never used or validated against the corresponding member of the `bandConfigs` instance of the `BandConfig` struct defined in `PolicyManager`.

Impact: If the actual block time is less than 12s, stale prices will be accepted as valid, since blocks passed per the calculation will be less than actual blocks passed on the chain. For the example of a 250ms block time on Arbitrum, 48 blocks will pass in 12 seconds.

Recommended Mitigation: If this functionality is not removed as redundant, consider passing the average block time as an `initializer` parameter.

Strong: Acknowledged. Will change in V2.

Cyfrin: Acknowledged.

7.5.5 `Band.Emergency` config cannot be enabled without `PolicyManager` contract upgrade

Description: The `Band.Emergency` config cannot currently be used because it is always being overridden by `Band.Red`. This means that the entire configuration is missing, and even when there is a table for the risk profile, it will not be available without upgrading the `PolicyManager`.

As can be observed below, `_resolveActiveConfig()`, `getRefundParameters()`, and `getMintParameters()` all override `Band.Emergency` with the `Band.Red` config:

```
// Convenience helper to grab the current band's config struct.
function _resolveActiveConfig() internal view returns (BandConfig memory active) {
    active = _bandConfigs[_band];
    if (_band == Band.Emergency) {
        // Emergency inherits RED spreads/fees until we design a distinct table.
        active = _bandConfigs[Band.Red];
    }
    return active;
}

function getRefundParameters(address user, uint256 amountBps)
    external
    view
    returns (MintParameters memory params)
{
    params.capPrice = this.getCAPPrice();
    params.currentBand = _band;
    BandConfig memory config = _bandConfigs[params.currentBand];

    // Handle Emergency band (inherits RED config)
    if (params.currentBand == Band.Emergency) {
        config = _bandConfigs[Band.Red];
    }

function getMintParameters(address user, uint256 amountBps)
    external
    view
    returns (MintParameters memory params)
```

```
{
    params.capPrice = this.getCAPPrice();
    params.currentBand = _band;
    BandConfig memory config = _bandConfigs[params.currentBand];

    // Handle Emergency band (inherits RED config)
    if (params.currentBand == Band.Emergency) {
        config = _bandConfigs[Band.Red];
```

Impact: The `Band.Emergency` configuration is unreachable even when configured:

```
// Update spreads/fees/caps for a specific band; banned for Emergency to keep logic simple.
function setBandConfig(Band band, BandConfig calldata config) external onlyRole(ADMIN_ROLE) {
    if (band == Band.Emergency) revert InvalidConfig();
    _validateBandConfig(config);
    _bandConfigs[band] = config;
    emit BandConfigUpdated(
        band,
        config.halfSpreadBps,
        config.mintFeeBps,
        config.refundFeeBps,
        config.oracleStaleSeconds,
        config.deviationThresholdBps,
        config.alphaBps,
        config.floorBps,
        config.distributionSkimBps,
        config.caps.mintAggregateBps,
        config.caps.refundAggregateBps,
        uint64(block.timestamp)
    );
}
```

Recommended Mitigation: If the intention is to never have a `Band.Emergency` config, then it can be removed. Otherwise, if it is disabled until the exact profile is defined, `PolicyManager` should be reworked such that `setBandConfig()` does not revert and `_resolveActiveConfig()`, `getRefundParameters()`, and `getMintParameters()` check a flag as to whether the config should be used.

Strong: Fixed in commits 8d0ecf0 and 644db67.

Cyfrin: Verified. `Band.Emergency` has been removed.

7.5.6 `LiquidityWindow::_getCAPPrice` is unused and can be removed

Description: `LiquidityWindow::_getCAPPrice` is currently unused. Instead, the CAP price is returned as a parameter from the call to `PolicyManager::getMintParameters` which is then used to determine USDC amounts for minting and refunding through `_applySpread()`.

It is understood that all `PolicyManager` queries are batched into `getMintParameters()` for the purpose of gas efficiency. As such, the unused `_getCAPPrice()` should be removed.

Strong: Fixed in commit 4f0904d.

Cyfrin: Verified. `LiquidityWindow::_getCAPPrice` has been removed.

7.5.7 System addresses cannot bypass the KYC check within `LiquidityWindow`

Description: The implementation of the `LiquidityWindow::_enforceKYC` check, used to verify the caller and recipient addresses in the `LiquidityWindow::requestMint` and `LiquidityWindow::requestRefund`, differs from `STRX::mint` in that system accounts are not explicitly marked as passing the KYC verification.

This inconsistency prevents system addresses from being direct recipients in `LiquidityWindow::requestMint` and `LiquidityWindow::requestRefund`.

Recommended Mitigation: Consider allowing system accounts to bypass the KYC check within `LiquidityWindow`.

Strong: Acknowledged. Any admin wallet needed to mint will KYC directly through the protocol.

Cyfrin: Acknowledged.

7.6 Gas Optimization

7.6.1 Redundant zero address check in `STRX::burn` can be removed

Description: `STRX::burn` exposes `ERC20Upgradeable::_burn` as a public function to be called by the `LiquidityWindow` system contract. However, it contains the following superfluous zero address validation that is already present inside of the inherited internal function and can thus be removed:

```
function burn(address from, uint256 amount)
    external
    nonReentrant
    onlyLiquidityWindow
    whenNotPaused
{
@>  if (from == address(0)) revert ZeroAddress();
    _burn(from, amount);
}

function _burn(address account, uint256 value) internal {
@>  if (account == address(0)) {
        revert ERC20InvalidSender(address(0));
    }
    _update(account, address(0), value);
}
```

Recommended Mitigation: Remove the redundant check from `STRX::burn`:

```
function burn(address from, uint256 amount)
    external
    nonReentrant
    onlyLiquidityWindow
    whenNotPaused
{
-   if (from == address(0)) revert ZeroAddress();
    _burn(from, amount);
}
```

Strong: Fixed in commit ac83fa5.

Cyfrin: Verified. The redundant zero address check has been removed.

7.6.2 Redundant `isRecoverySink` assignment logic should be removed from `LiquidityReserve::initialize`

Description: `LiquidityReserve::initialize` populates the `isRecoverySink` mapping for the `liquidityWindow` and `treasurer` addresses:

```
// Set recovery sinks
isRecoverySink[treasurer_] = treasurer_ != address(0);

if (liquidityWindow_ != address(0)) {
    liquidityWindow = liquidityWindow_;
    _grantRole(DEPOSITOR_ROLE, liquidityWindow_);
    isRecoverySink[liquidityWindow_] = true;
}
if (treasurer_ != address(0)) {
    treasurer = treasurer_;
    _grantRole(TREASURER_ROLE, treasurer_);
    _grantRole(DEPOSITOR_ROLE, treasurer_);
    isRecoverySink[treasurer_] = true;
}
```

However, the repeated conditional `treasurer` logic is unnecessary as `isRecoverySink[treasurer_]` will always be set to `true` if `treasurer_ != address(0)` and will always default to `false` if the parameter is not passed.

Recommended Mitigation:

```
-   // Set recovery sinks
-   isRecoverySink[treasurer_] = treasurer_ != address(0);

    if (liquidityWindow_ != address(0)) {
        liquidityWindow = liquidityWindow_;
        _grantRole(DEPOSITOR_ROLE, liquidityWindow_);
        isRecoverySink[liquidityWindow_] = true;
    }
    if (treasurer_ != address(0)) {
        treasurer = treasurer_;
        _grantRole(TREASURER_ROLE, treasurer_);
        _grantRole(DEPOSITOR_ROLE, treasurer_);
        isRecoverySink[treasurer_] = true;
    }
```

Strong: Fixed in commit c21034a.

Cyfrin: Verified. The redundant logic has been removed.

7.6.3 `STRX::_update` conditional branches can be simplified

Description: The following construction within `STRX::_update` contains redundant checks that can be reduced:

```
if (from == address(0)) {
    // mint - KYC is enforced in LiquidityWindow before calling this
    super._update(from, to, value);
    _notifyRewards(address(0), to, value);
    return;
}

if (to == address(0)) {
    // burn
    super._update(from, to, value);
    _notifyRewards(from, address(0), value);
    return;
}
```

Recommended Mitigation:

```
if (from == address(0) || to == address(0)) {
    super._update(from, to, value);
    _notifyRewards(from, to, value);
    return;
}
```

Strong: Fixed in commit 3646b3c.

Cyfrin: Verified.

7.6.4 Unreachable code in `OracleAdapter::_scalePythValue` should be removed

Description: `OracleAdapter::_scalePythValue` contains conditional logic that is not possible to ever execute. Specifically, when validating the `pow == 0` case indicated below, this will always evaluate as false since ten raised to any power will be non-zero:

```
function _scalePythValue(uint256 value, int32 expo) private pure returns (uint256) {
    int256 exp = int256(expo) + 18;
    if (exp > 60 || exp < -60) return 0;
```

```
    if (exp >= 0) {
        uint256 pow = 10 ** uint256(exp);
@>      if (pow == 0) return 0;
        uint256 result = value * pow;
@>      if (pow != 0 && result / pow != value) return 0;
        return result;
    } else {
        uint256 pow = 10 ** uint256(-exp);
@>      if (pow == 0) return 0;
        return value / pow;
    }
}
```

By extension, this is also true for `result` unless `value` is 0, but this condition is always first checked by the calling function.

Recommended Mitigation: Consider removing the unnecessary validation.

Strong: Fixed in commit 3c35f6e. Removed unreachable `pow == 0` validation since exponent is bounded to `[-60, 60]` and `10 ** x` is always non-zero in this range.

Cyfrin: Verified.

7.6.5 `LiquidityWindow::requestMint` should combine separate calls to `LiquidityReserve::recordDeposit` for both fee and net amount

Description: In `LiquidityWindow::requestMint`, the gross amount recorded through `LiquidityReserve::recordDeposit` is executed inefficiently by invoking the function twice - once for net USDC deposit and once for the fee:

```
function requestMint(
    address recipient,
    uint256 usdcAmount,
    uint256 minStrxOut,
    uint256 maxEffectivePrice
)
    external
    nonReentrant
    whenNotPaused
    blockFreshCheck
    returns (uint256 strxOut, uint256 feeUsdc)
{
    ...

    // Route fees (now the contract has the fee USDC)
@>  _routeFees(feeUsdc);

    // Record the deposit
@>  ILiquidityReserve(liquidityReserve).recordDeposit(usdcAmount - feeUsdc);

    ...
}

function _routeFees(uint256 feeUsdc) internal {
    if (feeUsdc == 0) return;
    if (usdc == address(0)) return; // Skip if USDC not configured

    // Split fees between Reserve and Treasury according to feeToReservePct
@>  uint256 toReserve = (feeUsdc * feeToReservePct) / BPS_DENOMINATOR;
    uint256 toTreasury = feeUsdc - toReserve;

    // Verify contract has sufficient USDC balance
```

```
    uint256 balance = IERC20(usdc).balanceOf(address(this));
    require(balance >= feeUsdc, "Insufficient USDC for fee routing");

    // Send Reserve portion to Reserve
    if (toReserve > 0) {
        IERC20(usdc).safeTransfer(liquidityReserve, toReserve);
@>      ILiquidityReserve(liquidityReserve).recordDeposit(toReserve);
    }

    // Send Treasury portion directly to Treasury
    if (toTreasury > 0) {
        IERC20(usdc).safeTransfer(treasury, toTreasury);
    }
}
```

This can instead be performed in a single call, since there is no benefit from executing two separate transfers and notifications:

```
// LiquidityWindow (and other approved roles) call this whenever USDC enters the vault.
// Non-window callers transfer funds in as part of the call; window assumes it already holds them.
function recordDeposit(uint256 amount) external {
    if (!hasRole(DEPOSITOR_ROLE, msg.sender)) revert NotAuthorized();
    if (amount == 0) revert InvalidAmount();

    if (msg.sender != liquidityWindow) {
        asset.safeTransferFrom(msg.sender, address(this), amount);
    }
    emit DepositRecorded(msg.sender, amount);
}
```

Furthermore, the `depositAmount` variable should be reused for clarity and gas efficiency, rather than passing `usdcAmount - feeUsdc` as argument.

Recommended Mitigation: Consider modifying `LiquidityWindow::_routeFees` to forward only treasury fees and perform the gross USDC deposit transfer in the `LiqudityWindow::requestMint` with one transfer and one `recordDeposit` invocation.

Strong: Fixed in commit 2dc8873. We now emit a single `recordDeposit` on mint (net+reserve fee) and refactored `_routeFees()` accordingly.

Cyfrin: Verified.