Buck Labs Security Review

Auditors

R0bert, Lead Security Researcher

Sujith Somraaj, Lead Security Researcher

T1moh, Associate Security Researcher

Report prepared by: Lucas Goiriz

February 3, 2026

Contents

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 7 days in total, Buck Labs engaged with Spearbit to review the buck-smart-contracts-v1 protocol. In this period of time a total of **54** issues were found.

Summary

Project Name	Buck Labs
Repository	buck-smart-contracts-v1
Commit	741e701f
Type of Project	Stablecoin, Yield
Audit Timeline	Jan 26th to Feb 2nd

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	14	9	5
Low Risk	13	11	2
Gas Optimizations	0	0	0
Informational	25	21	4
Total	**54**	**43**	**11**

4.1 Scope

The security review had the following components in scope for buck-smart-contracts-v1 on commit hash 741e701f:

```
src
├── access (V1)
│   └── AccessRegistry.sol
├── collateral
│   └── CollateralAttestationV2.sol
├── liquidity
│   ├── LiquidityReserveV2.sol
│   └── LiquidityWindowV2.sol
├── oracle
│   ├── OracleAdapterV3.sol
│   └── OracleAdapterV4.sol
├── policy
│   └── PolicyManagerV2.sol
├── token
│   └── BuckV2.sol
└── utils
    ├── ReentrancyGuardTransient.sol
```

```
    ├── RewardsEngineV1_1.sol
    └── TransientSlot.sol
```

5 Findings

5.1 High Risk

5.1.1 `RewardsEngine` V1.1 missing distribute

Severity: High Risk

Context: buckUpgradeProcedure.md#L309-L315, RewardsEngineV1_1.sol#L58-L64

Description: The upgrade procedure requires calling `distribute` after upgrading the `RewardsEngine` proxy to V1.1. The provided `RewardsEngineV1_1` implementation is a standalone contract that only adds `claimFor` and `batchClaimFor` and does not inherit from the V1 implementation. As a result, core V1 entrypoints like `distribute` and epoch configuration are not present in this implementation.

If this implementation is used for the UUPS upgrade, calls expected by the migration runbook will revert because the function selectors are missing. That would block the rewards distribution step and leave the migration sequence in a broken state, forcing a rollback or another upgrade deployment to restore the missing interface.

```
cast send $REWARDS_ENGINE_PROXY "distribute(uint256)" \
    $COUPON_USDC \
    --from $DISTRIBUTOR_WALLET
```

```
contract RewardsEngineV1_1 is
    Initializable,
    AccessControlUpgradeable,
    PausableUpgradeable,
    MulticallUpgradeable,
    UUPSUpgradeable
{
    // NEW V1.1: Batch Claim Functions
    function claimFor(address user, address recipient)
        external
        onlyClaimAdmin
        returns (uint256 amount)
    { ... }

    function batchClaimFor(address[] calldata users)
        external
        onlyClaimAdmin
        returns (uint256 totalClaimed)
    { ... }
}
```

Recommendation: Ensure the V1.1 implementation is a strict superset of the V1 interface. The safest pattern is to inherit the full V1 implementation and add only the new batch claim functionality. Re-run a fork test that upgrades the proxy and executes the exact runbook sequence, including `distribute` and `batchClaimFor`, before any production upgrade.

Buck Labs: Fixed in commit 22968a5.

Cantina Managed: Fix verified. `RewardsEngineV1_1` is now a strict superset of V1 and includes `distribute` and full V1 entrypoints.

5.1.2 Mint pricing bypasses oracle validation

Severity: High Risk

Context: LiquidityWindowV2.sol#L442, LiquidityWindowV2.sol#L446, LiquidityWindowV2.sol#L552-L560, OracleAdapterV4.sol#L238-L243, OracleAdapterV4.sol#L436, PolicyManagerV2.sol#L429

Description: `PolicyManagerV2` computes the CAP price for mints and refunds from the oracle adapter using the view-only path. That view path returns the first fresh oracle without cross validation and without updating any circuit

breaker state. `LiquidityWindowV2` uses this view price for mints and only checks the cached circuit breaker flag, so mint pricing can continue to use an unvalidated or manipulated primary oracle even when cross-oracle divergence exists.

The circuit breaker state also does not reliably latch in normal refund flow. `LiquidityWindowV2` calls `refreshPrice` and then immediately calls `checkAndRecordRefund`, which reverts when the circuit breaker is tripped. Because the refresh and the revert happen in the same transaction, the state update that tripped the breaker is rolled back. Unless a separate transaction successfully calls `refreshPrice`, the breaker can remain false even under conditions that should trip it and mint pricing continues to use the view-only path.

This combination means the system can keep minting against a single oracle that is stale or manipulated, with no enforced cross-validation and no persistent circuit breaker protection. In the worst case, an attacker can mint at a depressed price and later redeem at a higher price, extracting reserve assets.

```
function latestPrice() external view returns (uint256 price, uint256 updatedAt) {
    return _fetchPriceView();
}

function _fetchPriceView() internal view returns (uint256 price, uint256 updatedAt) {
    (uint256 redstonePrice, uint256 redstoneTime, bool redstoneFresh) = _tryRedstone();
    if (redstoneFresh) return (redstonePrice, redstoneTime);
    (uint256 pythPrice, uint256 pythTime, bool pythFresh) = _tryPyth();
    if (pythFresh) return (pythPrice, pythTime);
    return (lastGoodPrice, lastGoodPriceTime);
}
```

```
function _computeCAPPrice() internal view returns (uint256 price) {
    (uint256 basePrice,) = IOracleAdapterV3(oracleAdapter).latestPrice();
    // ...
}

function requestRefund(...) external nonReentrant returns (uint256 usdcOut, uint256 feeUsdc) {
    // ...
    IOracleAdapterV3(oracleAdapter).refreshPrice();
    IPolicyManagerV2(policyManager).checkAndRecordRefund(usdcOut);
    // ...
}
```

Recommendation: Make mint pricing use a validated oracle result that cannot bypass cross-validation. One option is to add a view validation function that compares Redstone and Pyth and returns a safe price without relying on a latched circuit breaker, then use that in `PolicyManagerV2`. If you want to keep the circuit breaker state machine, call `refreshPrice` in the mint path and ensure the breaker can latch in a standalone successful transaction, for example by running a keeper that refreshes prices or by separating breaker evaluation from refund state changes.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` replaces the V4 CB/refresh flow and `LiquidityWindow` no longer refreshes CB state. Pricing now uses the single V5 path with max-age fallback.

5.2 Medium Risk

5.2.1 `Buck.updateYieldRate()` rounds down vested yield

Severity: Medium Risk

Context: BuckV2.sol#L301-L314, BuckV2.sol#L398

Description: Yield is vested to Buck, so that it distributes X amount of yield over Y days. It increases internal multiplier, for example 1.1 multiplier means 10% of yield received. Below formula is used:

```
vestedBps = (unvestedYieldBps * elapsed) / currentVestDuration;
yieldMultiplier += (yieldMultiplier * vestedBps) / BPS_DENOMINATOR;
```

It's important that `unvestedYieldBps` and therefore `vestedBps` have 1e4 precision. Suppose 10% yearly yield is distributed over 30 days, it means 0.83% over 30 days. For the value to increase rather than be rounded, must pass `30 * 24 * 3600 / 83 = 8.6 hours`. I.e. it increases in discrete steps , not continuously.

Yield rate can be updated, so it saves vested yield and updates the values:

```
function updateYieldRate(uint256 newYieldBps, uint256 newVestDays) external
↪  onlyYieldDistributor {
    // ...

    // Calculate vested portion from current stream (if any)
    uint256 vestedFromStream = 0;
    if (unvestedYieldBps > 0 && currentVestDuration > 0) {
        uint256 elapsed = block.timestamp - lastYieldTime;
        if (elapsed >= currentVestDuration) {
            vestedFromStream = unvestedYieldBps;
        } else {
            vestedFromStream = (unvestedYieldBps * elapsed) / currentVestDuration; // <<<
        }
    }


    // Move vested yield to accrued (NOT to yieldMultiplier - prevents compounding)
    accruedYieldBps += vestedFromStream;

    // Start new stream with new rate
    unvestedYieldBps = newYieldBps;
    currentVestDuration = newVestDuration;
    lastYieldTime = block.timestamp;

    emit YieldRateUpdated(newYieldBps, newVestDays, accruedYieldBps, block.timestamp);
}
```

Keeping in mind discrete steps, it means `updateYieldRate()` can save stale vested yield, i.e. truncate at max 1 bps or in other words last 8 hours of yield accrual.

If it's called once each month, it will truncate at most 12 bps over 1 year. With 5% yearly yield it means actual yield is `12 / 500 = 2.4%` less than expected (note: that's percent and not bps).

Recommendation: Consider migrating yield precision from `1e4` to `1e18`.

Buck Labs: Fixed in PR 26. We think we want to watch the price always increase, every block, and not bump every 8 hours. So we updated the yield stream decimals to wad precision.

Cantina Managed: Fix verified.

5.2.2 `Haircut < 1` can create arbitrage opportunity

Severity: Medium Risk

Context: CollateralAttestationV2.sol#L271-L282, PolicyManagerV2.sol#L344-L347

Description: The Buck token is backed by a USDC reserve and offchain funds in Alpaca Finance. The collateral ratio is calculated using the formula:

$$\text{CR} = \frac{R + (\text{HC} \times V) + \text{CIF}}{L}$$

where:

- CR: Collateral ratio.
- R: USDC reserve.

- V: Funds in Alpaca Finance.
- CIF: Funds in transit between R and V that are not yet reflected in those values.
- L: Buck token supply.
- HC: Haircut factor (discount applied to V's valuation).

Calculated CR is used as Buck price cap during mint and refund:

```
function _computeCAPPrice() internal view returns (uint256 price) {
    // ...

    // Cap by collateral ratio if undercollateralized
    uint256 cr = getCollateralRatio();
    if (cr < price) {
        price = cr;
    }

    return price;
}
```

It means that if V is less than 1, i.e. it underestimates funds in V, then it will calculate different CR when funds are moved between R and V. Consider following example:

1. Price is 0.99, user deposits 990 USDC and mints 1000 Buck.
2. CR = 990/1000 = 0.99.
3. Admin moves 900 to Alpaca and buys STRC. It takes time, so must wait.
4. Now CR = $(90 + 900 * 0.98)/1000 = 0.972$.
5. So it means price is capped by 0.972. And therefore users can mint Buck for less USDC. And then sell when funds are transferred back from V to R.

Recommendation: Arbitrage is mitigated by following steps:

1. Fees make it less profitable.
2. It's minimised if V value doesn't flactuate a lot from low fraction to high.
3. The closer HC to 1, the lower the profit.

HC = 1 mitigates arbitrage completely.

Buck Labs: Acknowledged. We dont really use the haircut and plan on keeping it at 0% aka 1.0.

CIF is automatically tracked when USDC leaves the reserve, so CR doesn't drop at the time of withdrawal - it only shifts when the attestor publishes V and CIF is cleared, which is an admin-coordinated operation external users can't front-run.

Even if someone could time it perfectly, the 58 bps round-trip fee (29 bps each way) consumes most of the gap at our planned 1-2% haircut. The HC discount is intentional conservative accounting, and the combination of CIF bridging + fees makes this arbitrage economically unviable, and also like I said we don't really plan on using it much.

Cantina Managed: Acknowledged.

5.2.3 Queued admin withdrawals not deducted from daily refund cap calculation

Severity: Medium Risk

Context: LiquidityReserveV2.sol#L283-L288, PolicyManagerV2.sol#L574

Description: When an admin queues a withdrawal in `LiquidityReserveV2`, the funds remain in the reserve during the delay period (default: 24 hours) but are committed to be released. However, the daily refund cap in `PolicyManagerV2` is calculated based on the raw reserve balance, which still includes these committed funds:

```
//PolicyManagerV2.sol
function _computeDailyCap() internal view returns (uint256) {
  uint256 reserveBalance = IERC20(usdc).balanceOf(liquidityReserve);
  return (reserveBalance * dailyCapPct) / 100;
}
// LiquidityReserveV2.sol - Admin path queues but doesn't move funds
if (!hasRole(ADMIN_ROLE, msg.sender)) {
  revert NotAuthorized();
}
_enqueueWithdrawal(to, amount, msg.sender);  // Funds stay in reserve
```

This creates a mismatch between promised capacity (`dailyCapUsdc`) and actual liquidity, causing unexpected refund failures.

Example Scenario:

1. Reserve: $10M, new cycle starts.
2. First refund triggers cap freeze: $6.6M (66% of $10M).
3. Admin executes previously-queued $5M withdrawal.
4. Reserve now: $5M.
5. Frozen cap still promises $6.6M capacity.
6. Users attempt refunds, expecting $6.6M available.
7. Refunds fail when the actual reserve is exhausted at $5M.

Recommendation: Consider tracking queued admin withdrawal amounts and deducting them from the daily cap calculation (or) recalculate the frozen cap when admin withdrawals are executed.

Buck Labs: Acknowledged. Admin 24-hour delay withdrawals are a rare operational case; we plan to always process instant withdrawals to the treasury for cash flow operations.

However, we will add a note to the docs in e49657: if we are processing an admin withdrawal, it must be early in the cycle window, before user refunds come in, to avoid this issue.

Cantina Managed: Acknowledged.

5.2.4 Current CIF design is prone to lagging

Severity: Medium Risk

Context: CollateralAttestationV2.sol#L201-L205, CollateralAttestationV2.sol#L241-L249, CollateralAttestationV2.sol#L254-L257

Description: `cashInFlow` was introduced to reflect funds that are no more in `LiquidityReserve`, but not yet in Alpaca Finance (`V`) and vice versa. `V` is updated in function `publishAttestation()`:

```
function publishAttestation(uint256 _V, uint256 _HC, uint256 _attestedTimestamp)
    external
    onlyRole(ATTESTOR_ROLE)
{
    // ...

    V = _V; // <<<
    HC = _HC;
    lastAttestationTime = block.timestamp;
    attestationMeasurementTime = _attestedTimestamp;
}
```

Admin is expected to call function `setCashInFlight()` to decrease `setCashInFlight()`:

```
/// @notice Set cash-in-flight amount (admin clears when cash arrives)
/// @param amount New CIF amount (18 decimals) - typically 0 to clear // <<<
function setCashInFlight(uint256 amount) external onlyRole(ADMIN_ROLE) {
    cashInFlight = amount;
    emit CashInFlightSet(amount);
}
```

Here is how admin is expected to update `cashInFlight`:

```
* CIF Flow (works for BOTH directions):
* - Outgoing (Reserve → Alpaca): LiquidityReserve calls addCashInFlight() on withdrawal
* - Incoming (Alpaca → Reserve): Admin calls addCashInFlight() when dividend wire initiated
* - Admin clears CIF via setCashInFlight(0) when cash arrives at destination
```

Such behaviour introduces situation when `cashInFlight` is overestimated:

- Alpaca → Reserve. When wire is initiated, `V` is old, but amount is already added to `cashInFlight`.
- Alpaca → Reserve. When `V` is new, funds arrived, but amount is still in `cashInFlight`.
- Reserve → Alpaca. When `V` is new, but amount is still in `cashInFlight`.

It means that `totalAssets` is higher than actual value and therefore it overestimates Collateral Ratio. It means that price cap check is less strict:

```
function _computeCAPPrice() internal view returns (uint256 price) {
    // ...

    // Cap by collateral ratio if undercollateralized
    uint256 cr = getCollateralRatio();
    if (cr < price) {
        price = cr;
    }

    return price;
}
```

If cap check indeed limits the price, then in that period price will instantly drop. And again instantly rises after admin action.

Recommendation: It's hard to solve issue completely. Intuitively it should update `cashInFlight` in same transaction as `publishAttestation()`. But this way lagging is still possible on Alpaca -> Reserve, when USDC already arrived but `cashInFlight` is not decreased.

Buck Labs: Acknowledged. There's not much we can do about it. Operationally, we need it in order for our status to stay "collateralized" while moving money around. The good news is that it only matters when the `CR is < price`. Right now we're (truly, no CIF) over collateralized at a 1.68 ratio, which will continue to shrink as we grow, but it keeps a healthy enough buffer so that the scenario where we face this is rare until we're above like, 100M market cap or something.

Anyways, if we're near undercollateralization, we'll be at a heightened operational posture and manually coordinating attestations and cash in flight tx's carefully.

Cantina Managed: Acknowledged.

5.2.5 Inconsistent `OracleAdapterV4` state transitions

Severity: Medium Risk

Context: OracleAdapterV4.sol#L380-L394

Description: `OracleAdapterV4.sol` uses 2 oracles: Redstone (primary) and Pyth (secondary). Logically there are 4 possible states:

1. Redstone works, Pyth works.
2. Redstone works, Pyth doesn't work.
3. Redstone doesn't work, Pyth works.
4. Redstone doesn't work, Pyth doesn't work.

Worth noting that Status has only 3 states:

```
/// @notice Oracle trust state machine
enum Status {
    RedstoneWorking, // Normal: Redstone is primary
    UsingPythRedstoneUntrusted, // Redstone failed/untrusted, Pyth is backup
    UsingInternalBothUntrusted // Both failed, internal price + CB tripped
}
```

Let's observe which oracle is used when we take into account previous state and current state. In current implementation it is following:

1. 1 → 1. Redstone.
2. 1 → 2. Redstone.
3. 1 → 3. Pyth.
4. 1 → 4. Internal.
5. 2 → 1. Redstone.
6. 2 → 2. Redstone.
7. 2 → 3. Pyth.
8. 2 → 4. Internal.
9. 3 → 1. Redstone if low deviation, otherwise Pyth.
10. 3 → 2. Internal.
11. 3 → 3. Pyth.
12. 3 → 4. Internal.
13. 4 → 1. Redstone if low deviation, otherwise Pyth.
14. 4 → 2. Redstone.
15. 4 → 3. Pyth.
16. 4 → 4. Internal.

State transitions are tracked because it's not safe to trust oracle if in previous call it didn't work. For example If Redstone didn't work in previous call, then code will trust it only after additional checks, i.e. cross check with Pyth that deviation is low. In case Pyth is also unreliable or stale - it can't trust that Redstone price, and hence use internal price.

I've also reviewed behaviour with original battle tested Liquity V1 implementation and noticed some deviations:

- 2 → 3. Liquity uses Internal, Buck uses Pyth. In previous round Pyth didn't work, so it's dangerous to trust it now without cross checks.
- 4 → 1. In case of high deviation Liquity uses Internal, Buck uses Pyth.
- 4 → 2. Liquity uses Internal, Buck uses Redstone.
- 4 → 3. Liquity uses Internal, Buck uses Pyth.

As you can see, in some cases Buck trusts oracle answer even if previously it was untrusted:

- Buck recovers to any oracle if both were unavailable previously.

- Buck doesn't have state `RedstoneWorksPythUntrusted`, that's why it misses Pyth unavailability in previous call.

Interesting observation is that `3 \rightarrow 2` transition doesn't trust Redstone response without cross check and fallbacks to Internal. While `2 \rightarrow 3` trusts Pyth without cross check.

Recommendation: Consider following Liquity's approach on state transitions.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` replaces the V4 state machine entirely.

5.2.6 Treasurer withdrawals track CIF even when sent to arbitrary addresses

Severity: Medium Risk

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

Description: `LiquidityReserveV2.queueWithdrawal()` treats any treasurer-initiated withdrawal as a brokerage transfer and always adds Cash-in-Flight (CIF), regardless of the destination address. The kickoff notes and the contract's own documentation define CIF as accounting for brokerage transfers only. Because the treasurer can withdraw to any address, CIF can be inflated even when funds are sent to non-brokerage destinations. This creates a mismatch between the intended CIF semantics and the on-chain behavior, and can overstate collateral metrics that depend on CIF.

This can be considered an spec mismatch because:

- Kickoff notes define CIF as covering reserve $\leftrightarrow$ brokerage transfers (Alpaca), with explicit lifecycle expectations.
- `LiquidityReserveV2`'s header comment states "Treasurer $\rightarrow$ instant, YES CIF (brokerage transfers)".

Actual behavior:

- Any treasurer withdrawal to any `to` address adds CIF.

```
// Treasurer fast path: instant withdrawal for brokerage (YES CIF)
if (hasRole(TREASURER_ROLE, msg.sender)) {
    _instantWithdrawal(to, amount, true); // Track CIF for brokerage transfers
    return;
}
```

Because of this, CIF can be increased for transfers that are not brokerage transfers, inflating collateral ratio and CAP price limits.

Recommendation: Enforce CIF only for verified brokerage destinations. For example, consider restricting treasurer withdrawals to an allowlist of approved sinks (e.g., brokerage wallets), or splitting treasurer withdrawals into two explicit functions: one that tracks CIF for brokerage transfers and one that does not.

If unrestricted treasurer withdrawals are intended, document that CIF is no longer strictly "brokerage cash in transit" and update monitoring expectations accordingly.

Buck Labs: Fixed in commit 44f3704

Cantina Managed: Fix verified. Treasurer withdrawals now only track CIF when sending to the treasury address.

5.2.7 Slip compensation ignores skim fee

Severity: Medium Risk

Context: PolicyManager.sol#L278, RewardsEngine.sol#L632-L635, buckUpgradeProcedure.md#L298

Description: The runbook computes the final coupon amount as normal yield plus slip compensation, but `RewardsEngine` applies `PolicyManager` `distributionSkimBps` and only the net coupon amount drives reward minting. `PolicyManager` defaults `distributionSkimBps` to 10 percent across bands, so following

the runbook numbers will underpay holders by the skim amount and route the difference to the treasury. This undermines the intended compensation and can lead to observable under-distribution.

```
distributionSkimBps: 1000, // 10% skim
```

```
skimUsdc = Math.mulDiv(couponUsdcAmount, skimBps, BPS_DENOMINATOR);
netCouponUsdc = couponUsdcAmount - skimUsdc;
```

Recommendation: Make the distribution calculation skim-aware by reading `distributionSkimBps` immediately before the final distribution and either setting it to zero for the event or grossing up the coupon so the net reward matches the intended compensation.

Buck Labs: Acknowledged. `skim` is currently set to 0 on v1 across all bands, and will stay at 0 for the upgrade and deprecation.

Cantina Managed: Acknowledged.

5.2.8 Rollback to V1 unsafe after V2

Severity: Medium Risk

Context: buckUpgradeProcedure.md#L685-L693

Description: `LiquidityWindow` V2 reorders and repurposes storage slots compared to V1. After the proxy is upgraded to V2, those slots will hold V2 values. If the proxy is later pointed back to the V1 implementation, V1 will interpret those same slots as different fields, corrupting addresses and state used for minting, refunds, and access checks. The runbook suggests rolling `LiquidityWindow` back to V1, which is unsafe once V2 has been live. Impact is a likely loss of functionality and possible misrouting of funds during primary market operations.

```
// V1 storage order (abridged)
address buck;
address liquidityReserve;
address policyManager;
address usdc;
address accessRegistry;
mapping(address => bool) isLiquiditySteward;
uint16 feeToReservePct;
address treasury;

// V2 storage order (abridged)
address buck;
address policyManager;
address oracleAdapter;
address liquidityReserve;
address accessRegistry;
address usdc;
address treasury;
uint16 feeToReservePct;
```

Recommendation: Remove the ”optional” `LiquidityWindow` rollback-to-V1 path from the runbook, or provide a dedicated rollback implementation that migrates V2 layout back into V1 layout and test it on a fork. If rollback is required operationally, deploy a fresh proxy with a clean V1 layout instead of downgrading the existing proxy.

Buck Labs: Fixed in commit e9df9c5.

Cantina Managed: Fix verified. Runbook now explicitly forbids `LiquidityWindow` rollback to V1 due to incompatible storage layout.

5.2.9 Refund uses pre-refresh price

Severity: Medium Risk

Context: LiquidityWindowV2.sol#L408-L409, LiquidityWindowV2.sol#L414-L415, LiquidityWindowV2.sol#L442

Description: Refunds compute the effective price using a view-only read of the `PolicyManager` and only refresh the oracle afterward. The refund flow calls `getRefundParams` to obtain `capPrice` and `spread`, uses that value to compute the refund amount and then invokes `refreshPrice` on the oracle to update circuit breaker state. This creates a time-of-check versus time-of-use gap where the price used for the refund may be stale relative to the refreshed oracle state.

In practice, this means a refund can be priced using older data even when a fresh oracle update would have changed the price or tripped the circuit breaker. The CB check is evaluated after the price is locked in, so the updated state does not affect the already computed refund price. This is a fragile ordering that can be exploited around volatile price changes or stale oracle updates.

```
(uint256 capPrice, uint16 refundFee, uint16 halfSpreadBps, bool paused, ) =
    IPolicyManagerV2(policyManager).getRefundParams();
// ...
uint256 effectivePrice = _applySpread(capPrice, false, halfSpreadBps);
// ...
IOracleAdapterV3(oracleAdapter).refreshPrice();
IPolicyManagerV2(policyManager).checkAndRecordRefund(usdcOut);
```

If the oracle updates materially between the view read and the refresh, users can receive refunds at an outdated price.

Recommendation: Refresh the oracle before computing the refund price, or re-fetch the price after refresh to keep pricing and CB state consistent. Alternatively, have `PolicyManager` return pricing that is derived from a refreshed oracle state within the same transaction path.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `LiquidityWindowV2` no longer refreshes oracle/circuit breaker after pricing. V5 pricing is a single path without CB ordering issues.

5.2.10 Pyth `maxConf` treated as absolute value

Severity: Medium Risk

Context: buckUpgradeProcedure.md#L499-L502, OracleAdapterV4.sol#L515-L519

Description: The Pyth configuration parameter `maxConf` is treated as an absolute confidence interval value in 18 decimal price units. The upgrade procedure documents `maxConf` as a percentage and configures it with a value of 500 for "5%", but the implementation compares the scaled absolute confidence directly against `pythMaxConf`. Passing 500 in this code path makes the confidence limit effectively near zero and will usually reject Pyth prices, silently disabling the fallback.

This creates a mismatch between the operational runbook and on-chain behavior. If the system expects Redstone and Pyth redundancy, this mismatch removes the intended safety net and makes the oracle adapter more brittle during outages or manipulation.

```
# Configure Pyth (secondary)
cast send $ORACLE_V4 "configurePyth(address,bytes32,uint256,uint256)" \
    $PYTH_ADDRESS $STRC_PRICE_ID 120 500 \  # 120s staleness, 5% max confidence
    --from $ADMIN_WALLET
```

```
if (pythMaxConf != 0 && p.conf > 0) {
    uint256 scaledConf = _scalePythValue(uint256(p.conf), p.expo);
    if (scaledConf > pythMaxConf) return (0, 0, false);
}
```

Recommendation: Either change the code to interpret maxConf as a relative threshold in basis points, or update the documentation and operational tooling to pass an absolute 18 decimal confidence value. Add a unit test that configures a realistic Pyth price and confidence and verifies the fallback is accepted under the intended threshold.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` removes Pyth fallback entirely, eliminating `maxConf` configuration and its unit mismatch.

5.2.11 `lastGoodPrice` updates during divergence

Severity: Medium Risk

Context: OracleAdapterV4.sol#L277-L279, OracleAdapterV4.sol#L310-L312

Description: `OracleAdapterV4` trips the circuit breaker when Redstone and Pyth disagree beyond the configured deviation threshold, but it still proceeds to update `lastGoodPrice` in the same refresh flow. That means the `lastGoodPrice` fallback can be set using a disputed oracle value during a divergence event. This undermines the meaning of `lastGoodPrice` as a verified safe anchor and can pin a manipulated or incorrect value for extended periods, especially during weekends or when both feeds are stale. If the circuit breaker is later reset and the feeds go stale, the protocol can fall back to this poisoned price for mint and refund pricing.

```
if (redstoneFresh && pythFresh) {
    uint256 deviation = _calculateDeviation(redstonePrice, pythPrice);
    if (deviation > maxOracleDeviationBps) {
        _tripCircuitBreaker(redstonePrice, pythPrice, deviation, "ORACLE_DIVERGENCE");
    }
}

if (redstoneFresh) {
    _updateLastGoodPrice(redstonePrice, "REDSTONE");
    _updatePriceTracking(redstonePrice, redstoneTime);
    return (redstonePrice, redstoneTime);
}
```

The impact is a degraded safety fallback during the exact conditions the circuit breaker is intended to flag as suspicious. This can lead to mispricing after manual resets or during prolonged staleness windows.

Recommendation: Do not update `lastGoodPrice` when divergence triggers the circuit breaker. Keep the prior `lastGoodPrice`, or update it using a safer value such as the minimum of the two sources or an explicit median if additional sources are added. Consider also skipping `lastSavedPrice` updates when the circuit breaker is tripped to avoid anchoring to disputed data.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` removes the V4 divergence CB/state machine, so `lastGoodPrice` no longer updates during divergence events.

5.2.12 Daily cap ignores treasury outflow

Severity: Medium Risk

Context: LiquidityWindowV2.sol#L422-L427, LiquidityWindowV2.sol#L444-L452, PolicyManagerV2.sol#L468-L474

Description: The daily refund cap is enforced using the net USDC paid to the user, but the reserve outflow for a refund also includes the treasury fee portion. `LiquidityWindowV2` computes `grossUsdc`, subtracts fees to get `usdcOut`, then records `usdcOut` against the daily cap. The reserve still pays out `grossUsdc` and only receives the reserve fee portion back, so the net reserve drain is `usdcOut` plus the treasury fee.

If the daily cap is intended to limit how much reserve can leave the system in a day, this accounting understates usage and allows reserve depletion beyond the intended percentage. The discrepancy grows with higher refund fees.

```
uint256 grossUsdc = (buckAmount * effectivePrice) / 1e18 / DECIMAL_ADJUSTMENT;
feeUsdc = _calculateFee(grossUsdc, refundFee);
usdcOut = grossUsdc - feeUsdc;
```

```
IOracleAdapterV3(oracleAdapter).refreshPrice();
IPolicyManagerV2(policyManager).checkAndRecordRefund(usdcOut);

ILiquidityReserve(liquidityReserve).queueWithdrawal(address(this), grossUsdc);
uint256 feeToReserve = _routeFees(feeUsdc);
if (feeToReserve > 0) {
    IERC20(usdc).safeTransfer(liquidityReserve, feeToReserve);
}
IERC20(usdc).safeTransfer(recipient, usdcOut);
```

Recommendation: Record daily cap usage using the net reserve outflow rather than only the user payout. A simple fix is to pass `usdcOut` plus the treasury fee to `checkAndRecordRefund`, or to pass `grossUsdc` if you want the cap to reflect total withdrawals before fee recycling. Document the chosen definition so operators and risk models match the on chain behavior.

Buck Labs: Fixed in commit b6e4834.

Cantina Managed: Fix verified. Refund cap accounting now uses netReserveDrain (user payout + treasury fee), aligning cap usage with actual reserve outflow.

5.2.13 Accrued yield can grow without cap

Severity: Medium Risk

Context: BuckV2.sol#L384, BuckV2.sol#L402-L403

Description: `updateYieldRate` moves the vested portion of the current stream into `accruedYieldBps` and then starts a new stream, but there is no cap on the accumulated value of `accruedYieldBps`. The only cap enforced is on the new stream rate, not on the total accrued yield over time. This allows the accrued basis points to grow beyond the intended maximum yield cap across repeated updates.

Because `currentYieldMultiplier` later applies `accruedYieldBps` to the price, the accounting baseline can drift upward without a hard bound, even if each individual stream stays within limits. This can break invariants that assume yield is bounded over time by configuration.

```
function updateYieldRate(uint256 newYieldBps, uint256 newVestDays) external
↪  onlyYieldDistributor {
    // ...
    uint256 vestedFromStream = 0;
    if (unvestedYieldBps > 0 && currentVestDuration > 0) {
        uint256 elapsed = block.timestamp - lastYieldTime;
        if (elapsed >= currentVestDuration) {
            vestedFromStream = unvestedYieldBps;
        } else {
            vestedFromStream = (unvestedYieldBps * elapsed) / currentVestDuration;
        }
    }

    accruedYieldBps += vestedFromStream;
    // ...
}
```

Recommendation: Add a hard cap on the cumulative accrued yield, or fold accrued yield into the base multiplier with explicit bounds. Document whether the protocol expects total accrued yield to remain within a fixed maximum.

Buck Labs: Fixed in commit c858dfa.

Cantina Managed: Fix verified. `updateYieldRate` now caps `accruedYieldBps` and reverts if it exceeds `MAX_YIELD_BPS`.

5.2.14 AccessRegistry bypass disables denylist

Severity: Medium Risk

Context: buckUpgradeProcedure.md#L116-L119, BuckV2.sol#L757-L760

Description: The migration runbook proposes temporarily setting `accessRegistry` to `address(0)` to bypass allowlist checks for `batchClaimFor`. In `BuckV2`, denylist enforcement is also guarded by `accessRegistry` being nonzero. When the registry is set to zero, both allowlist and denylist checks are disabled for transfers. That creates a window where previously denylisted accounts can send and receive BUCK, which is broader than the stated goal of bypassing only allowlist checks.

This can undermine compliance or security controls during the migration window and is easy to overlook because the runbook describes it only as a KYC bypass.

```
# Option B: Temporary AccessRegistry Bypass
# Set accessRegistry = address(0) on Buck before batchClaimFor
# Execute batchClaimFor (no access checks)
# Restore accessRegistry after
```

```
function _update(address from, address to, uint256 value) internal override {
    if (accessRegistry != address(0)) {
        if (from != address(0) && IAccessRegistry(accessRegistry).isDenylisted(from)) revert
        ↪  Frozen(from);
        if (to != address(0) && IAccessRegistry(accessRegistry).isDenylisted(to)) revert
        ↪  Frozen(to);
    }
    // ...
}
```

Recommendation: Avoid disabling the registry for migration, or add a migration-specific bypass that only skips allowlist checks while keeping denylist enforcement active. If a full bypass is unavoidable, pause token transfers during the window or ensure denylisted addresses cannot interact until the registry is restored.

Buck Labs: Acknowledged. We have actually not denylisted anybody so far and probably won't before the upgrade is happening.

Cantina Managed: Acknowledged.

5.3 Low Risk

5.3.1 `LiquidityReserveV2` does not respect ecosystem pause

Severity: Low Risk

Context: LiquidityReserveV2.sol#L30

Description: `LiquidityReserveV2` has its own independent pause mechanism but does not check the `PolicyManagerV2` ecosystem-wide pause state. If `PolicyManagerV2` is paused but `LiquidityReserveV2` is not:

- Treasurer can still call `queueWithdrawal()` to perform instant withdrawals with CIF tracking.
- RewardsEngine can still call `withdrawDistributionSkim()` to withdraw distribution skim.
- Admin can still execute queued withdrawals via `executeWithdrawal()`.

This creates inconsistent pause behavior across the protocol. During an ecosystem emergency, funds can still flow out of the reserve even when the protocol is frozen.

Recommendation: Add a reference to PolicyManager and check its pause state in the affected functions:

```
address public policyManager;
modifier whenEcosystemNotPaused() {
    if (policyManager != address(0) && IPolicyManagerV2(policyManager).paused()) {
```

```
            revert EcosystemPaused();
        }
        _;
    }
```

Apply this modifier to `queueWithdrawal()`, `withdrawDistributionSkim()`, and `executeWithdrawal()`.

Alternatively, document that operational procedures must pause both contracts simultaneously during emergencies, though on-chain enforcement is more robust.

Buck Labs: Fixed in 1ad10c9.

Cantina Managed: Fix verified.

5.3.2 `ADMIN_ROLE` used for routine cash-in-flight operations increases attack surface

Severity: Low Risk

Context: CollateralAttestationV2.sol#L37-L38

Description: The `addCashInFlight()` and `setCashInFlight()` functions in `CollateralAttestationV2` require ADMIN_ROLE for callers other than liquidityReserve.

These functions are likely called frequently by automated watchers monitoring wire transfers between the reserve and Alpaca. Such systems typically run on ”hot” servers with keys readily available for signing transactions. If a hot server running CIF operations is compromised, the attacker gains full administrative control over the contract, not just CIF. management. This violates the principle of least privilege.

Recommendation: Introduce a dedicated CIF_OPERATOR_ROLE with permissions limited to cash-in-flight operations.

Buck Labs: Acknowledged. Every admin wallet call is under a 3/4-quorum Fireblocks multisig.

For now we are going to keep any Cash In Flight calls locked up under management operations + approval threshold until we build a keeper bot or something to watch wires out of alpaca.

Cantina Managed: Acknowledged.

5.3.3 Transaction re-ordering of `setCashInFlight` can cause transient collateral ratio divergence

Severity: Low Risk

Context: CollateralAttestationV2.sol#L254-L257

Description: The `setCashInFlight()` function in `CollateralAttestationV2` uses absolute value assignment to set the cash-in-flight amount. This design is vulnerable to transaction reordering, which can lead to an incorrect collateral ratio until it is manually detected and corrected.

The cash-in-flight (CIF) mechanism tracks USDC in transit between the LiquidityReserve and Alpaca (in either direction). The workflow involves:

1. `addCashInFlight(amount)` - increments CIF when a transfer is initiated.
2. `setCashInFlight(0)` - admin clears CIF when funds arrive at destination.

Consider the following scenario:

1. Admin observes that all inflight funds have arrived at Alpaca.
2. Admin submits transaction 1: `setCashInFlight(0)` to clear CIF.
3. Following that, a new withdrawal is initiated via transaction 2: `addCashInFlight(100e6)`.

If transaction 2 is mined before transaction 1 (due to gas price differences, MEV, or network conditions):

- After transaction 2: `cashInFlight = 100e18`.
- After transaction 1: `cashInFlight = 0` (incorrectly clears the new inflight amount).

The collateral ratio formula,

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

will now be understated because the 100 USDC that is actually in transit is not accounted for in the numerator (CIF). This misrepresentation persists until the admin discovers and rectifies the discrepancy, or until the next CIF-modifying transaction reveals the issue.

Recommendation: Consider introducing a new function that does a conditional clear by validating the expected state before modifying it:

```
error CashInFlightMismatch(uint256 actual, uint256 expected);

/// @notice Clear cash-in-flight only if it matches the expected amount
/// @param expectedAmount The expected current CIF value (18 decimals)
/// @dev Reverts if current CIF doesn't match, preventing reordering issues
function clearCashInFlight(uint256 expectedAmount) external onlyRole(ADMIN_ROLE) {
 if (cashInFlight != expectedAmount) {
   revert CashInFlightMismatch(cashInFlight, expectedAmount);
 }
 cashInFlight = 0;
 emit CashInFlightSet(0);
}
```

This ensures that the clear operation is specific to the in-flight amount it intends to clear. If transactions are re-ordered, the mismatch will revert rather than silently corrupt the CIF state.

Alternatively, consider implementing a subtractCashInFlight(uint256 amount) function that decrements the balance by a specified amount and reverts on underflow if reordering occurs.

Buck Labs: Fixed in 7001b3.

Cantina Managed: Fix verified.

5.3.4 Permit front-running can DoS `requestMintWithPermit()`

Severity: Low Risk

Context: LiquidityWindowV2.sol#L375

Description: The `requestMintWithPermit()` function executes a USDC permit before performing the mint:

```
// Execute permit - will revert on invalid signature
IERC20Permit(usdc).permit(msg.sender, address(this), usdcAmount, deadline, v, r, s);

return _executeMint(recipient, usdcAmount, minBuckOut, maxEffectivePrice);
```

An attacker can monitor the mempool, extract the permit signature from a pending transaction, and front-run by calling USDC.permit() directly with the same parameters. The victim's transaction will then revert because the permit nonce has already been consumed.

While the attacker gains nothing (the allowance is still set for the victim), this creates a griefing vector that forces users to fall back to the standard approve + requestMint flow.

Recommendation: Wrap the permit call in a try-catch to gracefully handle already-used permits:

```
try IERC20Permit(usdc).permit(msg.sender, address(this), usdcAmount, deadline, v, r, s) {}
↪  catch {}

// Proceed with mint - will fail naturally if allowance is insufficient
return _executeMint(recipient, usdcAmount, minBuckOut, maxEffectivePrice);
```

This pattern is recommended by OpenZeppelin and used in Uniswap's `Permit2`.

Buck Labs: Fixed in bad826.

Cantina Managed: Fix verified.

5.3.5 `lastGoodPrice` should reflect actual price update timestamp

Severity: Low Risk

Context: OracleAdapterV4.sol#L578-L583

Description: There are 4 types of prices in `OracleAdapterV4`:

- `redstone`.
- `pyth`.
- `internal`.
- `lastGoodPrice`.

First 3 have their own timestamp of last update. In case it uses price from those 3, it updates `lastGoodPrice` which is last resort in case other are unavailable:

```
function _updateLastGoodPrice(uint256 price, string memory source) internal {
    if (price == 0) return;
    lastGoodPrice = price;
    lastGoodPriceTime = block.timestamp; // <<<
    emit LastGoodPriceUpdated(price, source);
}
```

As you can see, it sets `lastGoodPriceTime = block.timestamp` which is not correct. Because price has it's own time, which is different from `block.timestamp`. Though in protocol `lastGoodPriceTime` is not used.

Recommendation: Consider setting actual timestamp:

```
- function _updateLastGoodPrice(uint256 price, string memory source) internal {
+ function _updateLastGoodPrice(uint256 price, uint256 priceTimestamp, string memory source)
↪   internal {
      if (price == 0) return;
      lastGoodPrice = price;
-     lastGoodPriceTime = block.timestamp;
+     lastGoodPriceTime = priceTimestamp;
      emit LastGoodPriceUpdated(price, source);
  }
```

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` updates `lastGoodPriceTime` with the source timestamp, not `block.timestamp`.

5.3.6 Buck takes swap fee on LP operations

Severity: Low Risk

Context: BuckV2.sol#L680-L681

Description: It takes fee on every transfer if Dex pair is involved:

```
    bool isBuy = isDexPair[from];
    bool isSell = isDexPair[to];
    if (!isBuy && !isSell) return 0;

    if (policyManager == address(0)) return 0;
```

```
    (uint16 buyFee, uint16 sellFee) = IPolicyManager(policyManager).getDexFees();

    if (isBuy) {
        if (_isFeeExempt(to)) return 0;
        // Ceil division: rounds UP to favor protocol
        uint256 numerator = amount * buyFee;
        return (numerator + BPS_DENOMINATOR - 1) / BPS_DENOMINATOR;
    }
    if (isSell) {
        if (_isFeeExempt(from)) return 0;
        // Ceil division: rounds UP to favor protocol
        uint256 numerator = amount * sellFee;
        return (numerator + BPS_DENOMINATOR - 1) / BPS_DENOMINATOR;
    }
    return 0;
}
```

Intention is to take fee on all Buck swaps. However LP operations like `addLiquidity` and `removeLiquidity` are also invoke fee, because `from` or `to` is also DEX address.

Recommendation: Looks like design limitation to distinguish LP operation and swap. Consider documenting this behaviour.

Buck Labs: Fixed in commit 009517e.

Cantina Managed: Fix verified. Docs now explicitly note LP add/remove incurs DEX fees (no swap/LP distinction).

5.3.7 Runbook calls missing `setRewardsEngine`

Severity: Low Risk

Context: buckUpgradeProcedure.md#L670-L675

Description: The upgrade runbook includes a cleanup step that calls `setRewardsEngine` on the BUCK proxy. The `BuckV2` implementation does not expose a `setRewardsEngine` function because the rewards hook was removed in V2. As written, this command will revert and can cause confusion or block automated runbook execution.

```
cast send $BUCK_PROXY "setRewardsEngine(address)" 0x0000000000000000000000000000000000000000 \
    --from $ADMIN_WALLET
```

Recommendation: Remove the step or replace it with the correct V2 action. If a rewards hook needs to be cleared, do it before the V2 upgrade using the V1 configuration method, or document that no action is required after the V2 upgrade.

Buck Labs: Fixed in commit 0595e3d.

Cantina Managed: Fix verified. Runbook now notes `BuckV2` removed `setRewardsEngine` and no cleanup call is required.

5.3.8 Stale lastGoodPrice has no max age

Severity: Low Risk

Context: OracleAdapterV4.sol#L408-L424, OracleAdapterV4.sol#L469-L470

Description: `OracleAdapterV4` falls back to `lastGoodPrice` when both external oracles are stale and the internal price is stale and it does so without any maximum age check. Because `PolicyManagerV2` uses `latestPrice` for primary market pricing, mint and refund operations can continue using an arbitrarily old pinned price during extended oracle outages. If the reference price moves materially while oracles are stale, allowed users can mint or refund at a stale price and extract value over time. Daily caps limit the rate but not the existence of the loss.

```
// Ultimate fallback
return (lastGoodPrice, lastGoodPriceTime);
```

Recommendation: Introduce a maximum age for `lastGoodPrice` in primary market paths. If `lastGoodPriceTime` exceeds the threshold, block refunds (and optionally mints) or require a manual internal price update before trading resumes.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` adds `lastGoodPriceMaxAge` and enforces age limits for `lastGoodPrice` fallback.

5.3.9 Fee distribution bypasses denylist

Severity: Low Risk

Context: BuckV2.sol#L710-L712, BuckV2.sol#L755-L760

Description: `BuckV2` enforces denylist checks in its overridden `_update` function, which is used for normal transfers. However, when DEX fees are collected, `_distributeFees` uses `super._update` to transfer the fee legs to the reserve and treasury. Because `super._update` bypasses the denylist checks, fee transfers can succeed even if a recipient is denylisted. This contradicts the stated behavior that denylisted accounts cannot send or receive tokens.

If denylisting is used as a compliance or security control, this creates an exception path that can still deliver funds to frozen system accounts, and it makes behavior inconsistent with user-facing expectations.

```
function _distributeFees(address source, uint256 feeAmount) internal {
    uint256 toReserve = (feeAmount * feeToReservePct) / BPS_DENOMINATOR;
    uint256 toTreasury = feeAmount - toReserve;
    if (toReserve > 0) {
        super._update(source, liquidityReserve, toReserve);
    }
    if (toTreasury > 0) {
        super._update(source, treasury, toTreasury);
    }
}

function _update(address from, address to, uint256 value) internal override {
    if (accessRegistry != address(0)) {
        if (from != address(0) && IAccessRegistry(accessRegistry).isDenylisted(from)) revert
        ↪  Frozen(from);
        if (to != address(0) && IAccessRegistry(accessRegistry).isDenylisted(to)) revert
        ↪  Frozen(to);
    }
    // ...
}
```

Recommendation: Route fee transfers through the overridden `_update` so the denylist checks apply, or explicitly document and enforce that system accounts are exempt and should never be denylisted. If exemption is intended, add a guard that prevents denylisting the fee recipients to avoid surprises.

Buck Labs: Fixed in commit 06cd7bd.

Cantina Managed: Fix verified. Docs now define system accounts as denylist-exempt for fee distribution, aligning behavior with expectations.

5.3.10 Cash-in-flight units inconsistent

Severity: Low Risk

Context: CollateralAttestationV2.sol#L237-L257

Description: `CollateralAttestationV2` accepts cash-in-flight values in two different unit conventions. `addCashInFlight` expects reserve asset decimals and scales the value to 18 decimals, while `setCashInFlight` expects an 18 decimal value directly. The comment above `setCashInFlight` advises

using a `setCashInFlightUsdc` helper for 6-decimal amounts, but no such function exists, which makes it easy for operators or integration code to pass a 6 decimal USDC value into `setCashInFlight` and accidentally overstate CIF by 10^{12}.

CIF also has no expiry or time-based decay. A stale CIF value can persist indefinitely and keep the collateral ratio inflated until an admin explicitly resets it, which can make the system appear healthier than it is and allow more favorable pricing than intended.

```
function addCashInFlight(uint256 amount) external {
    // ...
    uint256 scaledAmount = _scaleToEighteen(amount, reserveAssetDecimals);
    cashInFlight += scaledAmount;
    emit CashInFlightAdded(scaledAmount, cashInFlight);
}

/// @dev Use setCashInFlightUsdc() for 6-decimal USDC amounts
function setCashInFlight(uint256 amount) external onlyRole(ADMIN_ROLE) {
    cashInFlight = amount;
    emit CashInFlightSet(amount);
}
```

Recommendation: Add explicit helper functions that accept reserve-decimal amounts and scale internally and consider deprecating the raw 18 decimal setter or renaming it to make the unit explicit. Track a timestamp for CIF updates and either enforce a maximum age or require periodic re-affirmation so stale values cannot persist unnoticed.

Buck Labs: Fixed in commit bd07574.

Cantina Managed: Fix verified. CIF setters now accept reserve-decimal inputs and CIF expires after 14 days, addressing unit mismatch and staleness.

5.3.11 Allowlist enforced only on mint

Severity: Low Risk

Context: BuckV2.sol#L268-L272, BuckV2.sol#L755-L760

Description: The BUCK token enforces the allowlist only during minting. In `BuckV2`, the mint entrypoint calls the access registry check on the recipient, but transfers do not. The transfer path funnels through the ERC20 _update override, which only checks the denylist and then applies fees. As a result, any address can become a holder by receiving tokens via transfer, even if it would have failed the allowlist check during minting.

This creates a mismatch between the apparent access control model and the actual holder set. If the allowlist is intended to restrict who can hold or receive BUCK, then a compliant holder can transfer to a non-allowed address and bypass that restriction. If the allowlist is only intended as a mint gate, then the current behavior should be explicitly documented and downstream modules should not assume allowlist membership for holders.

```
function mint(address to, uint256 amount) external nonReentrant onlyLiquidityWindow
↪  whenNotPaused {
    if (to == address(0)) revert ZeroAddress();
    _enforceAccess(to);
    _mint(to, amount);
}

function _update(address from, address to, uint256 value) internal override {
    if (accessRegistry != address(0)) {
        if (from != address(0) && IAccessRegistry(accessRegistry).isDenylisted(from)) revert
        ↪  Frozen(from);
        if (to != address(0) && IAccessRegistry(accessRegistry).isDenylisted(to)) revert
        ↪  Frozen(to);
    }
```

```
    // ...
}
```

If the protocol relies on the allowlist to constrain holders, this gap can lead to unvetted addresses receiving and holding BUCK.

Recommendation: Decide whether the allowlist is meant to restrict holders or only mint recipients. If holders must be allowlisted, add allowlist checks in the transfer path for both sender and receiver (or at least the receiver). If mint-only gating is intentional, document it clearly and ensure any compliance or reporting logic does not assume holders are allowlisted.

Buck Labs: Fixed in commit 8475bc7.

Cantina Managed: Fix verified. Docs clarify allowlist gates primary market only; secondary transfers are permissionless while denylist still blocks send/receive.

5.3.12 Daily cap can be inflated via USDC transfer

Severity: Low Risk

Context: PolicyManagerV2.sol#L568-L576, PolicyManagerV2.sol#L588-L591

Description: The daily refund cap is derived from the raw USDC balance of the reserve and then frozen for the cycle when the first refund is recorded. The logic reads the ERC20 balance directly and stores it as the cap for the day.

```
function _computeDailyCap() internal view returns (uint256) {
    if (liquidityReserve == address(0) || usdc == address(0)) {
        return 0;
    }

    uint256 reserveBalance = IERC20(usdc).balanceOf(liquidityReserve);
    return (reserveBalance * dailyCapPct) / 100;
}
```

```
function _maybeResetCycle() internal {
    uint64 cycle = _currentCapCycle();

    if (cycle != currentCapCycle || dailyCapUsdc == 0) {
        currentCapCycle = cycle;
        dailyUsedUsdc = 0;
        dailyCapUsdc = _computeDailyCap();

        emit DailyCapReset(cycle, dailyCapUsdc);
    }
}
```

`LiquidityReserveV2` does not restrict inbound ERC20 transfers, so any address can transfer USDC directly to the reserve. Because the cap is based on `balanceOf` and is snapshotted on the first refund in the cycle, a user can inflate the cap by donating USDC just before that first refund, increasing the maximum refundable amount for the entire day. Subsequent reserve withdrawals do not reduce the stored cap and the source of the balance increase is not distinguished from protocol-managed liquidity.

A realistic path is a user who already holds BUCK (or has borrowed it) sends USDC to the reserve right before the first refund of the day, causing a larger cap to be set, then immediately refunds BUCK up to the inflated limit, effectively accelerating their redemption beyond the intended throttle.

Impact: the daily cap no longer reflects protocol-controlled liquidity and can be expanded by any third party, weakening the cap as a risk-control during stress or bank-run conditions.

Recommendation: Base the daily cap on protocol-accounted reserves rather than raw ERC20 balance. Track an internal `accountedReserveUsdc` value in the reserve contract that is updated only on known inflows and outflows,

and compute the cap from that value. Alternatively, require an admin action to recognize external transfers as donations before they affect cap calculations, or explicitly document that external donations can expand the cap and ensure operations are comfortable with that behavior.

Buck Labs: Acknowledged. Documentation now explicitly defines daily cap as a % of actual reserve balance and treats external USDC transfers as intentional liquidity additions.

Cantina Managed: Acknowledged.

5.3.13 Permissive collateral ratio defaults

Severity: Low Risk

Context: PolicyManagerV2.sol#L356-L363

Description: Collateral ratio helpers return permissive values when the system is unconfigured or has zero supply. In `PolicyManagerV2`, `getCollateralRatio` returns `1e18` when references are not set or supply is zero. In `CollateralAttestationV2`, `getCollateralRatio` returns the max `uint256` value when supply is zero. These defaults imply full or even infinite collateralization during initialization, upgrade gaps, or a zero-supply state.

Any component that relies on these CR values for pricing, caps, or safety checks can be misled into allowing operations under optimistic assumptions. This is particularly sensitive during initialization or upgrades, when misconfiguration is most likely and other invariants may not yet hold.

```
function getCollateralRatio() public view returns (uint256 cr) {
    if (buck == address(0) || collateralAttestation == address(0)) {
        return 1e18;
    }
    uint256 supply = IBuckV2(buck).totalSupply();
    if (supply == 0) {
        return 1e18;
    }
    // ...
}

function getCollateralRatio() public view returns (uint256) {
    uint256 L = IERC20(buckToken).totalSupply();
    if (L == 0) return type(uint256).max;
    // ...
}
```

Using optimistic CR defaults can lead to pricing and risk controls that are too permissive during edge states.

Recommendation: Treat unconfigured or zero-supply states explicitly by returning 0, reverting, or gating operations until a known-good configuration is in place. If a permissive bootstrap mode is intended, document it and constrain which operations are allowed during that mode.

Buck Labs: Fixed (upgrade window) in commit 8d35509.

Cantina Managed: Partial mitigation. Pausing `LiquidityWindow` before the `PolicyManager` upgrade reduces exposure during the upgrade window, but the underlying permissive CR defaults in `PolicyManagerV2/CollateralAttestationV2` remain after the upgrade.

5.4 Informational

5.4.1 Oversized V1 deprecation gap wastes storage slots

Severity: Informational

Context: PolicyManagerV2.sol#L94-L97

Description: The `__v1_deprecated_gap` is sized for 25 slots, but V1 `PolicyManager` uses only 20 slots (5-24) for its band/config state. This results in 5 wasted storage slots and incorrect documentation.

Recommendation:

```
- uint256[25] private __v1_deprecated_gap;
+ uint256[20] private __v1_deprecated_gap;
```

Buck Labs: Fixed in c520e2c.

Cantina Managed: Fix verified.

5.4.2 Incorrect gap in `PolicyManagerV2`

Severity: Informational

Context: PolicyManagerV2.sol#L607

Description: `PolicyManager` in earlier versions had gap 52, V2 version added 11 storage variables, which pack into 4 slots, from 30 - 33. Hence, the newer gap length should be 48, not 45.

Recommendation:

```
- uint256[45] private __gap;
+ uint256[48] private __gap;
```

Buck Labs: Fixed in 0a47009.

Cantina Managed: Fix verified.

5.4.3 Mismatched documentation for `maxMintPerTxUsdc` and `maxRefundPerTxUsdc`

Severity: Informational

Context: LiquidityWindowV2.sol#L124-L128

Description: The NatSpec documentation for the storage variables `maxMintPerTxUsdc` and `maxRefundPerTxUsdc` in `LiquidityWindowV2` incorrectly states that setting the value to 0 means "no limit". However, the actual implementation uses `type(uint256).max` to represent "no limit".

Recommendation: Update the storage variable NatSpec to match the actual implementation:

```
/// @notice Max USDC per mint transaction (type(uint256).max = no limit)
uint256 public maxMintPerTxUsdc;

/// @notice Max USDC output per refund transaction (type(uint256).max = no limit)
uint256 public maxRefundPerTxUsdc;
```

Buck Labs: Fixed in 4c3619d.

Cantina Managed: Fix verified.

5.4.4 Outdated documentation references non-existent function

Severity: Informational

Context: CollateralAttestationV2.sol#L253

Description: The NatSpec documentation for `setCashInFlight()` references a non-existent function `setCashInFlightUsdc()`:

```
/// @dev Use setCashInFlightUsdc() for 6-decimal USDC amounts
function setCashInFlight(uint256 amount) external onlyRole(ADMIN_ROLE) {
```

This function does not exist. The existing `addCashInFlight()` function already handles 6-decimal USDC amounts by scaling internally via `_scaleToEighteen()`. The outdated comment is misleading and may confuse integrators.

Recommendation: Remove the stale @dev comment.

Buck Labs: Fixed in commit bd07574.

Cantina Managed: Fix verified. Misleading `setCashInFlightUsdc` reference removed/clarified. Setter now documents reserve-decimal inputs.

5.4.5 Incorrect gap in `CollateralAttestationV2`

Severity: Informational

Context: CollateralAttestationV2.sol#L443

Description: `CollateralAttestationV1` had gap 50, V2 version added 2 storage variables: `uint256 cashInFlight`, `address attestor`. So gap is reduced by 2 slots:

```
// ------------------------------------------------------------------------
// Storage - V2 New Variables (added after V1 storage)
// ------------------------------------------------------------------------

/// @notice Cash-in-Flight: USDC in transit (both directions) not yet at destination (18
↪  decimals)
uint256 public cashInFlight;

/// @notice Attestor service address (for compatibility, but role-based access preferred)
address public attestor;
```

Recommendation:

```
- uint256[45] private __gap;
+ uint256[48] private __gap;
```

Buck Labs: Fixed in commit 7ddb61c.

Cantina Managed: Fix verified. `CollateralAttestationV2` storage gap corrected to 48 slots.

5.4.6 Consider reusing `CollateralAttestationV2.getCollateralRatio()` instead of `PolicyManagerV2.getCollateralRatio()`

Severity: Informational

Context: PolicyManagerV2.sol#L355-L371

Description: Basically they are same function except that `CollateralAttestationV2` version handles `1e18` precision, while `PolicyManagerV2` reduces to `1e6` and then expands to `1e18` after calculation.

`CollateralAttestationV2`:

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

    uint256 R = IERC20(usdc).balanceOf(liquidityReserve);
    uint256 scaledR = _scaleToEighteen(R, reserveAssetDecimals);
    uint256 haircutValue = Math.mulDiv(HC, V, PRECISION);

    uint256 numerator = scaledR + haircutValue + cashInFlight;

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

`PolicyManagerV2`:

```
function getCollateralRatio() public view returns (uint256 cr) {

    if (buck == address(0) || collateralAttestation == address(0)) {
        return 1e18; // Default to 100% CR if not configured
    }

    uint256 supply = IBuckV2(buck).totalSupply();
    if (supply == 0) {
        return 1e18; // Bootstrap: 100% CR when no supply
    }

    uint256 assets = ICollateralAttestationV2(collateralAttestation).totalAssets();
    return (assets * 1e30) / supply;
}

/// @return Total in reserve decimals (6 for USDC)
 /// @dev Used by PolicyManagerV2 for CR calculation
 function totalAssets() external view returns (uint256) {
    uint256 R = IERC20(usdc).balanceOf(liquidityReserve); // Raw 6 decimals
    // Scale 18-decimal values down to 6 decimals
    uint256 haircutValue = Math.mulDiv(HC, V, PRECISION); // 18 decimals
    uint256 scaledHaircut = haircutValue / (10 ** (18 - reserveAssetDecimals));
    uint256 scaledCIF = cashInFlight / (10 ** (18 - reserveAssetDecimals));
    return R + scaledHaircut + scaledCIF;
}
```

Recommendation: Consider reusing `CollateralAttestationV2.getCollateralRatio()`.

Buck Labs: Fixed in commit aa036d5

Cantina Managed: Fix verified. `PolicyManagerV2` now delegates `getCollateralRatio()` to `CollateralAttestationV2`.

5.4.7 Logic with `OracleAdapterV4._isSuspiciousPriceJump()` can be removed

Severity: Informational

Context: OracleAdapterV4.sol#L303-L308

Description: Function `_fetchPriceWithStateUpdate()` always triggers CB if both prices are fresh and deviation is high:

```
function _fetchPriceWithStateUpdate() internal returns (uint256 price, uint256 updatedAt) {
    // Try all oracles
    (uint256 redstonePrice, uint256 redstoneTime, bool redstoneFresh) = _tryRedstone();
    (uint256 pythPrice, uint256 pythTime, bool pythFresh) = _tryPyth();

    // Cross-validation when both are fresh
    if (redstoneFresh && pythFresh) { // <<<
        uint256 deviation = _calculateDeviation(redstonePrice, pythPrice);

        if (deviation > maxOracleDeviationBps) { // <<<
            _tripCircuitBreaker(redstonePrice, pythPrice, deviation, "ORACLE_DIVERGENCE"); //
            ↪  <<<
        }
    }

    // State machine transitions
    if (status == Status.RedstoneWorking) {
        return _handleRedstoneWorkingState(redstonePrice, redstoneTime, redstoneFresh,
        ↪  pythPrice, pythTime, pythFresh); // <<<
```

```
    } else if (status == Status.UsingPythRedstoneUntrusted) {
        return _handleUsingPythState(redstonePrice, redstoneTime, redstoneFresh, pythPrice,
        ↪  pythTime, pythFresh);
    } else {
        return _handleUsingInternalState(redstonePrice, redstoneTime, redstoneFresh,
        ↪  pythPrice, pythTime, pythFresh);
    }
}
```

Then it executes `_handleRedstoneWorkingState()` which once again triggers CB if those conditions are met:

```
function _handleRedstoneWorkingState(
    uint256 redstonePrice,
    uint256 redstoneTime,
    bool redstoneFresh,
    uint256 pythPrice,
    uint256 pythTime,
    bool pythFresh
) internal returns (uint256 price, uint256 updatedAt) {
    if (redstoneFresh) { // <<<
        // Check for suspicious price jump
        if (_isSuspiciousPriceJump(redstonePrice)) {
            // Verify against Pyth if available
            if (pythFresh && _calculateDeviation(redstonePrice, pythPrice) >
            ↪  maxOracleDeviationBps) { // <<<
                _tripCircuitBreaker(redstonePrice, pythPrice,
                ↪  _calculateDeviation(redstonePrice, pythPrice), "PRICE_JUMP_UNCONFIRMED");
                ↪  // <<<
            }
        }


        // ...
    }
```

Recommendation: You can remove `lastSavedPrice`, `lastPriceUpdateTime`, `_isSuspiciousPriceJump()` logic.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. `OracleAdapterV5` removes the V4 price-jump logic and redundant CB checks.

5.4.8 Stale documentation in `OracleAdapterV4.sol`

Severity: Informational

Context: OracleAdapterV4.sol#L61

Description: It mentions 3 conditions when CB will be tripped:

```
/**
 * @title OracleAdapterV4
 * @notice Three-tier oracle with Liquity-style state machine and cross-validation
 *
 * ...
 *
 * Circuit Breaker:
   * - Trips on: overnight gap + deviation, oracle divergence, or both oracles failed // <<<

 * - Blocks redemptions, allows mints
 * - Manual reset required
 */
```

However actual code doesn't trigger CB when both oracles failed or on deviation + overnight gap:

```
/// @notice Fallback to internal price or lastGoodPrice
/// @dev NO circuit breaker trip for staleness - CB is only for oracle DIVERGENCE // <<<
///      Staleness is expected (weekends/after-hours), divergence is suspicious (manipulation)

function _fallbackToInternalOrLastGood(string memory reason)
    internal
    returns (uint256 price, uint256 updatedAt)
{
    _changeStatus(Status.UsingInternalBothUntrusted, reason);

    // Try internal price first (manual override, fresh within 1 hour)
    if (_internalPrice != 0 && block.timestamp <= _internalUpdatedAt + 1 hours) {
        _updateLastGoodPrice(_internalPrice, "INTERNAL");
        return (_internalPrice, _internalUpdatedAt);
    }

    // Fall back to lastGoodPrice - this is the pinned price from market close
    // NO circuit breaker trip - staleness is expected behavior for weekends/after-hours
    // lastGoodPrice was the last verified price from Redstone, Pyth, or Internal
    return (lastGoodPrice, lastGoodPriceTime);
}

function _handleRedstoneWorkingState(
    uint256 redstonePrice,
    uint256 redstoneTime,
    bool redstoneFresh,
    uint256 pythPrice,
    uint256 pythTime,
    bool pythFresh
) internal returns (uint256 price, uint256 updatedAt) {
    if (redstoneFresh) {
        // Check for suspicious price jump
        if (_isSuspiciousPriceJump(redstonePrice)) { // <<<
            // Verify against Pyth if available
            if (pythFresh && _calculateDeviation(redstonePrice, pythPrice) >
            ↪  maxOracleDeviationBps) { // <<<
                _tripCircuitBreaker(redstonePrice, pythPrice,
                ↪  _calculateDeviation(redstonePrice, pythPrice), "PRICE_JUMP_UNCONFIRMED");
            }
        }

        // ...
    }
}
```

Same is described in `buckArch.md`:

> CB: trips ONLY on oracle divergence (>5%); staleness uses lastGoodPrice without tripping CB.

Recommendation: Consider updating Natspec.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified via scope change. `OracleAdapterV5` replaces V4 in the upgrade path, so the stale V4 NatSpec is no longer relevant to production usage.

5.4.9 Runbook uses missing `OracleAdapter getPrice`

Severity: Informational

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

Description: The runbook validates the oracle by calling `getPrice` on the `OracleAdapterV4` deployment. That function does not exist in V4, so the call will revert and the validation step cannot succeed. Operators following the runbook may interpret the failure as a deployment issue even when the oracle is configured correctly, or may skip validation entirely.

```
cast call $ORACLE_V4 "getPrice()"
```

Recommendation: Replace the validation step with calls to `latestPrice` for a view only check and `refreshPrice` for a state updating check when appropriate.

```
cast call $ORACLE_V4 "latestPrice()"
cast send $ORACLE_V4 "refreshPrice()" --from $ADMIN_WALLET
```

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. Runbook verification now uses `OracleAdapterV5` `latestPrice/getOracleStatus` instead of missing `getPrice`.

5.4.10 Runbook uses an incorrect signature for `setDailyCapPct` call

Severity: Informational

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

Description: The upgrade runbook configures the daily cap using a cast call with a `uint16` argument. The on chain function expects a `uint8`, so the selector does not match and the transaction will revert. If operators follow the runbook as written, the daily cap update will fail and the system will continue using whatever default or previously set cap value exists. This can silently leave the protocol operating under unintended limits.

```
cast send $POLICY_MANAGER_PROXY "setDailyCapPct(uint16)" 66 --from $ADMIN_WALLET
```

Recommendation: Update the runbook to use the correct selector with a `uint8` argument and add a post call verification of the stored cap value.

```
cast send $POLICY_MANAGER_PROXY "setDailyCapPct(uint8)" 66 --from $ADMIN_WALLET
cast call $POLICY_MANAGER_PROXY "dailyCapPct()"
```

Buck Labs: Fixed in commit 7ea160d.

Cantina Managed: Fix verified. Runbook updated to correct `setDailyCapPct(uint8)` signature.

5.4.11 Runbook calls missing `treasurerWithdraw`

Severity: Informational

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

Description: The runbook tests a treasurer withdrawal by calling `treasurerWithdraw` on `LiquidityReserveV2`. That function is not present in V2, so the call will revert and the test cannot validate cash in flight tracking. If this step is used as a go live check, it may block or delay the upgrade despite the reserve being correctly configured.

```
cast send $LIQUIDITY_RESERVE_PROXY "treasurerWithdraw(uint256)" 1000000000 --from
↪  $TREASURY_WALLET
```

Recommendation: Use the `queueWithdrawal` entrypoint which routes treasurer initiated withdrawals through the intended fast path and preserves cash in flight tracking.

```
cast send $LIQUIDITY_RESERVE_PROXY "queueWithdrawal(address,uint256)" \
  $TREASURY_WALLET 1000000000 \
  --from $TREASURY_WALLET
```

Buck Labs: Fixed in commit a58fb86.

Cantina Managed: Fix verified. Runbook now uses `queueWithdrawal` for treasurer CIF test instead of missing `treasurerWithdraw`.

5.4.12 Runbook checks attestation freshness incorrectly

Severity: Informational

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

Description: The runbook checks `lastAttestationTime` to decide whether collateral data is fresh. Pricing and refunds gate on isAttestationStale, which evaluates attestationMeasurementTime against configured staleness thresholds. A recent submission time can still be stale if the measurement timestamp is old. This mismatch can allow the runbook to proceed while pricing later reverts due to stale attestation data, causing unexpected downtime at go live.

```
cast call $COLLATERAL_ATTESTATION_PROXY "lastAttestationTime()"
```

Recommendation: Update validation to query the same predicate used on chain and include the underlying measurement and thresholds for operator visibility.

```
cast call $COLLATERAL_ATTESTATION_PROXY "isAttestationStale()"
cast call $COLLATERAL_ATTESTATION_PROXY "attestationMeasurementTime()"
cast call $COLLATERAL_ATTESTATION_PROXY "healthyStaleness()"
cast call $COLLATERAL_ATTESTATION_PROXY "stressedStaleness()"
```

Buck Labs: Fixed in commit b60d0b5.

Cantina Managed: Fix verified. Runbook now checks `isAttestationStale` and surfaces measurement time and thresholds.

5.4.13 Redundant `address(0)` checks in `mint()` function

Severity: Informational

Context: BuckV2.sol#L269

Description: The `mint()` function in BuckV2.sol validates if the `to` address is not a zero address. However, this validation is done by the internal `_mint()` function, making this redundant.

Recommendation: Consider removing the redundant address zero check.

Buck Labs: Fixed in commit e287818.

Cantina Managed: Fix verified. Redundant zero-address check removed. OZ `_mint` handles `address(0)` reverts.

5.4.14 Primary/DEX price gap arbitrage

Severity: Informational

Context: LiquidityWindowV2.sol#L408-L409, LiquidityWindowV2.sol#L552-L567

Description: `LiquidityWindowV2` mints and refunds BUCK strictly at the CAP price computed by PolicyManagerV2, with a fixed spread and fee schedule. The contract does not reference any secondary market pricing, so the primary market price can diverge materially from DEX price during volatility or liquidity shocks. When BUCK trades below CAP on a DEX, allowlisted users can buy cheaply on the DEX and redeem at CAP, extracting USDC

from the reserve. When BUCK trades above CAP, allowlisted users can mint at CAP and sell on the DEX, increasing circulating supply and pressuring the collateral ratio. Both paths are pure user actions and do not require oracle manipulation or privileged access.

```
(capPrice, mintFee, halfSpread, isPaused, cbTripped) = getMintParams();
(capPrice, refundFee, halfSpread, isPaused, cbTripped) = getRefundParams();
```

The protocol relies on the daily cap and spread to throttle these flows, but large price gaps can still result in persistent arbitrage pressure and reserve outflows over time.

Recommendation: If the intent is to limit primary/secondary market arbitrage, add an optional DEX price guard such as a TWAP-based deviation check that pauses or widens spreads when the DEX price diverges beyond a threshold. Alternatively, document that this arbitrage is an expected stabilization mechanism and size the daily cap and spreads to tolerate sustained gaps.

Buck Labs: Fixed in commit 015e677.

Cantina Managed: Fix verified. Docs now frame primary/DEX arbitrage as an intentional stabilization mechanism.

5.4.15 Batch claim can revert whole batch

Severity: Informational

Context: RewardsEngineV1_1.sol#L383-L384, RewardsEngineV1_1.sol#L488

Description: `RewardsEngineV1_1.batchClaimFor` iterates through users and calls `_executeClaimForSafe` for each. That helper only suppresses the "no rewards" condition, but it does not isolate other failures. Any revert inside `_executeClaimForSafe` or the downstream token mint reverts the entire batch, so a single failing recipient can block all claims in that transaction.

In the V1 token, mint enforces access checks on the recipient. If any recipient is no longer allowed or is denylisted, the mint will revert and the batch will fail. This creates a migration risk where one ineligible address can halt batch execution and require manual splitting or filtering.

```
function batchClaimFor(address[] calldata users)
    external
    onlyClaimAdmin
    returns (uint256 totalClaimed)
{
    // ...
    claimed = _executeClaimForSafe(user, user);
    // ...
}

function _executeClaimForSafe(address user, address recipient) internal returns (uint256
↪   amount) {
    // ...
    _mintRewards(recipient, amount);
}
```

```
function mint(address to, uint256 amount) external nonReentrant onlyMinter whenNotPaused {
    if (to == address(0)) revert ZeroAddress();
    _enforceAccess(to);
    _mint(to, amount);
}
```

Recommendation: Add a per-user try/catch and emit an event when a claim fails so the batch can continue. If you want to keep strict behavior, pre-filter the user list with access checks and pending rewards before calling `batchClaimFor` and document that any failing recipient will revert the entire batch.

Buck Labs: Acknowledged. We have actually not deny listed anybody yet and the fork tests where we distribute to everyone who has pending rewards currently passes. If the day comes for the upgrade and an address is messing

us up, I'll just exclude them from the list.

Cantina Managed: Acknowledged.

5.4.16 Runbook misstates distribute flow

Severity: Informational

Context: buckUpgradeProcedure.md#L317-L319

Description: The upgrade runbook in `docs/buckUpgradeProcedure.md` states that distribute mints BUCK to the `distributor` wallet and instructs operators to verify a `RewardsDistributed` event and a USDC transfer to `RewardsEngine`. In the V1 `RewardsEngine`, distribute instead pulls USDC directly into the `LiquidityReserve` and emits `DistributionPriced`, `DistributionSkimCollected` and `DistributionDeclared` and only the breakage sink share is minted immediately to `breakageSink` or `treasury`. This mismatch can cause operators to whitelist the wrong address and to treat a successful distribution as failed, or to overlook that the actual mint recipient must be permitted when `accessRegistry` is enforced. If the breakage sink or treasury is not allowed to receive mints, the distribution step can revert and block the migration.

```
IERC20(reserveUSDC).safeTransferFrom(msg.sender, liquidityReserve, couponUsdcAmount);
```

Recommendation: Update the runbook to match actual behavior by verifying the correct events, confirming USDC lands in `LiquidityReserve` and ensuring the breakage sink or treasury is allowed to receive BUCK mints rather than the distributor wallet.

Buck Labs: Fixed in commit 265a53c.

Cantina Managed: Fix verified. Runbook now correctly states distribute moves USDC to `LiquidityReserve` and only mints breakage to treasury/breakage sink.

5.4.17 Runbook missing USDC approve step

Severity: Informational

Context: RewardsEngine.sol#L619-L620, buckUpgradeProcedure.md#L309-L315

Description: The runbook instructs calling `distribute` from the distributor wallet but does not include a prerequisite approval of USDC to the `RewardsEngine`. The distribute function pulls USDC using `safeTransferFrom`, so it will revert unless the distributor has granted an allowance. If operators follow the runbook as written, the distribution step can fail and delay the upgrade sequence.

```
IERC20(reserveUSDC).safeTransferFrom(msg.sender, liquidityReserve, couponUsdcAmount);
```

Recommendation: Add an explicit USDC approve step for the distributor wallet before calling distribute and include a quick allowance check to confirm the approval matches the coupon amount.

Buck Labs: Fixed in commit 04fe600.

Cantina Managed: Fix verified. Runbook now includes USDC approve + allowance verification before distribute.

5.4.18 Circuit breaker needs manual reset

Severity: Informational

Context: OracleAdapterV4.sol#L258-L264

Description: Once the circuit breaker is tripped, it remains active until an owner calls `resetCircuitBreaker`. There is no automatic recovery path when oracles become healthy again. This means the protocol can remain in a halted state indefinitely if the owner is unavailable, delayed by governance processes, or operationally slow to respond.

Because the circuit breaker is a safety mechanism, requiring manual intervention to restore liveness introduces a central operational dependency. In emergency conditions, this can prolong the halt beyond what is economically necessary and can compound user impact if refunds or mints are blocked.

```
function resetCircuitBreaker() external onlyOwner {
    if (!circuitBreakerTripped) revert CircuitBreakerNotTripped();
    circuitBreakerTripped = false;
    emit CircuitBreakerReset(block.timestamp, msg.sender);
}
```

A manual-only reset is a single point of liveness control for the system.

Recommendation: Add an off-chain keeper or automation script that periodically calls `refreshPrice` and only calls `resetCircuitBreaker` after the oracle set is fresh and within deviation thresholds across multiple consecutive checks. This removes the manual liveness bottleneck while keeping recovery policy conservative. If manual reset is still required by policy, document the runbook and expected recovery SLAs.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. Circuit breaker is removed in V5, so no manual reset dependency remains.

5.4.19 Circuit breaker mint mismatch

Severity: Informational

Context: LiquidityWindowV2.sol#L564, OracleAdapterV4.sol#L62

Description: The oracle adapter documentation states that the circuit breaker blocks redemptions but allows mints. In the mint flow, however, `LiquidityWindowV2` treats a tripped circuit breaker as a hard revert. This is a semantic mismatch between the oracle contract's stated behavior and the actual primary market behavior. If operators expect mints to remain available during circuit breaker events, the current implementation can unexpectedly halt minting and disrupt recovery or stabilization flows.

```
(uint256 capPrice, uint16 mintFeeBps_, uint16 halfSpreadBps, bool isPaused, bool cbTripped) =
    IPolicyManagerV2(policyManager).getMintParams();

if (isPaused) revert EcosystemPaused();
if (cbTripped) revert CircuitBreakerTripped();
```

This difference between documented intent and enforcement can cause liveness failures and some operational confusion, especially in emergency conditions.

Recommendation: Align the mint flow with the intended circuit breaker semantics. Either allow mints when the CB is tripped, or update the oracle and operational documentation to reflect that mints are blocked in this state.

Buck Labs: Fixed in commit f40ce28.

Cantina Managed: Fix verified. Circuit breaker plumbing was removed from V5. `LiquidityWindow` no longer blocks mints based on CB state.

5.4.20 Rewards epoch math is order sensitive

Severity: Informational

Context: RewardsEngineV1_1.sol#L716-L738

Description: The `RewardsEngine` uses lazy settlement across epochs, relying on `epochReport`, `accRewardPerUnit` and `currentEpochId` to backfill missed accruals. This requires strict ordering of epoch configuration, distribution and balance-change updates. If a distribution is skipped, delayed, or occurs out of order, the epoch transition logic can produce inconsistent `unitsAccrued`, `pendingRewards` and global indices.

This makes accounting state sensitive to operational sequencing rather than purely to balances and time, which can break invariants about total distributed rewards matching aggregate accrual.

```
if (s.lastAccruedEpoch > 0 && s.lastAccruedEpoch < currentEpochId) {
    EpochReport storage report = epochReport[s.lastAccruedEpoch];
    // ...
    if (report.deltaIndex > 0 && s.unitsAccrued > 0) {
        s.pendingRewards += Math.mulDiv(s.unitsAccrued, report.deltaIndex, ACC_PRECISION);
    }
    // ...
    s.lastAccruedEpoch = currentEpochId;
}
```

Recommendation: Add explicit invariants and guards around epoch transitions and consider enforcing an operational sequence in code (for example, forbidding a new epoch start until the previous distribution is finalized).

Buck Labs: Acknowledged as V1 design characteristic; V2 eliminates epoch-based rewards entirely via yield streaming, making this moot after migration.

Cantina Managed: Acknowledged.

5.4.21 Rewards accounting relies on hooks

Severity: Informational

Context: Buck.sol#L484-L487, RewardsEngineV1_1.sol#L663-L673

Description: Rewards accounting is driven by the token calling `onBalanceChange` on every mint, burn, and transfer. If the hook is not wired, is disabled, or is bypassed for any reason, the `RewardsEngine`'s internal balances and accrued units diverge from actual token balances. The accounting then becomes inconsistent, which can distort rewards distribution and claims.

This design couples accounting correctness to an external callback, making invariants dependent on call ordering and configuration rather than on direct state reads.

```
function _update(address from, address to, uint256 value) internal override {
    // ...
    super._update(from, to, value);
    _notifyRewards(from, to, value);
}

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

Recommendation: Where possible, derive reward state from observable token balances or add reconciliation mechanisms that detect and correct divergence when hooks are missed.

Buck Labs: Acknowledged. This is v1 code and will go away with the upgrade.

Cantina Managed: Acknowledged.

5.4.22 PM upgrade before LW breaks calls

Severity: Informational

Context: LiquidityWindow.sol#L296-L297, LiquidityWindow.sol#L319-L322, buckUpgradeProcedure.md#L382-L389, buckUpgradeProcedure.md#L404-L418

Description: The upgrade runbook applies the `PolicyManager` V2 upgrade before the `LiquidityWindow` V2 upgrade. During that window, the live `LiquidityWindow` V1 contract continues to call `PolicyManager` using the V1 interface, while the proxy now points to a V2 implementation. The V1 `LiquidityWindow` expects methods like `getMintParameters`, `getRefundParameters`, `checkMintCap` and `checkRefundCap` to exist and return the V1 struct layout. `PolicyManagerV2` does not implement those selectors. If users interact during this window, mint and refund operations will revert or behave unexpectedly because the interface contract has changed out from under the caller.

```
function getMintParameters(uint256 amountTokens) external view returns (MintParameters memory);
function getRefundParameters(uint256 amountTokens) external view returns (MintParameters
↪  memory);
```

This creates a publicly visible downtime window unless the system is paused or the sequence is executed atomically. The primary market can be unavailable and can also surface confusing or inconsistent behavior while mixed versions are live.

Recommendation: Pause the `LiquidityWindow` and `PolicyManager` before the `PolicyManager` upgrade and only unpause after `LiquidityWindow` is upgraded and configuration is complete. If possible, execute the `PolicyManager` and `LiquidityWindow` upgrades as a private bundle so there is no interleaving window. An alternative is to upgrade `LiquidityWindow` first or introduce a temporary compatibility shim that keeps the V1 selectors alive until both upgrades are complete.

Buck Labs: Fixed in commit 8d35509.

Cantina Managed: Fix verified. Runbook now requires pausing `LiquidityWindow` before `PolicyManager` upgrade to prevent interface mismatch failures.

5.4.23 CIF tracking revert blocks withdrawals

Severity: Informational

Context: LiquidityReserveV2.sol#L361-L363, LiquidityReserveV2.sol#L370-L375

Description: `LiquidityReserveV2` tracks cash-in-flight by calling `CollateralAttestationV2.addCashInFlight` during treasurer and admin withdrawals. This call is performed inline and is not wrapped in try/catch. If the collateral attestation address is misconfigured or the attestation contract reverts, the entire withdrawal reverts and the treasury cannot move funds off-chain.

Because cash-in-flight tracking is a secondary accounting update, a temporary attestation outage can freeze critical treasury operations. This turns a non-critical subsystem into a hard dependency for withdrawals.

```
function _instantWithdrawal(address to, uint256 amount, bool trackCIF) internal {
    // ...
    asset.safeTransfer(to, amount);
    if (trackCIF) {
        _trackCashInFlight(amount);
    }
    // ...
}

function _trackCashInFlight(uint256 amount) internal {
    if (collateralAttestation != address(0)) {
        ICollateralAttestationV2(collateralAttestation).addCashInFlight(amount);
        emit CashInFlightTracked(amount);
    }
}
```

Recommendation: Decide whether availability or strict accounting is the priority. If availability matters, wrap the CIF call in try/catch and emit an event on failure so off-chain systems can reconcile later. If strict accounting is required, add operational tooling and pre-flight checks to ensure the collateral attestation contract is configured and responsive before withdrawals are allowed.

Buck Labs: Fixed in commit 8475bc7.

Cantina Managed: Fix verified. CIF tracking now uses try/catch and emits `CashInFlightTrackingFailed`, so withdrawals no longer revert if CIF update fails.

5.4.24 `LiquidityWindow` pause does not block refunds

Severity: Informational

Context: LiquidityWindowV2.sol#L391-L396, LiquidityWindowV2.sol#L534-L539

Description: `LiquidityWindowV2` has a local pause mechanism, but the refund entry point does not honor it. The mint path is protected by `whenNotPaused`, while `requestRefund` is not, so a local pause only blocks mints. Refunds only check the ecosystem pause from `PolicyManagerV2`, which means a `LiquidityWindow` pause alone does not stop redemptions.

If the `LiquidityWindow` owner expects the local pause to halt all primary market activity during an incident, refunds can still drain reserve funds as long as `PolicyManagerV2` is not paused. This undermines the emergency response playbook and can extend the impact of an oracle or collateral incident.

```
function requestRefund(...)
    external
    nonReentrant
    returns (uint256 usdcOut, uint256 feeUsdc)
{
    // ...
}

function _executeMint(...)
    internal
    whenNotPaused
    returns (uint256 buckOut, uint256 feeUsdc)
{
    // ...
}
```

Recommendation: Gate `requestRefund` with `whenNotPaused` or add an explicit local pause check at the top of the function. If the intended design is for refunds to remain live during a `LiquidityWindow` pause, document that clearly and consider adding a separate explicit refund pause to avoid operator confusion during incidents.

Buck Labs: Fixed in commit 487eb88.

Cantina Managed: Fix verified. `requestRefund` is now `whenNotPaused`, so `LiquidityWindow` pause blocks refunds.

5.4.25 CIF not tracked during reserve upgrade

Severity: Informational

Context: buckUpgradeProcedure.md#L440-L449, LiquidityReserveV2.sol#L368-L375

Description: The upgrade runbook upgrades `LiquidityReserve` to V2 and then sets the `collateralAttestation` address in a separate transaction. `LiquidityReserveV2` only tracks cash in flight when `collateralAttestation` is nonzero. If a treasurer withdrawal or an admin queued withdrawal executes between those two transactions, the reserve outflow will not be recorded as cash in flight and the collateral ratio will be overstated until a manual correction is made.

```
function _trackCashInFlight(uint256 amount) internal {
    if (collateralAttestation != address(0)) {
        ICollateralAttestationV2(collateralAttestation).addCashInFlight(amount);
        emit CashInFlightTracked(amount);
```

```
    }
}
```

This creates a correctness gap in a mempool visible window. Overstated collateral can influence mint and refund pricing if the collateral ratio is used downstream.

Recommendation: Set `collateralAttestation` in the same transaction as the `LiquidityReserve` upgrade, or pause treasurer and admin withdrawals until the configuration call is complete. If atomic configuration is not possible, execute both transactions as a private bundle and add a post upgrade check to confirm `collateralAttestation` is set before any withdrawals.

Buck Labs: Fixed in commit 8d35509.

Cantina Managed: Fix verified. Upgrade runbook now pauses `LiquidityWindow` before the `LiquidityReserve` upgrade gap, preventing withdrawals before CIF tracking is configured.