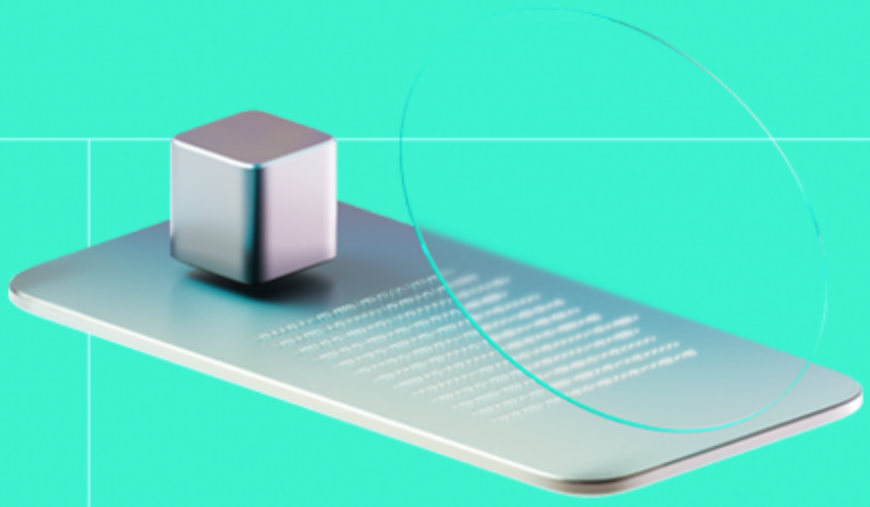
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Upshift Finance

Date: 07/04/2026

We express our gratitude to the Upshift Finance team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Upshift Finance Atomic Flow is an upgradeable multi-asset vault system that lets authorized users deposit supported assets, receive LP shares, and redeem into a reference asset using oracle-based valuation and timelocked governance controls.

Document

Name	Smart Contract Code Review and Security Analysis Report for Upshift Finance
Audited By	Ivan Bondar; Khrystyna Tkachuk
Approved By	Kornel Światłowski
Website	https://www.upshift.finance/
Changelog	20/03/2025 - Preliminary Report
	07/04/2026 - Final Report
Platform	Ethereum
Language	Solidity
Tags	Vault; Yield Farming; Centralization; Upgradable
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/fractal-protocol/august-contracts-v2/
Commit	0445312
Final Commit	f282f65

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

16	11	3	2
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	5
Low	5

Vulnerability	Severity	Status
F-2026-15376 - Oracle Price Conversion Formula in EnableOnlyAssetsWhitelist Produces Incorrect Results	High	Fixed
F-2026-15400 - Operator Can Cancel and Pre-Schedule Timelocked Hashes, Vetoing All Governance Operations	Medium	Fixed
F-2026-15407 - No L2 Sequencer Uptime Check in Oracle Validation, Risking Stale Prices on L2 Deployments	Medium	Fixed
F-2026-15436 - Untracked Subaccount Composition Drift Blocks LP Redemptions and Enables Owner Extraction	Medium	Accepted
F-2026-15469 - withdrawFromSubaccount Price Divergence Causes Custody Lockout and Phantom NAV	Medium	Fixed
F-2026-15474 - Immutable operatorAddress Lacks a Mechanism to Replace a Compromised Operator Enabling Persistent NAV Distortion	Medium	Fixed
F-2026-15389 - Post-Emergency externalAssets Correction Throttled by Rate Limiter	Low	Fixed
F-2026-15394 - updateAssetsWhitelist Lacks Timelock, Enabling Instant Replacement of the Pricing Oracle Layer	Low	Accepted
F-2026-15398 - updatePerformanceFee Has No Upper Bound and No Timelock	Low	Mitigated
F-2026-15420 - Several Owner-Only Functions Are Not Protected by Timelocks	Low	Mitigated
F-2026-15438 - Lack of Cross-System Validation Between Subaccount Redeemable Assets and Vault Whitelist	Low	Fixed

Vulnerability	Severity	Status
F-2026-15416 - Multiple Admin Functions Emit No Events, Hindering Off-Chain Monitoring	Info	Fixed
F-2026-15421 - _owner Is Assigned Twice, in initialize and configure, Potentially with Different Values	Info	Fixed
F-2026-15427 - Floating Pragma	Info	Accepted
F-2026-15441 - Inconsistent Storage-Gap Strategy Increases Upgrade Collision Risk	Info	Fixed
F-2026-15450 - Deposits in Non-Reference Assets May Not Be Immediately Redeemable via the Instant Redemption Flow	Info	Fixed

Documentation quality

- Functional requirements are detailed.
  - Project overview is detailed
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract, all futures are described.
  - All interactions are described.
- Technical description is detailed.
  - Run instructions are provided.
  - Technical specification is provided.
  - The NatSpec documentation is sufficient.

Code quality

- The code duplicates commonly known contracts instead of reusing them.
- The development environment is configured.

Test coverage

Code coverage of the project is **34.29%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missed.
- Interactions by several users are tested.

Table of Contents

System Overview

Upshift Finance Atomic Flow is an upgradeable multi-asset vault system that lets authorized users deposit supported assets, receive LP shares, and redeem into a reference asset using oracle-based valuation and timelocked governance controls. This scope covers the Atomic Flow vault stack within the Upshift protocol, centered on **TokenizedVault**, with **TimelockedVault**, **OraclizedMultiAssetVault**, and **OperableVault** providing the redemption, valuation, and operational layers beneath it. Users deposit the reference asset or approved auxiliary assets, receive LP shares, and redeem back into the reference asset through an atomic instant-redemption flow.

The vault's NAV is derived from on-vault balances plus a manually reported `externalAssets` value, while non-reference assets are priced through **EnableOnlyAssetsWhitelist** using external oracle feeds. Governance is split between a local owner, an operator, and **ResourceBasedTimelockedCall**, which timelocks selected privileged actions. The scoped contracts therefore form the core Atomic Flow vault module of Upshift rather than a standalone product: **TokenizedVault** is the concrete upgradeable vault, **EnableOnlyAssetsWhitelist** supplies asset admission and pricing, and **GuardedProxyOwnable2Steps** / **OwnableGuarded** / **BaseReentrancy** provide the ownership and reentrancy foundations.

Files In Scope

- **TokenizedVault**: Concrete upgradeable vault implementation with bootstrap configuration, performance-fee logic, share-price tracking, and emergency withdrawal.
- **TimelockedVault**: Atomic redemption layer with instant redeem, withdrawal/deposit limits, redemption-fee handling, and instant-redemption subaccount wiring.
- **OraclizedMultiAssetVault**: Multi-asset deposit, valuation, fee accrual/collection, external-assets accounting, and subaccount allocation flows.
- **OperableVault**: Core vault governance surface for owner/operator permissions, sender whitelist, fee collectors, pauses, and subaccount registry.
- **EnableOnlyAssetsWhitelist**: One-way asset whitelist tied to a single immutable reference asset, with oracle freshness checks and valuation/share-conversion helpers.
- **ResourceBasedTimelockedCall**: External timelock queue keyed by call hash for a specific vault resource. Used for non-atomic vaults deployments.
- **RestrictedTimelockedCall**: An external timelock queues operations using a call hash tied to a specific vault resource. This timelock implementation is used for atomic vaults and excludes the operator from `_enforceValidSender`. Only the resource (vault) and its owner may schedule or cancel queued hashes.
- **GuardedProxyOwnable2Steps**: Initializer-aware ownership base with configured/not-configured gates and pending-owner acceptance flow.
- **OwnableGuarded**: Minimal local ownership primitive plus a built-in non-reentrancy guard.
- **SendersWhitelist**: Simple owner-managed boolean sender allowlist.
- **BaseReentrancy**: Standalone reentrancy guard used by the timelock helper.

Privileged roles

TokenizedVault:

- `owner`: Can configure the vault, update performance fees, update operator, pause or resume deposits and withdrawals, update sender and asset whitelist references, update fee collector configuration, update instant redemption fees and vault limits, manage subaccounts, set the `instantRedemptionSubaccount`, initiate ownership transfer, and execute emergency withdrawal.
- `operator`: Can update `externalAssets` and move assets between the vault and whitelisted subaccounts or whitelisted wallets.
- `owner or operator`: Can perform operational asset-allocation actions such as `updateTotalAssets`, `depositToSubaccount`, and `withdrawFromSubaccount`.
- `pending owner`: Can accept ownership after the transfer is proposed and the required timelock flow is satisfied.
- `whitelisted sender`: Can access deposit and redemption entrypoints when a sender whitelist is configured.
- `public`: Can call `chargeManagementFee()`, `collectFees()`, and `chargePerformanceFees()`.

EnableOnlyAssetsWhitelist:

- `owner`: Can enable new assets, assign oracle feeds, update oracle staleness limits, and transfer ownership.

ResourceBasedTimelockedCall:

- `resource owner`: Can schedule and cancel timelocked call hashes.
- `resource operator`: Can also schedule and cancel timelocked call hashes.
- `resource contract`: The bound vault contract is the only address that can consume a matured hash.

SendersWhitelist:

- `owner`: Can add or remove whitelisted senders and transfer ownership.

Potential Risks

Out-of-Scope Components & Third-Party Dependencies

- **Dependency on Morpho V2 ERC-4626 Vault:** The vault system routes reference assets through a subaccount that deploys them into a Morpho V2 ERC-4626 idle vault for yield. The vault's liquidity for redemptions, fee collection, and emergency withdrawals depends on the ability to withdraw from this Morpho position. Any issue with the Morpho V2 vault, including liquidity constraints, pauses, or vulnerabilities, could prevent the vault from fulfilling redemptions or recovering funds during emergencies.
- **Dependency on Chainlink Oracle Infrastructure:** The **EnableOnlyAssetsWhitelist** contract relies on Chainlink **AggregatorV3Interface** oracles for all non-reference-asset price conversions used in NAV computation and share pricing. Oracle downtime, stale data delivery, or feed manipulation would directly affect the vault's valuation and all deposit/withdrawal calculations.
- **L2 Sequencer Downtime Halts All NAV-Dependent Operations:** On L2 deployments, the **EnableOnlyAssetsWhitelist** validates the Chainlink sequencer uptime feed before every oracle price conversion. When the sequencer is down or within the configured grace period after restart, `_fromInputAssetToReferenceAsset` reverts, which propagates through `_getTotalAssetsValuation`, `_getTotalAssets`, and `_convertToAssets`. As a result, all NAV-dependent operations (deposits, redemptions, fee collection, management fee charging, performance fee charging, `depositToSubaccount`, and `withdrawFromSubaccount`) revert for the duration of the sequencer outage plus the grace period. The `emergencyWithdraw` function is unaffected because it transfers balances directly without querying NAV. The vault owner can still pause and unpause deposits and withdrawals during this period. Normal operation resumes automatically once the sequencer has been online longer than the grace period.
- **Dependency on External LP Token and Whitelist Contracts:** The vault delegates share minting/burning to an external **IMintableBurnable** LP token contract and sender authorization to an external **ISendersWhitelist** or **ISendersAllocationWhitelist** contract. The vault trusts these contracts to behave correctly; compromise of either would affect share accounting or access control across the entire system.

Permissions, Authorization & Access

- **Operator Trust for NAV Reporting and Fund Movement:** The operator role is trusted to honestly report `externalAssets` via `updateTotalAssets` and to move vault funds to/from whitelisted subaccounts and wallets via `depositToSubaccount` and `withdrawFromSubaccount`. A compromised operator can influence share pricing through NAV reporting and move funds to owner-whitelisted wallet addresses where they are no longer under smart contract custody.
- **Operator Timelock Interaction Authority:** The operator can schedule and cancel hashes on the **ResourceBasedTimelockedCall** contract, giving the operator the ability to cancel pending timelocked governance operations scheduled by the owner.
- **Frontrunning of updateTotalAssets by Whitelisted Users:** The `updateTotalAssets` function changes `externalAssets` and therefore NAV in a single transaction visible in the mempool. Whitelisted users who monitor pending transactions can deposit before an upward NAV adjustment or redeem before a downward one, extracting value at the expense of other LP holders. The rate limiter bounds the per-call profit from such sandwiches but does not prevent

them. The sender whitelist, allocation limits, deposit/withdrawal caps, and redemption fee provide partial mitigation.

- **Rate Limiter Bypass on updateTotalAssets While Vault Is Fully Paused:** The `_enforceRateLimit` function on `updateTotalAssets` is bypassed when both `depositsPaused` and `withdrawalsPaused` are true. This provides operational flexibility for rapid NAV correction during emergency scenarios without requiring multi-day incremental updates. Since no user operations execute while the vault is fully paused, the bypass does not directly expose depositors or redeemers. The owner should verify `externalAssets` accuracy before calling `pauseDepositsAndWithdrawals` to resume operations, as unpausing without verification would allow deposits and redemptions against a potentially stale or incorrect NAV.

Centralization

- **Single Owner Authority Over Critical Parameters:** The vault owner has unilateral control over sensitive parameters including fees, deposit/withdrawal limits, whitelist addresses, subaccount configuration, fee collector addresses, and pause state. Only three operations (`transferOwnership`, `updateMaxChangePercent`, `updateManagementFee`) are subject to timelock delays; all other owner-only state changes take effect immediately.
- **Single-Entity Emergency Authority:** The `emergencyWithdraw` function is callable only by the owner and sweeps all vault assets to a single receiver address with no multi-signature requirement or timelock.

Upgradeability

- **Proxy Upgrade Authority and Contract Logic Replacement:** The **TokenizedVault** is deployed behind an upgradeable proxy. The proxy admin can replace the entire contract implementation, changing all vault logic. The timelock on `transferOwnership` does not extend to proxy upgrades themselves.
- **One-Time Configuration with No Reconfiguration Path:** Several critical addresses set during `configure` - `scheduledCallerAddress`, `operatorAddress`, `lpTokenAddress`, and `_referenceAsset` have no setter functions and cannot be changed post-configuration. Correcting a misconfiguration requires deploying a new implementation and upgrading the proxy.

External Asset Accounting

- **Static Principal Tracking for Dynamic Subaccount Positions:** The vault tracks funds deposited to subaccounts via a static `externalAssets` counter that records nominal principal amounts. It does not reflect yield accrual, losses, or fee dilution occurring within the subaccount's ERC-4626 idle vault position. Reconciliation depends entirely on the operator calling `updateTotalAssets` at appropriate intervals.
- **NAV-to-Redemption Liquidity Mismatch from Non-Reference Asset Deposits:** The vault accepts deposits in any whitelisted asset and values them at oracle prices for NAV and share pricing. However, redemptions are serviced exclusively in the reference asset. When a significant portion of vault NAV is backed by non-reference assets held on the vault balance or in subaccounts, the vault may be fully solvent by valuation while lacking sufficient reference-asset liquidity to fulfill redemptions. Converting non-reference assets into reference-asset liquidity requires administrative action (operator-driven subaccount routing or off-chain conversion) and is not automated.
- **Subaccount Composition Changes Without Vault Notification:** The **InstantRedeemSubaccount** allows whitelisted users to swap non-reference assets for reference

assets via `redeemAsset` without any callback to the parent vault. These swaps reduce the subaccount's reference-asset balance while increasing its non-reference asset holdings. The vault's `externalAssets` counter and the `_ensureLiquidity` function are unaware of this composition shift. The operator is expected to monitor subaccount composition off-chain and settle any drift via `withdrawFromSubaccount` or by pausing subaccount redemptions.

Backward Compatibility

- **Upgrade Incompatibility with Pending Lagged Claims:** As documented in the `configure` NatSpec, this implementation must not be used to upgrade vaults that have pending lagged withdrawal claims from a previous implementation. Doing so would strand LP tokens and underlying assets permanently, as the lagged withdrawal processing functions are now no-op stubs.

Findings

Vulnerability Details

F-2026-15376 - Oracle Price Conversion Formula in EnableOnlyAssetsWhitelist Produces Incorrect Results - High

Description:

The `_fromInputAssetToReferenceAsset` function in **EnableOnlyAssetsWhitelist** uses a complex branching formula with conditional overrides that fails for the majority of (`tokenDecimals`, `oracleDecimals`, `referenceDecimals`) combinations. Five combinations multiply by the oracle price instead of dividing (inverted), and nine others apply incorrect scaling factors. This directly corrupts share pricing, NAV computation, and deposit/withdrawal amounts. Additionally, the formula diverges from the **InstantRedeemSubaccount** simpler, correct implementation, creating an arbitrage vector.

The function computes a base result and then applies conditional patches:

```
// EnableOnlyAssetsWhitelist.sol
function _fromInputAssetToReferenceAsset(address assetAddr, uint256 amount) i
nternal view returns (uint256) {
    // ...
    uint256 d1 = (oracleDecimals > tokenDecimals) ? 10 ** (oracleDecimals - t
okenDecimals) : 1;
    uint256 d2 = (REFERENCE_ASSET_DECIMALS > oracleDecimals) ? 10 ** (REFEREN
CE_ASSET_DECIMALS - oracleDecimals) : 1;
    uint256 amountInNormalized = amount * d1;
    uint256 d3 = 10 ** (18 - oracleDecimals);
    uint256 ratio = (10 ** oracleDecimals) * d3 / uint256(quoteAnswer);
    uint256 result = ratio * amountInNormalized * d2 / d3;
    // ... conditional overrides for specific decimal combos ...
}
```

The correct approach, used by **InstantRedeemSubaccount**, normalizes both the price and the amount to 18 decimals, performs the division, and scales back down - no branching required:

```
normalizedPrice  = quoteAnswer * 10^(18 - oracleDecimals)
normalizedAmount = amount      * 10^(18 - tokenDecimals)
result           = (normalizedAmount * 1e18) / normalizedPrice / 10^(18 - ref
erenceDecimals)
```

The base formula only produces the correct result when `oracleDecimals >= tokenDecimals` and `referenceDecimals >= oracleDecimals`. Outside that range, the `d1` and `d2` multipliers default to `1` instead of scaling down, so the formula silently over- or under-scales the result. The conditional patches (lines 246-267) attempt to fix specific combinations but introduce worse errors.

Inverted-price bugs: Several conditional overrides reassign `ratio = quoteAnswer`, which flips the formula from dividing by the oracle price to multiplying by it. For example, with all-18-decimal assets and a 1.02 oracle price, the function returns `1.02 * amount` instead of `amount / 1.02` — a directional error that grows with price deviation.

Scaling bugs: When `oracleDecimals > tokenDecimals`, the `d1` multiplier inflates the input amount without a compensating downscale, producing results that are off by a power of 10.

Precision loss: The base `ratio = 10^18 / quoteAnswer` performs integer division before multiplication. When `quoteAnswer > 10^18`, this truncates to zero, zeroing the entire result for high-price 18-decimal oracles.

For the affected combinations, share pricing is corrupted: deposits mint wrong share amounts, redemptions pay wrong asset amounts, and NAV computation is incorrect. For a stablecoin pair near 1.0, the error is small, but for any significant price deviation the formula produces wildly wrong results. The divergence from **InstantRedeemSubaccount** formula creates an arbitrage vector between vault share pricing and subaccount redemption pricing.

Assets:

- /src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation:

- Use the same uniform 18-decimal normalization approach already implemented in **InstantRedeemSubaccount** to replace the branching conversion logic in **EnableOnlyAssetsWhitelist**:

```
uint256 normalizedPrice = uint256(quoteAnswer) * (10 ** (18 - oracleDecimals)
);
uint256 normalizedAmount = amount * (10 ** (18 - tokenDecimals));
uint256 normalizedResult = (normalizedAmount * 1e18) / normalizedPrice;
return normalizedResult / (10 ** (18 - REFERENCE_ASSET_DECIMALS));
```

- The existing `6`/`8`/`18` decimal restrictions should be retained.
- Verify that the vault and subaccount produce identical conversion results for the same inputs.

Resolution: Fixed in `5141b8f`:

The branching `_fromInputAssetToReferenceAsset` formula in **EnableOnlyAssetsWhitelist** has been replaced with the uniform normalize-to-18-decimals approach recommended, matching the **InstantRedeemSubaccount** implementation. The new logic normalizes both oracle price and input amount to 18 decimals, performs the division, and scales the result to reference asset decimals.

Evidences

Reproduce:

PoC Steps:

- Deploy `EnableOnlyAssetsWhitelist` with 18-decimal reference asset, 18-decimal input asset, and 18-decimal oracle at price 1.02e18
- Call `fromInputAssetToReferenceAsset(input, 1e18)` — whitelist returns 1.02e18 (multiplied); subaccount formula returns ~0.9804e18 (divided)
- Set oracle price to 2.0e18 — whitelist returns 20e18 for 10e18 input; subaccount correctly returns 5e18 (4x divergence)
- Repeat with 8/8/8 (price 1.05e8) and 8/8/18 (price 1.10e8) — same inversion confirmed via override branches at lines 261-266
- Verify 6/8/18 (USDC/WETH common path) — both formulas agree, confirming only override branches are affected

PoC Code:

```
function test_priceInversion_multipleDecimalCombos() public {
    // --- 18/18/18: override at line 249 multiplies instead of dividing ---
    {
        ConfigurableToken ref18 = new ConfigurableToken("REF18", "REF18", 18)
```

```
;
        ConfigurableToken inp18 = new ConfigurableToken("INP18", "INP18", 18)
;
        ConfigurableOracle oracle18 = new ConfigurableOracle(18, 1.02e18);

        EnableOnlyAssetsWhitelist wl = new EnableOnlyAssetsWhitelist(owner, a
ddress(ref18));
        vm.prank(owner);
        wl.enableAsset(address(inp18), address(oracle18), 1 hours);

        uint256 wlResult = wl.fromInputAssetToReferenceAsset(address(inp18),
1e18);
        uint256 correct = SubaccountFormula.convert(1e18, 18, 18, 18, 1.02e18
);

        emit log_named_uint("[18/18/18] Whitelist  ", wlResult);
        emit log_named_uint("[18/18/18] Subaccount ", correct);

        // Whitelist returns 1.02e18 (multiplied); correct is ~0.9804e18 (div
ided)
        assertGt(wlResult, 1e18, "18/18/18: whitelist exceeds input amount");
        assertLt(correct, 1e18, "18/18/18: subaccount below input amount");
        assertEq(wlResult, 1.02e18, "18/18/18: whitelist = amount * price");

        oracle18.setPrice(2e18);
        uint256 wl2 = wl.fromInputAssetToReferenceAsset(address(inp18), 10e18
);
        uint256 correct2 = SubaccountFormula.convert(10e18, 18, 18, 18, 2e18)
;
        assertEq(wl2, 20e18, "18/18/18 @2x: whitelist = 20e18");
        assertEq(correct2, 5e18, "18/18/18 @2x: subaccount = 5e18");
    }

    // --- 8/8/8: override at line 264 multiplies instead of dividing ---
    {
        ConfigurableToken ref8 = new ConfigurableToken("REF8", "REF8", 8);
        ConfigurableToken inp8 = new ConfigurableToken("INP8", "INP8", 8);
        ConfigurableOracle oracle8 = new ConfigurableOracle(8, 1.05e8);

        EnableOnlyAssetsWhitelist wl = new EnableOnlyAssetsWhitelist(owner, a
ddress(ref8));
        vm.prank(owner);
        wl.enableAsset(address(inp8), address(oracle8), 1 hours);

        uint256 wlResult = wl.fromInputAssetToReferenceAsset(address(inp8), 1
e8);
        uint256 correct = SubaccountFormula.convert(1e8, 8, 8, 8, 1.05e8);

        emit log_named_uint("[8/8/8] Whitelist  ", wlResult);
```

```
        emit log_named_uint("[8/8/8] Subaccount ", correct);

        assertGt(wlResult, 1e8, "8/8/8: whitelist exceeds input");
        assertLt(correct, 1e8, "8/8/8: subaccount below input");
    }

    // --- 8/8/18: override at line 261 multiplies instead of dividing ---
    {
        ConfigurableToken ref8 = new ConfigurableToken("REF8b", "REF8b", 8);
        ConfigurableToken inp18 = new ConfigurableToken("INP18b", "INP18b", 1
8);
        ConfigurableOracle oracle8 = new ConfigurableOracle(8, 1.10e8);

        EnableOnlyAssetsWhitelist wl = new EnableOnlyAssetsWhitelist(owner, a
ddress(ref8));
        vm.prank(owner);
        wl.enableAsset(address(inp18), address(oracle8), 1 hours);

        uint256 wlResult = wl.fromInputAssetToReferenceAsset(address(inp18),
1e18);
        uint256 correct = SubaccountFormula.convert(1e18, 18, 8, 8, 1.10e8);

        emit log_named_uint("[8/8/18] Whitelist  ", wlResult);
        emit log_named_uint("[8/8/18] Subaccount ", correct);

        assertGt(wlResult, correct, "8/8/18: whitelist overvalues input");
    }

    // --- 6/8/18 (USDC/WETH common path): base formula, no override → correc
t ---
    {
        ConfigurableToken usdc = new ConfigurableToken("USDC", "USDC", 6);
        ConfigurableToken weth = new ConfigurableToken("WETH", "WETH", 18);
        // Oracle: 1 USDC = 0.00025 WETH → 25000 (8-dec)
        ConfigurableOracle oracle = new ConfigurableOracle(8, 25000);

        EnableOnlyAssetsWhitelist wl = new EnableOnlyAssetsWhitelist(owner, a
ddress(usdc));
        vm.prank(owner);
        wl.enableAsset(address(weth), address(oracle), 1 hours);

        uint256 wlResult = wl.fromInputAssetToReferenceAsset(address(weth), 1
e18);
        uint256 correct = SubaccountFormula.convert(1e18, 18, 8, 6, 25000);

        emit log_named_uint("[6/8/18] Whitelist  ", wlResult);
        emit log_named_uint("[6/8/18] Subaccount ", correct);

        assertApproxEqAbs(wlResult, 4000e6, 1, "6/8/18: whitelist correct");
```

```
        assertApproxEqAbs(correct, 4000e6, 1, "6/8/18: subaccount correct");
    }
}
```

Results:

```
Ran 1 test for test/PoC_WhitelistPriceInversion.t.sol:PoC_WhitelistPriceInver
sion
[PASS] test_priceInversion_multipleDecimalCombos() (gas: 10377200)
Logs:
  [18/18/18] Whitelist  : 1020000000000000000
  [18/18/18] Subaccount : 980392156862745098
  [8/8/8] Whitelist  : 105000000
  [8/8/8] Subaccount : 95238095
  [8/8/18] Whitelist  : 110000000
  [8/8/18] Subaccount : 90909090
  [6/8/18] Whitelist  : 4000000000
  [6/8/18] Subaccount : 4000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.01ms (713.83µs
CPU time)
```

Files: PoC_WhitelistPriceInversion.t.sol

F-2026-15400 - Operator Can Cancel and Pre-Schedule Timelocked Hashes, Vetoing All Governance Operations - Medium

Description:

The `cancel` and `schedule` functions from the **ResourceBasedTimelockedCall** contracts both accept the operator as a valid caller. This gives the operator an effective veto over all timelocked governance operations. Ownership transfers, management fee changes, and `maxChangePercent` changes with no owner recourse short of `emergencyWithdraw`, since `operatorAddress` is immutable after `configure`.

Both `schedule` and `cancel` validate the caller via `_enforceValidSender`, which permits the resource (vault), owner, or operator (**ResourceBasedTimelockedCall**):

```
function _enforceValidSender() private view {
    address theOwner = RESOURCE.owner();
    address theOperator = RESOURCE.operatorAddress();
    if ((msg.sender != address(RESOURCE)) && (msg.sender != theOwner) && (msg
.sender != theOperator)) {
        revert Unauthorized();
    }
}
```

Cancellation griefing: The operator can cancel any pending hash, including ownership transfer hashes scheduled by the owner. Since `acceptOwnership` calls `consume(h)` atomically (**OperableVault**), cancelling the hash causes the entire `acceptOwnership` transaction to revert - the ownership transfer does not go through.

No operator replacement: `operatorAddress` is set only in `configure` (**TokenizedVault**), which is gated by `ifNotConfigured`. There is no `updateOperator` function anywhere in the codebase. The owner cannot replace a misbehaving operator.

A compromised or adversarial operator can permanently block ownership transfers, management fee updates, and `maxChangePercent` changes. The owner's only recourse is `emergencyWithdraw`, which pauses the entire vault and requires full redeployment. This effectively gives the operator co-equal governance power over timelocked operations despite the role being designed as subordinate to the owner.

Assets:

- /src/core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

- Restrict `cancel` to only `RESOURCE` and the owner, removing the operator. The operator has no legitimate need to cancel hashes, as the owner can always cancel if needed.
- Add an `updateOperator` function, `onlyOwner` and timelocked, so the owner can replace a compromised operator.

Resolution: Fixed in `41a3d34`:

A new **RestrictedTimelockedCall** contract has been introduced, which excludes the operator from `_enforceValidSender`. Only the resource (vault) and its owner may schedule or cancel hashes. The `MasterDeployer` now deploys `RestrictedTimelockedCall` instead of `ResourceBasedTimelockedCall` for atomic vaults. The original `ResourceBasedTimelockedCall` remains unchanged, preserving backward compatibility for non-atomic deployments.

```
// RestrictedTimelockedCall._enforceValidSender()
function _enforceValidSender() private view {
    address theOwner = RESOURCE.owner();
    if ((msg.sender != address(RESOURCE)) && (msg.sender != theOwner)) {
        revert Unauthorized();
    }
}
```

F-2026-15407 - No L2 Sequencer Uptime Check in Oracle Validation, Risking Stale Prices on L2 Deployments - Medium

Description:

EnableOnlyAssetsWhitelist `fromInputAssetToReferenceAsset` function validates oracle freshness through `maxOracleUpdatesDuration` but does not check the L2 sequencer uptime feed. On L2 chains such as Arbitrum and Optimism, if the sequencer goes down and comes back up, Chainlink prices may appear fresh within `maxOracleUpdatesDuration` while still reflecting pre-downtime values.

```
if (quoteAnswer < 1) revert InvalidOraclePrice();
if (quoteAnsweredInRound < quoteRoundId) revert StalePrice();
if (quoteTimestamp < 1) revert RoundNotComplete();
if (quoteTimestamp > block.timestamp) revert InvalidOracleTimestamp();
if (block.timestamp - quoteTimestamp > maxOracleUpdatesDuration[assetAddr]) r
evert InvalidTimePeriod();
```

`npm test` defaults to an Arbitrum fork, and on Optimism via `npm run test-opti`. Chainlink recommends checking the sequencer uptime feed on L2s to avoid using stale prices after sequencer restarts.

After an L2 sequencer restart, the vault may accept deposits or process redemptions at stale oracle prices, enabling arbitrage at the expense of existing depositors.

Assets:

- /src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Integrate the Chainlink L2 sequencer uptime feed check, for example **sequencerUptimeFeed.**`latestRoundData`, with a grace period after restart.

Resolution: Fixed in `5141b8f`:

EnableOnlyAssetsWhitelist now accepts a Chainlink L2 sequencer uptime feed address and grace period as immutable constructor parameters. A `_validateSequencer` internal function is called at the beginning of `_fromInputAssetToReferenceAsset` and reverts with `SequencerDown` or `GracePeriodNotOver` when the sequencer is offline or the grace period after restart has not elapsed. For L1 deployments, passing `address(0)` makes the check a no-op. Constructor-time validation ensures the feed is functional and the grace period is within the `MAX_SEQUENCER_GRACE_PERIOD` (3600 seconds) bound.

F-2026-15436 - Untracked Subaccount Composition Drift Blocks LP Redemptions and Enables Owner Extraction - Medium

Description:

InstantRedeemSubaccount's `redeemAsset` pays out reference asset and receives non-reference tokens, but makes no callback to the vault. The vault's `externalAssets` counter, used as the ceiling in `_ensureLiquidity`, remains unchanged while the subaccount's actual reference-asset balance shrinks. Once the subaccount's reference-asset position is depleted below the amount `_ensureLiquidity` attempts to pull, LP redemptions, fee collection, and performance fee distribution revert. Separately, the vault owner can extract the stranded non-reference tokens from the subaccount via `sweepToken` to an arbitrary address, then gradually reduce `externalAssets` via `updateTotalAssets`, causing LP holders to absorb the NAV reduction.

`redeemAsset` swaps non-reference tokens for reference asset at oracle price minus spread:

```
// InstantRedeemSubaccount.sol
function redeemAsset(address asset, uint256 amount, uint256 minOut) external
... {
    //...
    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    //...
    IERC20(referenceAsset).safeTransfer(msg.sender, referenceOut);
}
```

No vault callback occurs. The subaccount's total value is preserved (it actually increases by the spread fee), so the vault's NAV, which relies on `externalAssets`, remains correct or slightly understated. The problem is not NAV inflation but a liquidity mismatch: `_ensureLiquidity` can only pull reference asset from the subaccount, and the subaccount may no longer have enough:

```
// OraclizedMultiAssetVault.sol
function _ensureLiquidity(uint256 needed) internal {
    //...
    uint256 deficit = needed - vaultBalance;
    if (deficit > externalAssets) revert InvalidExternalAssets();
    IAllocableSubAccount(instantRedemptionSubaccount).withdraw(
        _referenceAsset, deficit, payable(address(this))
    );
    //...
}
```

The `externalAssets` guard passes because it was never decremented, but the subaccount's `withdraw` reverts with `InsufficientLiquidity` when its actual reference-asset balance (including idle vault position) cannot cover the request.

The operator can settle the composition drift via `withdrawFromSubaccount`, which pulls non-reference tokens to the vault and decrements `externalAssets`. However, the vault owner can bypass this path entirely via `sweepToken`:

```
// InstantRedeemSubaccount.sol
function sweepToken(address token, uint256 amount, address to) external nonRe
entrant onlyVaultOwner {
    if (token == address(idleVault)) revert CannotSweepIdleVaultShares();
    if (token == referenceAsset) revert CannotSweepReferenceAsset();
    //...
    IERC20(token).safeTransfer(to, amount);
}
```

After sweeping, the non-reference tokens leave the subaccount entirely, but `externalAssets` still counts their value. The owner can then reduce `externalAssets` via `updateTotalAssets`, bounded by the rate limiter (`maxChangePercent * elapsed / 86400`), causing the NAV to drop at LP holders' expense.

This can lead to:

- **Temporary LP redemption lockup:** Once `redeemAsset` volume depletes the subaccount's reference-asset balance below the level `_ensureLiquidity` requires, LP redemptions, `collectFees`, and `chargePerformanceFees` revert. The vault is not paused during this window. The operator can resolve this by settling via `withdrawFromSubaccount` or pausing subaccount redemptions, but there is no proactive on-chain detection or circuit breaker.
- **Owner value extraction:** The vault owner can extract non-reference tokens from the subaccount to an arbitrary address via `sweepToken`, then gradually reduce `externalAssets` via `updateTotalAssets`. LP holders absorb the resulting NAV reduction. The extraction is rate-limited but not prevented.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 4/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

- Assets that contribute to subaccount NAV should be recoverable only through a vault-controlled, accounting-aware path. Any removal of tracked assets should atomically return the assets to the vault and reconcile `externalAssets`.
- Reserve `sweepToken` for unsupported, accidental, or otherwise non-accounted tokens only. This preserves an emergency recovery mechanism without permitting tracked assets to bypass vault accounting.
- Have `_ensureLiquidity` query the subaccount's `availableLiquidity` before attempting the pull.
- Consider having the subaccount notify the vault of reference-asset outflows so the vault can track pullable liquidity separately from total external value.

Resolution: Accepted:

The finding remains unresolved by design. The Upshift team considers the vault owner a trusted multisig with strictly more powerful capabilities than `sweepToken` (notably `emergencyWithdraw`). Restricting `sweepToken` to vault-only transfers is deemed to reduce operational flexibility without meaningful security gain. Documentation comments have been added to `sweepToken` in **InstantRedeemSubaccount** clarifying this trust assumption.

> **Upshift Team Comment:** Accepted risk. Vault owner is a trusted multisig; sweepToken restriction would reduce operational flexibility without meaningful security gain.

F-2026-15469 - withdrawFromSubaccount Price Divergence Causes Custody Lockout and Phantom NAV - Medium

Description:

`withdrawFromSubaccount` gates every withdrawal against the aggregate `externalAssets` scalar using the asset's **current** oracle price, but `externalAssets` reflects prices at **deposit time**. Under normal market appreciation, the current-price valuation of a single position can exceed the entire stale aggregate, causing the function to revert and locking the operator out of custody. Under depreciation, the withdrawal subtracts less than was originally added, leaving phantom residual value in `externalAssets` that overstates NAV. The only correction path (`updateTotalAssets`) is rate-limited, compounding both problems into multi-day operational lockouts and pricing errors.

When the operator moves a non-reference asset to a subaccount via `depositToSubaccount`, `externalAssets` is incremented by the oracle-converted value at that moment:

```
// OraclizedMultiAssetVault.sol:246-261
uint256 amountInReferenceAssets = (inputAssetAddr == _referenceAsset)
    ? depositAmount
    : _fromInputAssetToReferenceAsset(inputAssetAddr, depositAmount);
// ...
externalAssets += amountInReferenceAssets;
```

When the same asset is later withdrawn via `withdrawFromSubaccount`, the function re-prices the asset at the **current** oracle rate and checks it against the stale aggregate:

```
// OraclizedMultiAssetVault.sol
uint256 amountInReferenceAssets =
    (inputAssetAddr == _referenceAsset) ? amount : _fromInputAssetToReference
Asset(inputAssetAddr, amount);

if (amountInReferenceAssets > externalAssets) revert InvalidExternalAssets();
externalAssets -= amountInReferenceAssets;
```

This creates two failure modes under honest market movement alone:

Appreciation → Withdrawal Blocked

1. Operator deposits 1 WBTC to SubaccountA when BTC = 60,000 USDC. `externalAssets` increases by 60,000.
2. BTC rises to 100,000 USDC.

3. Operator attempts to withdraw the same 1 WBTC. `_fromInputAssetToReferenceAsset` returns 100,000.
4. Guard: `100,000 > 60,000` → reverts `InvalidExternalAssets()`.

The asset is physically present in the subaccount and belongs to the vault, but the operator cannot recover it. Recovery requires first calling `updateTotalAssets` to raise `externalAssets` to at least 100,000, but this is rate-limited by `getMaxAllowedChange`:

```
// OraclizedMultiAssetVault.sol
function getMaxAllowedChange() public view returns (uint256) {
    return (maxChangePercent * (block.timestamp - assetsUpdatedOn)) / uint256
(86400);
}
```

With `maxChangePercent = 200` (2%/day) and 1 hour elapsed, the maximum allowed change is ~0.083%. A 66% appreciation requires ~33 days of accumulated allowance or multiple sequential incremental updates over days.

Depreciation → Phantom NAV

1. Operator deposits 1 WBTC when BTC = 60,000. `externalAssets = 60,000`.
2. BTC falls to 40,000.
3. Operator withdraws 1 WBTC. `_fromInputAssetToReferenceAsset` returns 40,000.
4. `externalAssets -= 40,000` → `externalAssets = 20,000`.

The vault now holds 1 WBTC directly (counted at 40,000 via `balanceOf` in `_getTotalAssetsValuation`), **plus** 20,000 phantom `externalAssets`. NAV is overstated by 20,000 until `updateTotalAssets` corrects it. During this window:

- Redeemers extract more real value per share than warranted.
- New depositors receive fewer shares than deserved.
- Management and performance fees accrue on phantom value.

Multi-Position Amplification

Because `externalAssets` is a single aggregate scalar across all subaccounts and all asset types, the problem compounds. If the vault holds positions in multiple volatile assets across multiple subaccounts, a single appreciated position can consume the entire `externalAssets` budget, blocking withdrawal of **all** positions - even those that haven't moved in price.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Allow the owner to bypass the rate limiter on `updateTotalAssets`. The rate limiter exists to constrain the operator, but the owner already has strictly higher privileges (`emergencyWithdraw`, subaccount control, vault pausing). Bypassing the rate limit for the owner does not expand the trust surface - it only removes operational friction that blocks timely correction:

```
function updateTotalAssets(uint256 externalAssetsAmount) external nonReentran
t ifConfigured onlyOwnerOrOperator {
    if (msg.sender != _owner) {
        uint256 perChange = getChangePercentage(externalAssetsAmount);
        uint256 maxAllowedChangePerc = getMaxAllowedChange();
        if (perChange > maxAllowedChangePerc) revert MaxAllowedChangeReached(
);
    }

    externalAssets = externalAssetsAmount;
    assetsUpdatedOn = block.timestamp;
}
```

This resolves both directions: the owner corrects `externalAssets` upward before withdrawing an appreciated asset, or downward after withdrawing a depreciated one, in a single unrestricted call. The operator remains rate-limited.

Optionally, add an owner-gated `forceWithdrawFromSubaccount` with saturating subtraction to handle the appreciation case atomically (correct + withdraw in one transaction) rather than requiring two separate calls.

Resolution: Fixed in `5141b8f`:

Two changes address the finding. In `updateTotalAssets`, the rate limiter is now bypassed for the owner via `_enforceRateLimit` being called only when `msg.sender != _owner`, allowing unrestricted NAV correction. In `withdrawFromSubaccount`, when `amountInReferenceAssets > externalAssets`, the owner is permitted to proceed with saturating subtraction (`externalAssets =`

`0`) instead of reverting, while the operator still receives `InvalidExternalAssets`. This resolves both the appreciation-lockout and depreciation-phantom-NAV scenarios described in the finding.

```
if (amountInReferenceAssets > externalAssets) {
    if (msg.sender != _owner) revert InvalidExternalAssets();
    externalAssets = 0;
} else {
    externalAssets -= amountInReferenceAssets;
}
```

F-2026-15474 - Immutable operatorAddress Lacks a Mechanism to Replace a Compromised Operator Enabling Persistent NAV Distortion - Medium

Description:

The `operatorAddress` is set during `configure`, and there is no `updateOperator` function anywhere in the codebase. If the operator key is compromised, the owner cannot replace it without deploying a new implementation and upgrading the proxy.

TokenizedVault:

```
operatorAddress = newConfig.operatorAddress;
```

The operator has significant powers: `updateTotalAssets`, `depositToSubaccount`, `withdrawFromSubaccount`, and `schedule` / `cancel` on the timelock. A compromised operator can manipulate NAV, move funds to whitelisted subaccounts or wallets, and block timelocked governance operations.

Because of these permissions, a compromised or malicious operator can materially disrupt vault behavior. In particular, the operator can use `updateTotalAssets` to gradually inflate `externalAssets` within the configured rate limit. Since the allowed change grows linearly with elapsed time and each update resets the timer, the operator can compound incremental increases daily to inflate `externalAssets` far beyond actual subaccount holdings.

OraclizedMultiAssetVault:

```
function getMaxAllowedChange() public view returns (uint256) {
    if (block.timestamp + _TIMESTAMP_MANIPULATION_WINDOW < assetsUpdatedOn) r
evert InvalidTimestamp();
    return (maxChangePercent * (block.timestamp - assetsUpdatedOn)) / uint256
(86400);
}
```

With `maxChangePercent = 500` (5% per day), the operator can compound daily: after 20 days of daily max-increase updates, `externalAssets` grows by a factor of `1.05^20 = ~2.65x`.

Each update resets `assetsUpdatedOn`, so the operator must wait a full day between updates, but the compounding effect is significant over weeks.

The inflated `externalAssets` feeds directly into `_getTotalAssets` → `_getSharePrice` → `chargePerformanceFees`. While actual fee extraction is

bounded by real liquidity (`_ensureLiquidity` reverts if the subaccount cannot deliver), the inflated NAV still causes:

- New depositors to receive fewer shares than deserved
- `chargeManagementFee` to accrue fees on a phantom balance

A misbehaving operator can gradually inflate NAV over weeks, harming new depositors and inflating fee accruals. The owner's only recourse to mitigate the risk is `emergencyWithdraw`, which pauses the entire vault or a full proxy implementation upgrade. The lack of an operator replacement mechanism elevates what would otherwise be a rate-limited risk into a persistent one.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

- Add an `updateOperator(address newOperator)` function with `onlyOwner` access control.
- Consider timelocking this function to prevent the owner from silently swapping the operator.
- Document the operator trust assumption prominently for users and integrators.

Resolution: Fixed in `5141b8f`:

An `updateOperator(address newOperator)` function has been added to **OperableVault**, gated by `onlyOwner` and `ifConfigured`, with timelock consumption via `scheduledCallerAddress`.

F-2026-15389 - Post-Emergency externalAssets Correction Throttled by Rate Limiter - Low

Description:

When `emergencyWithdraw` fails to pull funds from the instant-redemption subaccount, `externalAssets` retains its pre-emergency value. Upon unpausing, the `maxChangePercent` rate limiter on `updateTotalAssets` prevents the owner from rapidly correcting the stale value, delaying NAV realignment.

`externalAssets` is a nominal principal counter reconciled by the owner/operator via `updateTotalAssets`. This is by design - the variable tracks off-vault holdings across potentially multiple chains and subaccounts.

During `emergencyWithdraw`, the vault attempts to pull all subaccount funds. On failure, `externalAssets` is unchanged:

TokenizedVault:

```
try IAllocableSubAccount(instantRedemptionSubaccount).withdraw(
    _referenceAsset, pullAmount, payable(address(this))
) {
    externalAssets = 0;
} catch {
    emit EmergencySubaccountPullFailed(instantRedemptionSubaccount, pullAmoun
t);
    // externalAssets unchanged
}
```

The vault is paused immediately , so no user operations are affected while paused. However, upon unpausing, correction is throttled:

OraclizedMultiAssetVault:

```
function updateTotalAssets(uint256 externalAssetsAmount) external nonReentran
t ifConfigured onlyOwnerOrOperator {
    uint256 perChange = getChangePercentage(externalAssetsAmount);
    uint256 maxAllowedChangePerc = getMaxAllowedChange();
    if (perChange > maxAllowedChangePerc) revert MaxAllowedChangeReached();
    //...
}
```

`getMaxAllowedChange` returns `(maxChangePercent * elapsed) / 86400`. If the stale value is large relative to the real value, full correction may require multiple days of incremental updates.

As a result:

- Between unpausing and full correction, share pricing, management fees, and performance fees operate on stale NAV.
- If `externalAssets` overstates reality, redeemers extract more value than fair; new depositors are diluted.
- If it understates (subaccount accrued yield), new depositors get excess shares at existing LPs' expense.
- Practical impact is bounded: the vault is paused during the emergency itself, and the owner controls when to unpause (presumably after sufficient correction).

Assets:

- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

- Allow `updateTotalAssets` to bypass the rate limiter when `depositsPaused && withdrawalsPaused`, since no user operations can exploit an unconstrained change while the vault is paused. This lets the owner correct NAV before unpausing.
- Alternatively, add a dedicated `emergencyCorrectionExternalAssets` function gated by `onlyOwner` that is only callable while the vault is paused.

Resolution: Fixed in `5141b8f`:

The rate-limit logic has been extracted into `_enforceRateLimit`, which is bypassed when the vault is fully paused (`depositsPaused && withdrawalsPaused`). This allows the owner or operator to correct stale `externalAssets` before unpausing, eliminating the multi-day correction delay described. No user operations (deposits, redemptions, fee collection) can execute while both flags are set.

```
function _enforceRateLimit(uint256 newValue) internal view {
    if (depositsPaused && withdrawalsPaused) return;
    // ... rate limit checks ...
}
```

F-2026-15394 - updateAssetsWhitelist Lacks Timelock, Enabling Instant Replacement of the Pricing Oracle Layer - Low

Description:

`updateAssetsWhitelist` instantly swaps `assetsWhitelistAddress` - the contract that governs all NAV computation (`getTotalAssetsValuation`, `convertToShares`, `fromInputAssetToReferenceAsset`). The only validation is a `REFERENCE_ASSET` match. Unlike `updateMaxChangePercent`, `updateManagementFee`, and `transferOwnership`, this function is not routed through **ResourceBasedTimelockedCall**.

```
// OraclizedMultiAssetVault.sol:146-154
function updateAssetsWhitelist(address newWhitelistAddr) external nonReentran
t ifConfigured onlyOwner {
    if (newWhitelistAddr == address(0)) revert InvalidAddress();
    assetsWhitelistAddress = newWhitelistAddr;
    if (_referenceAsset != IEnableOnlyAssetsWhitelist(newWhitelistAddr).REFER
ENCE_ASSET()) {
        revert ReferenceAssetMismatch();
    }
}
```

A malicious whitelist contract satisfying `REFERENCE_ASSET` can return arbitrary values from `getTotalAssetsValuation` and `convertToShares`. This controls every downstream pricing path: `_getTotalAssets`, `_convertToAssets`, `previewDeposit`, and `_previewRedemption`.

For context, the codebase already timelocks three other sensitive owner operations:

- `transferOwnership` - via `_scheduleOwnershipTransfer` (**OperableVault**)
- `updateMaxChangePercent` - consumes timelock hash (**OraclizedMultiAssetVault**)
- `updateManagementFee` - consumes timelock hash (**OraclizedMultiAssetVault**)

A compromised owner can in a single transaction: deploy a whitelist returning inflated valuations, call `updateAssetsWhitelist`, then redeem shares at the inflated price or deflate valuations and deposit at a discount. Performance fee extraction is also possible since `chargePerformanceFees` relies on `_getTotalAssets`.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Accepted

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Route `updateAssetsWhitelist` through **ResourceBasedTimelockedCall**, consistent with the existing pattern for other sensitive parameters.

Resolution: Accepted:

No timelock has been added to `updateAssetsWhitelist`. The function remains `onlyOwner` with instant execution.

> **Upshift Team Comment:** updateAssetsWhitelist timelock deferred. Owner trust model deemed sufficient for this function.

F-2026-15398 - updatePerformanceFee Has No Upper Bound and No Timelock - Low

Description:

`updatePerformanceFee` allows the owner to set `performanceFeeRate` to any `uint256` value instantly, with no cap and no timelock. A misconfiguration (or a compromised key) could result in an excessive fee extraction when `chargePerformanceFees` is called.

TokenizedVault:

```
function updatePerformanceFee(uint256 newValue) external nonReentrant onlyOwn
er {
    performanceFeeRate = newValue;
}
```

The rate is applied uncapped in `chargePerformanceFees`:

```
uint256 totalAssetsIncrease = currentTotalAssets - highWatermarkNav;
uint256 totalPerformanceFee = (totalAssetsIncrease * performanceFeeRate) / 1e
6;
```

With `performanceFeeRate = 1_000_000`, the entire NAV increase above the watermark is taken as fees. Values above `1_000_000` would extract more than the increase. The fee is transferred directly to `performanceFeeRecipients` (also owner-settable without timelock via `updatePerformanceFeeCollectors`).

An erroneous value set by the owner cannot be caught before damage occurs because `chargePerformanceFees` is callable by anyone - once `watermarkTimeWindow` has elapsed and `currentSharePrice > highWatermark`, any address can trigger the fee extraction.

For context, the codebase already timelocks `updateManagementFee` and `updateMaxChangePercent` via **ResourceBasedTimelockedCall**, but `updatePerformanceFee` lacks equivalent protection.

An incorrect `performanceFeeRate` would cause disproportionate fee extraction from the vault on the next `chargePerformanceFees` call, directly reducing LP share value. The risk is elevated by the absence of both an upper bound and a timelock - there is no safety net against misconfiguration and no window for detection before execution.

Assets:

- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Mitigated

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

- Add an upper bound constant (e.g., `MAX_PERFORMANCE_FEE_RATE = 500_000` for 50%).
- Route `updatePerformanceFee` through `ResourceBasedTimelockedCall`, consistent with `updateManagementFee` and `updateMaxChangePercent`.

Resolution: Mitigated in `5141b8f`:

A `MAX_PERFORMANCE_FEE_RATE` constant of `500_000` (50%) has been added to **TokenizedVault**. The cap is enforced in both `configure` and `updatePerformanceFee`, preventing values above 50% of NAV increase from being set. A `PerformanceFeeUpdated` event has been added. The timelock component of the recommendation was not implemented; `updatePerformanceFee` remains instantly executable by the owner.

> **Upshift Team Comment:** Upper bound cap added (primary ask). Timelock was optional and not implemented.

F-2026-15420 - Several Owner-Only Functions Are Not Protected by Timelocks - Low

Description:

Multiple sensitive owner-only functions execute instantly without timelock protection. While some, such as `pauseDepositsAndWithdrawals`, legitimately need instant execution for emergency response, others such as `updateInstantRedemptionFee`, `updateSendersWhitelist`, `setInstantRedemptionSubaccount`, and `updateFeeCollectors` could benefit from timelocks to give users time to react.

Function	If misconfigured
`updateAssetsWhitelist`	Swaps the entire pricing oracle layer; controls NAV, share pricing, and deposit/withdrawal amounts
`updatePerformanceFee`	No upper bound; can set rate to 100%+ of NAV increase above watermark
`updateInstantRedemptionFee`	Instantly impose up to 10% exit fee, capped by `MAX_INSTANT_REDEMPTION_FEE`
`updateSendersWhitelist`	Lock out users by changing to a restrictive whitelist, or open the vault to all by setting to `address(0)`
`setInstantRedemptionSubaccount`	Redirect liquidity pulls to a different subaccount
`updateFeeCollectors` / `updatePerformanceFeeCollectors`	Redirect all fee payments to new addresses
`updateLimits`	Set `maxWithdrawalAmount` to 1 wei, effectively blocking redemptions
`enableSubAccount`	Whitelist a subaccount that `setInstantRedemptionSubaccount` can then point to

For contrast, `transferOwnership`, `updateMaxChangePercent`, and `updateManagementFee` are already routed through **ResourceBasedTimelockedCall**.

Any of these changes can be executed instantly with no warning period for users to react. The most impactful are `updateAssetsWhitelist` (controls the entire pricing layer - **F-2026-15394**) and `updatePerformanceFee` (uncapped fee extraction on next `chargePerformanceFees` call - **F-2026-15398**).

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

- /src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Mitigated

Classification

Impact Rate: 5/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

- Prioritize timelocking `updateAssetsWhitelist`, `updatePerformanceFee`, `updateFeeCollectors`, `updatePerformanceFeeCollectors`, and `setInstantRedemptionSubaccount`.
- `pauseDepositsAndWithdrawals` and `updateLimits` may legitimately need instant execution for operational flexibility.

Resolution: Mitigated in `5141b8f`:

Two of the recommended functions now consume timelocked hashes: `updateInstantRedemptionFee` in **TimelockedVault** (which also gained the `ifConfigured` modifier) and `updateOperator` in **OperableVault**. The remaining functions listed in the finding remain instantly executable.

> **Upshift Team Comment:** 2 of 8 recommended functions timelocked (updateInstantRedemptionFee, updateOperator). Others remain instant for operational flexibility.

F-2026-15438 - Lack of Cross-System Validation Between Subaccount Redeemable Assets and Vault Whitelist - Low

Description:

The **InstantRedeemSubaccount** `setRedeemableAsset` function does not verify that the enabled asset is whitelisted on the vault's **EnableOnlyAssetsWhitelist**. If a redeemable asset is configured on the subaccount without a corresponding entry in the vault whitelist, the intended settlement path via `withdrawFromSubaccount` reverts at the oracle lookup, leaving the operator unable to recover accumulated non-reference tokens through the standard mechanism.

`setRedeemableAsset` on the subaccount validates oracle configuration, spread bounds, and decimal constraints, but does not check whether the asset is whitelisted on the vault:

```
// InstantRedeemSubaccount.sol
function setRedeemableAsset(
    address asset, bool enabled, uint256 spreadBps,
    address oracle, uint256 maxStaleness, uint256 maxPriceDeviationBps
) external nonReentrant onlyVaultOwner {
    if (enabled) {
        if (asset == referenceAsset) revert AssetMismatch();
        if (oracle == address(0)) revert ZeroAddress();
        if (spreadBps > MAX_SPREAD_BPS) revert SpreadTooHigh(spreadBps, MAX_S
PREAD_BPS);
        // ... no check against vault's EnableOnlyAssetsWhitelist
    }
    // ...
}
```

When the operator later attempts to settle accumulated tokens via `withdrawFromSubaccount`, the function calls `_fromInputAssetToReferenceAsset` on the vault's whitelist contract to compute the oracle-converted value:

```
// OraclizedMultiAssetVault.sol
uint256 amountInReferenceAssets =
    (inputAssetAddr == _referenceAsset) ? amount : _fromInputAssetToReference
Asset(inputAssetAddr, amount);
```

The two configuration surfaces, **EnableOnlyAssetsWhitelist** `enableAsset` on the vault side and `setRedeemableAsset` on the subaccount side, must be kept in sync manually. No on-chain validation enforces this dependency.

If a redeemable asset is enabled on the subaccount without being whitelisted on the vault, users can redeem that asset for reference asset

via `redeemAsset`, but the operator cannot settle the accumulated tokens through `withdrawFromSubaccount`. The only remaining recovery paths are `sweepToken` (which bypasses vault accounting) or `emergencyWithdraw`.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: `setRedeemableAsset` should query the vault's `assetsWhitelistAddress` and verify that the asset being enabled is whitelisted before allowing configuration. This can be achieved by calling `IEnableOnlyAssetsWhitelist(IVaultState(vault).assetsWhitelistAddress()).isWhitelisted(asset)` and reverting if it returns `false`.

Resolution: Fixed in `5141b8f`:

`setRedeemableAsset` in **InstantRedeemSubaccount** now queries the vault's **EnableOnlyAssetsWhitelist** via `IVaultState(vault).assetsWhitelistAddress()` and verifies `isWhitelisted(asset)` before enabling an asset for redemption. A custom `AssetNotOnVaultWhitelist` error has been added for the revert path. This is a point-in-time check as documented in code comments; removing an asset from the vault whitelist does not auto-disable it on the subaccount.

```
address whitelist = IVaultState(vault).assetsWhitelistAddress();
if (!IEnableOnlyAssetsWhitelist(whitelist).isWhitelisted(asset)) revert Asset
NotOnVaultWhitelist();
```

F-2026-15416 - Multiple Admin Functions Emit No Events, Hindering Off-Chain Monitoring - Info

Description:

Eleven owner/operator-restricted functions across the in-scope contracts modify security-critical state without emitting events. This prevents off-chain monitoring systems from detecting configuration changes and reduces auditability.

The following admin functions emit no events:

SendersWhitelist.sol:

```
function enableSender(address addr) external override onlyOwner { ... }
function disableSender(address addr) external override onlyOwner { ... }
```

EnableOnlyAssetsWhitelist.sol:

```
function enableAsset(address assetAddr, address oracleAddr, uint256 newOracle
Duration) external onlyOwner { ... }
function updateOracleLagDuration(uint256 newMaxOracleUpdatesDuration, address
assetAddr) external onlyOwner { ... }
```

OraclizedMultiAssetVault.sol:

```
function updateAssetsWhitelist(address newWhitelistAddr) external onlyOwner {
... }
function updateTotalAssets(uint256 externalAssetsAmount) external onlyOwnerOr
Operator { ... }
```

TimelockedVault.sol:

```
function updateInstantRedemptionFee(uint256 newValue, bool pKeepFeeInVault) e
xternal onlyOwner { ... }
function updateLimits(uint256 newMaxDepositAmount, uint256 newMaxWithdrawalAm
ount, uint256 newDepositCap) external onlyOwner { ... }
```

TokenizedVault.sol:

```
function updatePerformanceFee(uint256 newValue) external onlyOwner { ... }
```

OperableVault.sol:

```
function updateFeeCollectors(CollectorDefinition[] calldata collectors) exter
nal onlyOwner { ... }
```

```
function updatePerformanceFeeCollectors(CollectorDefinition[] calldata collectors) external onlyOwner { ... }
```

For contrast, other admin functions in the same codebase do emit events correctly: `pauseDepositsAndWithdrawals`, `updateSendersWhitelist`, `enableSubAccount`, `disableSubAccount`, `updateMaxChangePercent`, `updateManagementFee`, `setInstantRedemptionSubaccount`.

Off-chain monitoring tools cannot detect changes to fee rates, fee recipients, deposit/withdrawal limits, oracle configurations, whitelist membership, or the pricing layer contract. This reduces transparency for users and governance watchers and eliminates the on-chain audit trail for these mutations.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Add and emit descriptive events in each of the listed functions. At minimum, each event should index the caller and include the new value(s).

Resolution: Fixed in `5141b8f`:

Events have been added to all atomic vault admin functions identified in the finding. The `enableSender` and `disableSender` functions in `SendersWhitelist.sol` remain without events.

> **Upshift Team Comment:** All atomic vault functions emit events. Legacy SendersWhitelist.sol is outside atomic scope.

F-2026-15421 - _owner Is Assigned Twice, in initialize and configure, Potentially with Different Values - Info

Description: The `initialize` function sets `_owner = ownerAddr`, and `configure` later sets `_owner = newConfig.futureOwnerAddress`. If `futureOwnerAddress` differs from the original `ownerAddr`, ownership silently changes during configuration without emitting an `OwnershipTransferred` event.

TokenizedVault:

```
function initialize(address ownerAddr) external virtual initializer {
    depositsPaused = true;
    withdrawalsPaused = true;
    _owner = ownerAddr;
}
```

```
// TokenizedVault.sol
_owner = newConfig.futureOwnerAddress;
```

The silent ownership change during configuration could confuse off-chain monitoring.

Assets:

- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

- Emit `OwnershipTransferred(ownerAddr, newConfig.futureOwnerAddress)` in `configure` if the addresses differ.
- Alternatively, remove the `_owner` assignment from `configure` and require the initial owner to use `transferOwnership` after configuration.

Resolution:

Fixed in `5141b8f`:

`initialize` now calls `_transferOwnership(ownerAddr)` instead of directly assigning `_owner`, which emits the `OwnershipTransferred` event. In `configure`, the ownership reassignment to `newConfig.futureOwnerAddress` now explicitly emits `OwnershipTransferred(previousOwner, newConfig.futureOwnerAddress)`, ensuring both assignments produce an auditable on-chain event trail.

F-2026-15427 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.xx`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.20`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/BaseReentrancy.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	1/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: It is recommended to **lock the pragma version** to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.20;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: Accepted:

Floating pragma (`^0.8.20`) has been retained across all in-scope contracts.

> **Upshift Team Comment:** Floating pragma retained per team review. Compiler version is pinned in foundry.toml; fixed pragma limits future compilation flexibility.

F-2026-15441 - Inconsistent Storage-Gap Strategy Increases Upgrade Collision Risk - Info

Description:

The atomic vault system applies an inconsistent storage-gap strategy across its upgradeable inheritance chain. Some intermediate contracts reserve small gaps, the shared upgradeable ownership base reserves none, and the final leaf implementation still retains a gap. No immediate storage issue is introduced because these contracts are intended for fresh deployment, but the layout increases upgrade-safety risk if future maintenance upgrades are ever performed.

Within that chain, several contracts reserve no gaps:

```
// OwnableGuarded
uint256 private _status;
address internal _owner;
```

GuardedProxyOwnable2Steps then appends additional state and likewise reserves no gap:

```
// GuardedProxyOwnable2Steps
address internal _pendingOwner;
bool internal _configured;
```

Above those shared bases, the vault-specific contracts reserve small and inconsistent storage gaps:

```
uint256[10] private __gap; // OperableVault
uint256[8] private __gap; // OraclizedMultiAssetVault
uint256[9] private __gap; // TimelockedVault
uint256[10] private __gap; // TokenizedVault
```

This produces different storage-extension rules within the same upgradeable hierarchy:

- **OwnableGuarded** and **GuardedProxyOwnable2Steps** have no reserved storage capacity.
- Intermediate vault contracts reserve only limited and non-uniform capacity.
- The final concrete **TokenizedVault** still contains a gap even though no child implementation currently inherits from it.

Because the atomic vaults are intended for fresh deployment, these gap sizes are not constrained by the need to preserve a prior V2 proxy layout.

If future state variables were ever introduced into **OwnableGuarded** or **GuardedProxyOwnable2Steps**, storage slots in all derived vault contracts would shift. If future changes exceeded the small reserved capacity in **OperableVault**, **OraclizedMultiAssetVault**, or **TimelockedVault**, the same class of storage-layout hazard would arise there as well.

Assets:

- /src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /src/core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: A consistent storage-layout policy should be adopted across the full upgradeable inheritance chain.

- Shared base and intermediate upgradeable contracts such as **OwnableGuarded**, **GuardedProxyOwnable2Steps**, **OperableVault**, **OraclizedMultiAssetVault**, and **TimelockedVault** should use a standard reserved storage gap, such as `50`.
- Final leaf implementations such as **TokenizedVault** should not reserve storage gaps unless further inheritance is intentionally planned.
- Contracts that are intended to remain frozen should be documented clearly so that future upgrades are not attempted under incorrect storage assumptions.

Resolution: Fixed in `5141b8f`:

Storage gaps in the atomic-vault-specific contracts have been standardized to `uint256[50]`. **TokenizedVault** has had its `uint256[10]` gap removed entirely, consistent with its role as a leaf implementation. **GuardedProxyOwnable2Steps** and **OwnableGuarded** remain without gaps.

> **Upshift Team Comment:** Atomic-specific contracts standardized. Shared core contracts unchanged to preserve non-atomic vault upgrade safety.

F-2026-15450 - Deposits in Non-Reference Assets May Not Be Immediately Redeemable via the Instant Redemption Flow - Info

Description:

The **OraclizedMultiAssetVault** is designed as a multi-asset vault that accepts deposits in whitelisted assets while using a single reference asset as the primary accounting and redemption unit. Users may deposit any whitelisted asset, including the reference asset, through `deposit()`. In addition, both reference and non-reference assets may be routed to subaccounts via `depositToSubaccount()`.

```
function depositToSubaccount(address inputAssetAddr, uint256 depositAmount, a
ddress subAccountAddr)
    external
    nonReentrant
    ifConfigured
    onlyOwnerOrOperator
{
    ...

    if (accountType == ACCOUNT_TYPE_SUBACCOUNT) {
        // Deposit funds in the sub account
        SafeERC20Upgradeable.forceApprove(IERC20Upgradeable(inputAssetAddr),
subAccountAddr, depositAmount);
        IAllocableSubAccount(subAccountAddr).deposit(inputAssetAddr, depositA
mount);
        SafeERC20Upgradeable.forceApprove(IERC20Upgradeable(inputAssetAddr),
subAccountAddr, 0);
    } else {
        // Transfer the funds to a whitelisted wallet or EOA
        SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(inputAssetAddr),
subAccountAddr, depositAmount);
    }
}
```

When assets are deposited into an **InstantRedeemSubaccount**, only the reference asset is further deposited into the idle ERC4626 vault to earn yield. Non-reference assets may instead be held in other subaccounts or EOAs for external strategy execution. Both asset types can later be returned to the vault through `withdrawFromSubaccount()` by the operator or owner.

However, the standard instant redemption flow in **TimelockedVault** `instantRedeem()` only redeems in the reference asset.

```
function instantRedeem(uint256 shares, address receiverAddr)
    external
    nonReentrant
    ifConfigured
    ifWithdrawalsNotPaused
    ifSenderWhitelisted
{
    // Checks
    if ((receiverAddr == address(0)) || (receiverAddr == address(this))) reve
rt InvalidReceiver();
    if (shares < 1) revert InvalidAmount();

    _executeRedemption(shares, receiverAddr);
}
```

If the vault does not have enough reference-asset liquidity on hand, it attempts to source the deficit through `_ensureLiquidity()` from the **InstantRedeemSubaccount**. In practice, this mechanism only pulls the **reference asset**. Non-reference assets are not automatically converted or redeemed as part of the instant redemption flow. To make such liquidity available, those assets must first be externally converted into the reference asset and then returned to the vault through an explicit operator or owner action.

As a result, a portion of the vault's total value may be reflected in NAV while not being immediately available for instant redemption.

For example, assume the reference asset is **USDC**, and the whitelisted assets are **WETH, WBTC,** and **USDT**.

- User 1 deposits **1,000 USDC**
- User 2 deposits **1 WETH = 3,000 USDC**
- User 3 deposits **1,000 USDT = 1,000 USDC**

The vault's total asset value becomes **5,000 USDC** based on oracle valuation. However, only **1,000 USDC** is actually available as redeemable reference-asset liquidity. The deposited **WETH** and **USDT** remain held as non-reference assets and are not automatically converted into **USDC**. As a result, the vault may appear fully collateralized by NAV, while only a small portion of that value is actually available for instant redemptions at the current moment.

Accordingly, users depositing non-reference assets should not assume that their full position value will always be instantly redeemable in the reference asset at any given time. If available reference-asset liquidity is insufficient, instant redemption may depend on prior operator rebalancing, external asset conversion, or other liquidity management actions.

Assets:

- /src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: The current documentation already states that `instantRedeem` transfers reference tokens only. However, it is recommended to expand this documentation to more explicitly clarify the associated liquidity implication: although non-reference assets contribute to vault NAV, deposits made in non-reference assets may not be instantly redeemable at a given moment unless sufficient reference-asset liquidity is available. If vault value is held in non-reference assets, instant redemption may require prior operator rebalancing, external conversion of those assets into the reference asset, and/or returning liquidity from subaccounts before redemption can be fulfilled. This would help align user expectations around instant redemption availability and liquidity behavior.

Resolution: Fixed in `5141b8f`:

NatSpec documentation has been added to `deposit` in **OraclizedMultiAssetVault** and `instantRedeem` in **TimelockedVault**, explicitly clarifying that non-reference asset deposits contribute to vault NAV but may not be instantly redeemable without prior operator rebalancing. The `@notice` tags now state that instant redemption is serviced exclusively in the reference asset and that redemption availability depends on operator rebalancing or external conversion when vault value is held in non-reference assets.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/fractal-protocol/august-contracts-v2/
Commit	04453121294fc9b164f4c4291aa0d8759d2b0714
Final Commit	f282f65e5af9fc841e85def65f338ccf6d9dd8a3
Whitepaper	N/A
Requirements	https://docs.upshift.finance/architecture/vault-architecture; NatSpec
Technical Requirements	README.md

Asset	Type
/src/core/BaseReentrancy.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/tokenized-vaults-atomic/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/tokenized-vaults-atomic/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/tokenized-vaults-atomic/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/src/tokenized-vaults-atomic/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.