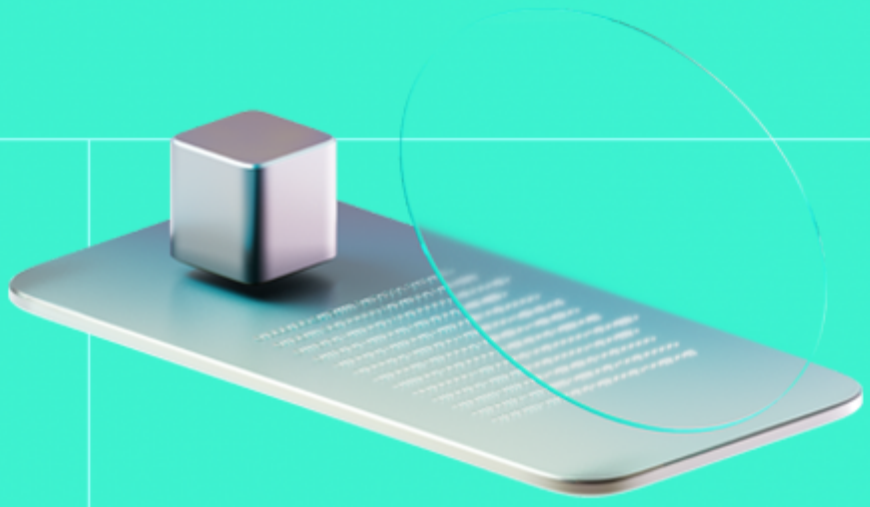
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Upshift Finance

Date: 25/03/2026

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 **InstantRedeemSubaccount** is an instant-redemption module that allows whitelisted users to swap supported assets into a reference asset at oracle-based pricing minus spread.

Document

Name	Smart Contract Code Review and Security Analysis Report for Upshift Finance
Audited By	Ivan Bondar
Approved By	Khrystyna Tkachuk
Website	https://www.upshift.finance/
Changelog	12/03/2026 - Preliminary Report
	25.03.2026 - Final Report
Platform	Ethereum
Language	Solidity
Tags	ERC4626; Yield Farming; Centralization; Claims; Vault
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/fractal-protocol/august-contracts-v2/
Commit	0445312
Final Commit	b544e1e

Audit Summary

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

7	7	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	1
Low	2

Vulnerability	Severity	Status
F-2026-15346 - Oracle Validation Omits L2 Sequencer-Uptime Protection	Medium	Fixed
F-2026-15329 - Unbounded maxPriceDeviationBps Causes Arithmetic Underflow and Bricks Asset Redemption	Low	Fixed
F-2026-15364 - Missing Morpho V2 forceDeallocate Recovery Path May Delay Reference Asset Recovery	Low	Fixed
F-2026-15327 - Missing Zero-Address Validation in Constructor Allows Permanent Loss of Administrative Control	Info	Fixed
F-2026-15328 - Missing Events in Sender Whitelist Management Functions Reduces Administrative Transparency	Info	Fixed
F-2026-15343 - Morpho V2 Exit Liveness Depends on External Conditions	Info	Fixed
F-2026-15345 - setRedemptionsPaused Duplicates Role Checks Instead of Reusing Existing Access-Control Modifiers	Info	Fixed

Documentation quality

- Functional requirements are detailed.
  - Project overview is provided.
  - Most roles in the system are described.
  - Use cases are described.
  - For each contract, the core features are described.
  - Core interactions are described.
- Technical description is limited.
  - Run instructions are provided.
  - Technical specification is partially provided.
  - The NatSpec documentation is generally sufficient.

Code quality

- The development environment is configured.

Test coverage

Code coverage of the project is **79.12%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage are partially missed.
- Interactions by several users are tested.
- Not all branches are covered with tests.

Table of Contents

System Overview

This system is an upgradeable multi-asset vault architecture featuring a dedicated instant redemption module layered on top of an external ERC-4626 idle strategy. Users deposit into an out-of-scope **TokenizedVault** / **OraclizedMultiAssetVault** hierarchy, which prices assets via whitelist-managed Chainlink-style oracles and tracks off-vault allocations through an `externalAssets` counter. The scoped **InstantRedeemSubaccount** is one such allocation target: it accepts assets from the parent vault, allows whitelisted users to redeem selected assets instantly into the vault's reference asset at oracle NAV minus spread, and deploys idle reference-asset liquidity into an external ERC-4626 idle vault.

In the intended deployment, that idle vault is a Morpho Vault V2 instance. This creates a second layer of trust and liveness dependency beyond the local code: the subaccount depends on Morpho's ERC-4626 accounting, liquidity adapter routing, gate configuration, fee mechanics, and allocator behavior to make reference-asset liquidity available when redemptions or vault-side pulls occur. The parent vault also remains a critical trust anchor because the scoped contracts do not own their own governance model: **InstantRedeemSubaccount** reads owner, operator, sender whitelist, and asset-whitelist references from the parent vault on every call rather than storing those permissions locally.

Architecturally, the scoped contracts are therefore not standalone products. **InstantRedeemSubaccount** is an execution module controlled by the parent vault; **SendersWhitelist** is an optional access-control helper used by the parent vault and mirrored by the subaccount; and **OwnableGuarded** is the minimal local ownership primitive used by the whitelist and other core contracts. Correctness and security depend on three external systems behaving as expected:

- The upgradeable vault hierarchy for valuation, accounting, and delegated permissions.
- Chainlink-compatible oracle feeds chosen by governance.
- Morpho Vault V2 as the concrete idle ERC-4626 strategy for reference-asset liquidity.

Files in Scope

- **InstantRedeemSubaccount.sol:** Standalone type-1 subaccount that accepts vault deposits, enables whitelisted instant redemption of configured assets into the reference asset, and parks idle reference liquidity in an ERC-4626 idle vault.
- **SendersWhitelist.sol:** Simple owner-managed address whitelist used to authorize senders for sensitive flows such as deposits/redemptions when the parent vault points to it.
- **OwnableGuarded.sol:** Lightweight ownable base contract with a local `onlyOwner` modifier and internal non-reentrancy guard, inherited by **SendersWhitelist** and related core contracts.

Privileged roles

InstantRedeemSubaccount.sol: Access control is delegated. The contract does not store an owner or operator locally; instead it reads them from the parent vault through `IVaultState(vault)` on every call.

- `onlyVault` Source of authority: `msg.sender` must equal the immutable parent `vault`. Permitted actions:
  - The parent vault has unrestricted ability to move any token into or out of the subaccount through this interface (`deposit` / `withdraw`), subject only to local receiver checks on withdrawals.
- `onlyVaultOwner` Source of authority: `IVaultState(vault).owner()`. Permitted actions:

    - The parent vault owner controls redemption configuration, supported assets/oracles/spreads, transaction limits, and emergency token recovery (`setMaxRedemptionPerTx` / `setRedeemableAsset` / `setMaxRedemptionPerTx` / `setRedemptionsPaused` / `sweepToken`).
- `onlyVaultOwnerOrOperator` Source of authority: `IVaultState(vault).owner()` or `IVaultState(vault).operatorAddress()`.
    - The parent vault owner or operator manages day-to-day idle-vault liquidity and emergency withdrawal flows through `depositToIdle`, `withdrawFromIdle`, `emergencyWithdrawAll`, and the pause branch of `setRedemptionsPaused`.
- `ifSenderWhitelisted` Source of authority: parent vault owner and operator bypass this check; all other callers are checked against the contract returned by `IVaultState(vault).sendersWhitelistAddress()`. If no whitelist is configured, the check is skipped.
    - End users may call `redeemAsset` only if they are authorized by the currently configured external sender-whitelist.

SendersWhitelist.sol

- `onlyOwner` Source of authority: local `_owner` set in the constructor and changeable via `transferOwnership`. Permitted actions:
    - `transferOwnership(address newOwner)` through inherited `OwnableGuarded`. Trust implication:
        - The whitelist owner fully controls (`enableSender` / `disableSender`) which addresses pass whitelist-gated flows in any vault/subaccount that points to this contract.

OwnableGuarded.sol

`onlyOwner` Source of authority: local `_owner`. Permitted action in this base:

- `transferOwnership(address newOwner)`. Permitted action in inheritors:
    - Any inheriting function that applies `onlyOwner`, such as whitelist mutations in `SendersWhitelist`.

Potential Risks

Out-of-Scope Components & 3rd Party Dependencies

- **Scope Definition and Security Guarantees:** The audit does not cover all code in the repository. The scope is limited to **InstantRedeemSubaccount.sol**, **SendersWhitelist.sol**, and **OwnableGuarded.sol**. Contracts outside the audit scope may introduce vulnerabilities, potentially impacting the overall security due to the interconnected nature of smart contracts.
- **Dependency on External Logic for Implemented Redemption and Accounting Flows:** The implemented instant-redemption and idle-vault accounting logic highly depends on external contracts not covered by the audit — specifically **EnableOnlyAssetsWhitelist**, **OraclizedMultiAssetVault**, **TimelockedVault**, and **TokenizedVault**. This reliance introduces risks if these external contracts are compromised or contain vulnerabilities, affecting the audited project's integrity.
- **System Reliance on External Contracts:** The functioning of the system significantly relies on specific external contracts: the parent vault, configured Chainlink-style oracle contracts, ERC-20 token implementations, and the ERC-4626 idle vault. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
- **Interactions with External DeFi Protocols:** The subaccount deploys reference assets into an external ERC-4626 idle vault. Dependence on this external DeFi protocol inherits its risks and vulnerabilities. This might lead to direct financial losses if the protocol is exploited, indirectly affecting the audited project.
- **Morpho Vault V2–Specific Dependency Risks:** The intended idle vault is a Morpho Vault V2 instance, which introduces protocol-specific trust dependencies beyond generic ERC-4626 assumptions:
  - Gate changes can block deposits or withdrawals.
  - All ERC-4626 `max*` functions return `0`.
  - Standard exits only use idle assets plus the configured `liquidityAdapter`.
  - Deposit flows can revert because of gates, caps, or adapter configuration.
  - `maxRate` separates ERC-4626 claim value from raw underlying assets.
  - Performance and management fees dilute the subaccount's share position over time.

Permissions, Authorization & Access

- **Coarse-grained Authorization Model Risks:** The broad authorization model increases the risk of protocol control loss if any authorized address is compromised. Critical operations depend on privileged actors — primarily the vault owner and, in some cases, the vault operator. Compromise or misuse of those permissions can pause redemptions, alter redeemable asset configuration, redirect recovered tokens, or change whitelist controls, potentially leading to unauthorized actions and significant financial loss.
- **Absence of Time-lock Mechanisms for Critical Operations:** Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions. The scoped contracts do not enforce timelocks for critical actions such as redeemable-asset configuration, pausing, or token sweeping, increasing the risk of rapid exploitation and irreversible changes unless constrained by off-chain governance or the parent system.

Centralization

- **Administrative Key Control Risks:** The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds. The architecture relies on privileged keys controlling redemption configuration, pausing, token recovery, and whitelist administration.
- **Single Points of Failure and Control:** The project is partially centralized, introducing single points of failure and control. The owner-controlled and operator-assisted control model means the security and availability of redemption functionality depend on those roles being managed safely and consistently, making the system more susceptible to targeted attacks or manipulation.
- **Centralized Oracles as Data Sources:** The system relies on manually selected Chainlink-style oracle feeds. Dependence on a singular or limited set of data sources can introduce accuracy and manipulation risks, potentially affecting the system's operations and decision-making processes. Incorrect feed selection, stale rounds, or L2 sequencer issues can lead to mispricing and unfair redemptions.

Findings

Vulnerability Details

F-2026-15346 - Oracle Validation Omits L2 Sequencer-Uptime Protection - Medium

Description:

The contract validates Chainlink price rounds for freshness and consistency, but it does not integrate an L2 sequencer uptime feed. On rollups such as Arbitrum or Optimism, this leaves a post-outage window where a price may satisfy local staleness checks while still being unsafe to use.

```
// Query the external Oracle
(uint80 quoteRoundId, int256 quoteAnswer,, uint256 quoteTimestamp, uint80 quoteAnsweredInRound) =
    IAggregatorV3Interface(oracleAddr).latestRoundData();

// Validate the Oracle's response
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 > info.maxOracleStaleness) revert InvalidTimePeriod();
```

Chainlink recommends implementing a sequencer uptime feed check on L2 networks, as price feeds may appear up to date immediately after a sequencer outage while still reflecting pre-downtime prices.

During the recovery window after a sequencer outage, a whitelisted user can redeem at a stale (pre-downtime) price. If the market moved significantly during the outage, the redeemer extracts more reference assets than fair value.

Assets:

- /src/subaccounts/InstantRedeemSubaccount.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: If the contract is intended for L1 (mainnet) only, this restriction should be explicitly documented.

Alternatively, consider making the sequencer uptime feed an optional constructor parameter to enable L2-specific checks when required. For example:

```
function _validateSequencer() internal view {
    if (address(sequencerUptimeFeed) == address(0)) return; // mainnet / non-
L2 deployment

    (, int256 answer, uint256 startedAt,,) =
        sequencerUptimeFeed.latestRoundData();

    if (startedAt == 0) revert InvalidSequencerFeed();
    if (answer != 0) revert SequencerDown();

    if (block.timestamp <= startedAt || block.timestamp - startedAt < sequenc
erGracePeriod) {
        revert GracePeriodNotOver(startedAt, startedAt + sequencerGracePeriod
);
    }
}
```

Resolution: Fixed in `b544e1e`

The remediation introduces an optional Chainlink L2 sequencer uptime feed as immutable constructor parameters (`SEQUENCER_UPTIME_FEED`, `SEQUENCER_GRACE_PERIOD`). A new internal function `_validateSequencer` is invoked at the top of `_fromInputAssetToReferenceAsset()`, which gates all oracle-dependent paths (`redeemAsset`, `previewRedemption`, `setRedeemableAsset`). The implementation validates round completeness (`startedAt`, `updatedAt`, `answeredInRound`), confirms the sequencer is online (`answer == 0`), and enforces the grace period before trusting oracle data. On L1 deployments the check is a no-op (`SEQUENCER_UPTIME_FEED == address(0)`). The constructor

validates the feed at deployment time, including a liveness probe via `latestRoundData()`.

F-2026-15329 - Unbounded maxPriceDeviationBps Causes Arithmetic Underflow and Bricks Asset Redemption - Low

Description:

The `setRedeemableAsset()` function allows `maxPriceDeviationBps` to be set without enforcing an upper bound. During redemption logic, `_checkPriceDeviation()` computes `10000 - maxPriceDeviationBps`, which causes an arithmetic underflow if the value exceeds `10000`. Under Solidity 0.8.x checked arithmetic, this results in a revert, preventing all redemption-related operations for the affected asset.

The `setRedeemableAsset()` function stores configuration parameters for redeemable assets, including the allowed oracle price deviation expressed in basis points:

```
function setRedeemableAsset(address asset, bool enabled, uint256 spreadBps, a
ddress oracle, uint256 maxStaleness, uint256 maxPriceDeviationBps) external .
.. {
    //...
    redeemableAssets[asset] = RedeemableAssetInfo({
        //...
        maxPriceDeviationBps: maxPriceDeviationBps, // <-- no upper bound che
ck
        //...
    });
    //...
    _fromInputAssetToReferenceAsset(asset, 1); // <-- does NOT call _checkPri
ceDeviation
}
```

The value `maxPriceDeviationBps` is stored without any validation ensuring that it remains within a valid range.

Later, the `_checkPriceDeviation()` function calculates the acceptable lower price bound using the following expression:

```
function _checkPriceDeviation(...) internal view {
    //...
    uint256 lowerBound = normalizedAmount * (10000 - info.maxPriceDeviationBp
s) / 10000; // <-- underflow if > 10000
    //...
}
```

The oracle sanity check passes because it calls `_fromInputAssetToReferenceAsset()` which does not invoke `_checkPriceDeviation()`.

The result is denial of service for that asset. Every `previewRedemption()` and `redeemAsset()` call for normal-size amounts reverts in `_checkPriceDeviation()` due to underflow. The asset is effectively bricked until governance reconfigures it.

Assets:

- /src/subaccounts/InstantRedeemSubaccount.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: Reject `maxPriceDeviationBps > 10000` in `setRedeemableAsset`:

```
if (maxPriceDeviationBps > 10000) revert InvalidPriceDeviationBps();
```

If the intention is to disable the check entirely, keep `0` as the only opt-out value.

Resolution: Fixed in `b544e1e`

The remediation adds an explicit upper-bound check in `setRedeemableAsset()`:

```
if (maxPriceDeviationBps > BPS_DENOMINATOR) revert InvalidPriceDeviationBps()
;
```

This prevents configuration of values exceeding 10,000 bps, eliminating the arithmetic underflow in `_checkPriceDeviation()` when computing `BPS_DENOMINATOR - maxPriceDeviationBps`.

F-2026-15364 - Missing Morpho V2 forceDeallocate Recovery Path May Delay Reference Asset Recovery - Low

Description:

InstantRedeemSubaccount supports standard ERC-4626 withdrawal paths through `withdraw()` and `redeem()`, and the owner/operator can retry manually through `withdrawFromIdle()` with smaller amounts as liquidity becomes available. However, when the idle vault is Morpho V2, standard exits can only use idle assets plus the configured `liquidityAdapter`. Liquidity parked in other adapters remains inaccessible to this contract because it exposes no owner/operator wrapper for Morpho's `forceDeallocate()`. In stressed market conditions, this can prevent or delay recovery of at least part of the reference assets that could otherwise be made available, even if doing so requires accepting a penalty.

The subaccount's internal withdrawal helper only attempts standard ERC-4626 exit paths:

```
function _withdrawFromIdleVault(uint256 amount) internal returns (uint256 act
ual) {
    uint256 shares = idleVault.balanceOf(address(this));
    if (shares == 0) return 0;

    uint256 estimated = idleVault.convertToAssets(shares);
    if (estimated == 0) return 0;
    uint256 toWithdraw = amount < estimated ? amount : estimated;

    uint256 balBefore = IERC20(referenceAsset).balanceOf(address(this));
    try idleVault.withdraw(toWithdraw, address(this), address(this)) {}
    catch (bytes memory reason) {
        // withdraw() may revert on vaults where maxWithdraw returns 0 (e.g.
Morpho V2).
        // Fall back to redeeming the equivalent shares. If redeem also fails
, let it revert.
        emit IdleVaultFallback(toWithdraw, reason);
        uint256 sharesNeeded = idleVault.previewWithdraw(toWithdraw);
        if (sharesNeeded == 0) {
            // previewWithdraw unreliable — estimate proportionally
            sharesNeeded = (toWithdraw * shares) / estimated;
            if (sharesNeeded == 0) sharesNeeded = 1; // absolute dust floor
        }
        if (sharesNeeded > shares) sharesNeeded = shares;
        idleVault.redeem(sharesNeeded, address(this), address(this));
    }
```

```
        actual = IERC20(referenceAsset).balanceOf(address(this)) - balBefore;
    }
```

With Morpho V2, standard `withdraw` / `redeem` only consume:

- idle assets held directly by the Morpho vault, and
- liquidity deallocated from the configured `liquidityAdapter`.

They do **not** free liquidity held in other adapters. Morpho exposes `forceDeallocate()` specifically for that recovery path, but the subaccount does not integrate it. As a result, the owner/operator can retry standard withdrawals indefinitely, but cannot actively force partial recovery from non-liquidity-adapter positions even when that would be preferable to waiting.

As a result:

- In stressed or fragmented liquidity conditions, some reference assets may remain unrecoverable through the contract's available admin flows even though Morpho's `forceDeallocate` could potentially free part of them.
- Emergency recovery may be slower or less complete than it could be.
- The protocol may be forced to rely on pausing redemptions and waiting for external Morpho-side liquidity conditions to improve instead of proactively recovering part of the position.

Assets:

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

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Add an **owner-only** Morpho-specific emergency recovery function that exposes `forceDeallocate` as a strict last-resort tool. Unlike `withdrawFromIdle`,

this path can impose a penalty on the subaccount's shares, so it should not be available to the operator for routine liquidity management.

Example:

```
function forceDeallocateFromIdle(
    address adapter,
    bytes calldata data,
    uint256 referenceAssetAmount
) external nonReentrant onlyVaultOwner {
    IMorphoVaultV2(address(idleVault)).forceDeallocate(
        adapter,
        data,
        referenceAssetAmount,
        address(this)
    );
}
```

This path should be documented as an emergency-only mechanism to recover liquidity from non-`liquidityAdapter` positions when standard `withdraw` / `redeem` flows cannot do so. Because `forceDeallocate` applies a penalty, it should be used only when accepting a controlled loss is preferable to leaving assets inaccessible.

Resolution: Fixed in `b544e1e`

The remediation adds `forceDeallocateFromIdle()`, an owner-only (`onlyVaultOwner`) function protected by `nonReentrant`, which wraps **IMorphoVaultV2** `forceDeallocate()` function. The function validates inputs (`adapter != address(0)`, `assets != 0`), delegates to the idle vault cast as `IMorphoVaultV2`, and emits `ForceDeallocateExecuted` with the penalty shares.

F-2026-15327 - Missing Zero-Address Validation in Constructor Allows Permanent Loss of Administrative Control - Info

Description:

The constructor of the **SendersWhitelist** contract assigns the owner without validating that the provided address is non-zero. If the zero address is supplied during deployment, administrative control becomes permanently inaccessible, preventing any future management of the whitelist.

The **SendersWhitelist** contract inherits from **OwnableGuarded** and sets the contract owner directly in the constructor:

```
constructor(address ownerAddr) {
    _owner = ownerAddr;
}
```

The **OwnableGuarded** contract contains a validation that prevents transferring ownership to the zero address:

```
function _transferOwnership(address newOwner) internal virtual {
    if (newOwner == address(0)) revert OwnerAddressRequired();
```

However, this validation is bypassed during deployment because the constructor assigns `_owner` directly without invoking `_transferOwnership`. As a result, no verification ensures that `ownerAddr` is a valid non-zero address.

If the contract is deployed with `ownerAddr` set to `address(0)`, the following conditions occur:

- All functions protected by the `onlyOwner` modifier become permanently inaccessible.
- Administrative functions such as `enableSender()` and `disableSender()` cannot be executed.
- Ownership cannot be recovered because `transferOwnership()` requires the caller to be the current owner.

Assets:

- /src/core/SendersWhitelist.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: Constructor initialization should enforce the same validation used in ownership transfer logic to ensure the owner address is not the zero address.

Example mitigation approach:

```
constructor(address ownerAddr) {
    if (ownerAddr == address(0)) revert OwnerAddressRequired();
    _owner = ownerAddr;
}
```

Resolution: Fixed in `b544e1e`

The `SendersWhitelist` constructor now validates the owner address before assignment:

```
constructor(address ownerAddr) {
    if (ownerAddr == address(0)) revert OwnerAddressRequired();
    _owner = ownerAddr;
}
```

This enforces the same invariant as `_transferOwnership`, preventing deployment with a zero-address owner.

F-2026-15328 - Missing Events in Sender Whitelist Management Functions Reduces Administrative Transparency - Info

Description:

The **SendersWhitelist** contract modifies whitelist state through `enableSender` and `disableSender` without emitting events. The absence of event logging reduces transparency and prevents reliable off-chain tracking of whitelist changes.

The **SendersWhitelist** contract manages a whitelist of approved message senders through the `_addresses` mapping:

```
mapping(address => bool) internal _addresses;
```

Two functions allow the owner to modify this whitelist:

```
function enableSender(address addr) external override onlyOwner {
    if (_addresses[addr]) revert AlreadyApproved();
    _addresses[addr] = true;
}

function disableSender(address addr) external override onlyOwner {
    _addresses[addr] = false;
}
```

Both functions modify critical contract state but do not emit events reflecting these changes.

Event emission is an important component of smart contract design because events allow external systems to track state transitions without requiring expensive storage reads or historical state reconstruction. Monitoring systems, indexing services, and cross-chain infrastructure frequently rely on event logs to observe contract activity.

Assets:

- /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: Events should be emitted whenever whitelist state is modified to provide a reliable on-chain record of administrative actions.

Example implementation:

```
event SenderEnabled(address indexed sender);
event SenderDisabled(address indexed sender);

function enableSender(address addr) external override onlyOwner {
    if (_addresses[addr]) revert AlreadyApproved();
    _addresses[addr] = true;
    emit SenderEnabled(addr);
}

function disableSender(address addr) external override onlyOwner {
    _addresses[addr] = false;
    emit SenderDisabled(addr);
}
```

Resolution: Fixed in `b544e1e`

The issue has been addressed by adding `SenderEnabled` and `SenderDisabled` events to the **ISendersWhitelist** interface and emitting them in the corresponding `enableSender()` and `disableSender()` functions of **SendersWhitelist**.

F-2026-15343 - Morpho V2 Exit Liveness Depends on External Conditions - Info

Description:

InstantRedeemSubaccount deploys idle reference assets into an external ERC-4626 vault (`idleVault`) to earn yield. As a result, redemptions that depend on pulling funds back from `idleVault` inherit that external vault's withdrawal liveness, liquidity conditions, and operational constraints.

For Morpho V2 specifically, `maxWithdraw`/`maxRedeem` return `0` by design, so the subaccount relies on `convertToAssets` as an estimate of position value. That estimate reflects claim value, but it does not guarantee immediate exit liquidity in all external states. If Morpho liquidity is unavailable or exits are operationally constrained, `redeemAsset()`, vault liquidity pulls, or emergency withdrawal paths may fail once the subaccount's directly held idle balance is exhausted.

As a result:

- Redemption liveness depends partly on external Morpho exit conditions.
- `availableLiquidity()` should be interpreted as an estimate, not a hard guarantee of immediately withdrawable funds.
- If external exits are impaired, redemptions may need to rely on the subaccount's directly held idle balance or be paused operationally.

Assets:

- /src/subaccounts/InstantRedeemSubaccount.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: Document clearly that `idleVault` liquidity is subject to external protocol liveness and may not always be instantly withdrawable.

Resolution: Fixed in `b544e1e`

The remediation adds explicit NatSpec documentation to `availableLiquidity()` stating that estimates are upper-bound and subject to Morpho V2 market liquidity constraints at call time, and that the idle vault must be operational (liveness dependency) for ERC4626 view calls to return accurate results. The `_withdrawFromIdleVault()` documentation similarly notes that actual withdrawable amounts depend on Morpho market liquidity.

F-2026-15345 - setRedemptionsPaused Duplicates Role Checks Instead of Reusing Existing Access-Control Modifiers - Info

Description:

`setRedemptionsPaused` reimplements the same owner/operator authorization logic already encapsulated in `onlyVaultOwnerOrOperator`.

The contract already defines a reusable delegated access-control modifier:

```
modifier onlyVaultOwnerOrOperator() {
    address vaultOwner = IVaultState(vault).owner();
    address vaultOperator = IVaultState(vault).operatorAddress();
    if (msg.sender != vaultOwner && msg.sender != vaultOperator) revert OnlyV
aultOwnerOrOperator();
    _;
}
```

However, `setRedemptionsPaused()` duplicates the same authorization logic inline:

```
function setRedemptionsPaused(bool paused) external {
    if (paused) {
        address vaultOwner = IVaultState(vault).owner();
        address vaultOperator = IVaultState(vault).operatorAddress();
        if (msg.sender != vaultOwner && msg.sender != vaultOperator) revert O
nlyVaultOwnerOrOperator();
    } else {
        if (msg.sender != IVaultState(vault).owner()) revert OnlyVaultOwner()
;
    }
    redemptionsPaused = paused;
    emit RedemptionsPausedUpdated(paused);
}
```

There is no direct security impact under the current implementation. The risk is maintainability and future authorization inconsistency:

- if delegated role semantics change,
- if additional authorization conditions are added later,
- or if one code path is updated without the other.

That can create subtle permission drift in a security-sensitive control function.

Assets:

- /src/subaccounts/InstantRedeemSubaccount.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: Reuse `onlyVaultOwnerOrOperator` on `setRedemptionsPaused` and keep the stricter owner-only rule as a single explicit conditional for the unpause path:

```
function setRedemptionsPaused(bool paused) external onlyVaultOwnerOrOperator
{
    if (!paused && msg.sender != IVaultState(vault).owner()) revert OnlyVault
Owner();

    redemptionsPaused = paused;
    emit RedemptionsPausedUpdated(paused);
}
```

Resolution: Fixed in `b544e1e`

The issue has been addressed by updating `setRedemptionsPaused` to use the `onlyVaultOwnerOrOperator` modifier, while retaining the owner-specific unpause restriction as an inline conditional:

```
function setRedemptionsPaused(bool paused) external onlyVaultOwnerOrOperator
{
    if (!paused && msg.sender != IVaultState(VAULT).owner()) revert OnlyVault
Owner();
    redemptionsPaused = paused;
    emit RedemptionsPausedUpdated(paused);
}
```

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	b544e1eb86a0515e0626d817035d257e4b0f0cd7
Whitepaper	N/A
Requirements	https://github.com/fractal-protocol/august-contracts-v2/docs; NatSpec
Technical Requirements	README.md

Asset	Type
/src/core/OwnableGuarded.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/subaccounts/InstantRedeemSubaccount.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.