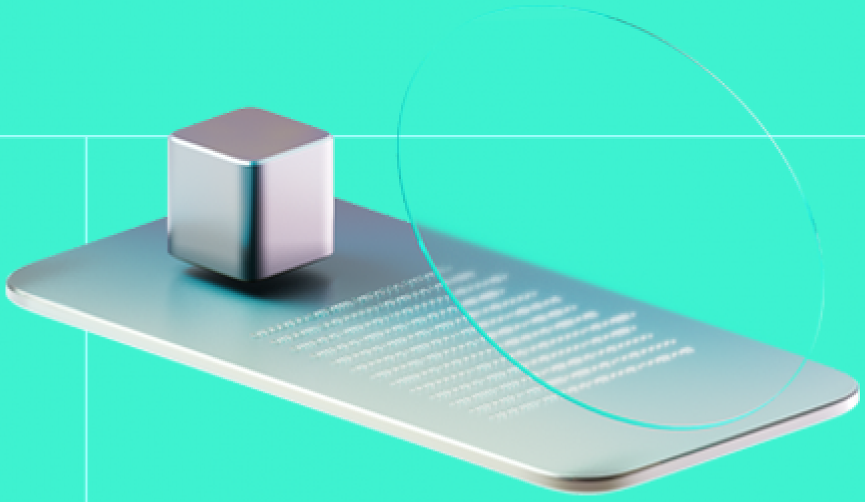
Smart Contract Code Review And Security Analysis Report



Customer: Upshift Finance

Date: 24/09/2025

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

Upshift Vault is a core ERC-4626 vault that enables users to deposit funds while earning yield through deployment to August subaccounts, which manage strategies securely across multiple chains. It simplifies user experience with a single reference token, supports multi-chain DeFi opportunities, and enforces strict roles and permissions for secure capital management.

Document

Name	Smart Contract Code Review and Security Analysis Report for Upshift Finance
Audited By	David Camps Novi, Georgi Krastenov
Approved By	Ivan Bondar
Website	https://www.upshift.finance/
Changelog	01/09/2025 - Preliminary Report
	24/09/2025 - Final Report
Platform	Any EVM-compatible chain
Language	Solidity
Tags	ERC4626; Upgradable; Yield Farming; Centralization; Claims; Vault
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/fractal-protocol/august-contracts-v2
Commit	a1f8599e73796c75d0a62ea79dbff78fb97f0b98
Remediation Commit	95c8cb1f3cb27e513b4bb20424690fbaefb2fdbf
2nd Remediation Commit	e5a91bbceecb5439943bb443a4ed3dc1b277356e

Audit Summary

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

21	15	4	2
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	7
Low	7

Vulnerability	Severity	Status
F-2025-12493 - Incorrect Signature Deadline Validation in permit()	High	Fixed
F-2025-12412 - Tokens WETH and DAI Can Not Be Reference Asset	Medium	Fixed
F-2025-12498 - Malicious User Can Block The processAllClaimsByDate Function For a Specific Epoch	Medium	Fixed
F-2025-12500 - Missing Storage Gaps in Upgradeable Base Contracts	Medium	Fixed
F-2025-12518 - External Assets Valuation Fixed at SubAccount Transfer May Cause TVL Inaccuracy	Medium	Accepted
F-2025-12522 - Direct Token Donations Can Distort TVL and Share Accounting	Medium	Accepted
F-2025-12492 - Incorrect Calculation in Total Assets Percentage Change	Medium	Mitigated
F-2025-12550 - Inaccurate Total Assets Valuation Due to Oracle and Conversion Logic Limitations	Medium	Mitigated
F-2025-12413 - Missing Stale Price Validation in Chainlink 's Agreggator latestRoundData() Call	Low	Fixed
F-2025-12489 - Not Time Locked Vault Apply Instant Redemption Fee	Low	Fixed
F-2025-12497 - Missing maxWithdrawalAmount Validation in Delayed Redemption Requests	Low	Fixed
F-2025-12501 - enableAsset() Allows Duplicate Entries in Vault Whitelisted Assets Array	Low	Fixed
F-2025-12520 - maxDepositAmount Not Enforced in Deposit Function	Low	Fixed

Vulnerability	Severity	Status
F-2025-12559 - Improper and Undocumented Handling of Performance Fees	Low	Fixed
F-2025-12563 - Change Percentage Bypassed for Null externalAssetsAmount in updateTotalAssets	Low	Fixed
F-2025-12336 - Redundant Errors	Info	Fixed
F-2025-12488 - Typos in Error Naming	Info	Fixed
F-2025-12502 - Unnecessary nonReentrant Modifier	Info	Fixed
F-2025-12540 - Unused Variable	Info	Fixed
F-2025-12494 - Floating Pragma	Info	Accepted
F-2025-12505 - Lack of Fee-on-Transfer Token Compatibility Corrupts Vault TVL and Accounting	Info	Accepted

Documentation quality

- Functional requirements are limited.
- Technical descriptions are limited.

Code quality

- The development environment is well-configured.
- Code architecture has a modular design with clear separation of concerns.
- NatSpec is present but does not extensively describe functionality.

Test coverage

Code coverage of the project is **58.29%** (statements coverage).

- Deployment and basic user interactions are not covered with tests.
- Interactions by several users are not tested thoroughly.

Table of Contents

System Overview **7**

Privileged Roles 7

Potential Risks **9**

Findings **11**

Vulnerability Details 11

Disclaimers 54

Appendix 1. Definitions **55**

Severities 55

Potential Risks 55

Appendix 2. Scope **56**

Appendix 3. Additional Valuables **58**

System Overview

The **Upshift Finance** system consists of an **OFT (Omnichain Fungible Token)** for cross-chain token transfers via LayerZero and a **vault** contract for asset management, The vault issues the aforementioned OFT as its shares, enforces withdrawal/redemption logic, and supports configurable fees and external asset reporting. Owner/operator roles control updates, limits, and fee distribution.

Core Contracts

- **TokenizedVault** - An upgradeable ERC-4626 vault that issues receipt tokens, manages deposits/withdrawals, calculates share prices, and handles performance and management fees, while enforcing timelocks and emergency withdrawals. It integrates with whitelisted assets to standardize all deposits to a single reference asset.
- **EnableOnlyAssetsWhitelist** - Maintains a list of whitelisted assets for vault deposits, assigns Chainlink oracles to convert asset values into the reference asset, and enforces decimal consistency to ensure accurate vault accounting.
- **TimelockedVault** - Adds withdrawal timelocks and instant redemption fee logic to the vault, enforcing delayed withdrawals and tracking lag durations for security and proper fee application.
- **BridgeReceiptToken** - An ERC-20 token representing user shares in a vault that can be minted, burned, and locked, with cross-chain bridging enabled via LayerZero, ensuring controlled token issuance and secure transfer restrictions during timelocks or emergency scenarios.
- **OraclizedMultiAssetVault** - extends `OperableVault` to manage deposits, withdrawals, and subaccount interactions across multiple whitelisted assets

Fee structure

- **Management fee**: Charged on total value locked (TVL) in the vault.
- **Performance fee**: Charged when high watermark is exceeded; distributed to fee recipients.
- **Instant Redemption fee**: Charged for immediate withdrawals, incentivizing delayed redemption.

Privileged roles

- **Owner**:
  - Deploy upgradeable implementations via `ProxyFactory`
  - Update configurable parameters such as `maxWithdrawAmount` or `maxChangePercent`.
  - Update fee-related parameters such as fee receivers.
  - Add/Remove users from the whitelist.
  - Add/Remove subAccounts for yield strategies
  - Emergency withdraw the assets from the vault.
  - Update the underlying assets of the vault.
  - Deposit/Withdraw assets in the subAccount to generate yield.
  - Pause deposits and withdrawals.

    - Enable/disable subAccounts.
- **Operator**
    - Add/Remove subAccounts for yield strategies
    - Deposit/Withdraw assets in the subAccount to generate yield.
    - Pause deposits and withdrawals.
    - Enable/disable subAccounts.
    - Add/Remove users from the whitelist.
- **Whitelisted User**
    - Deposit assets in exchange for shares.
    - Redeem vault shares in exchange for reference assets.

Potential Risks

- In the `ProxyFactory`, each deployed proxy is controlled by a `ProxyAdmin` whose owner is set at deployment. Using a single EOA as the `ProxyAdmin` owner without safeguards creates a central point of failure and allows immediate upgrades to new implementations. Consider ownership being assigned to a multisig wallet, and introducing a timelock delay to provide review time before upgrades take effect.
- The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- The `BridgeableReceiptToken` contract relies on the `minters` and `burners` mappings to control access to the `mint` and `burn` functions, which are configured once via the `configure` method. If addresses other than intended vault contracts—such as admin wallets—are added as minters or burners, they can arbitrarily inflate the token supply through `mint()` or remove tokens from users via `burn()`. This risk is further amplified if an admin key is compromised or impersonated, potentially allowing malicious actors to manipulate balances and destabilize the system.
- The `lockTokens` function allows the contract owner to arbitrarily set or extend token locks for any user. This gives the owner unilateral control over when users can transfer or manage their tokens, creating a risk of misuse or disruption. If the owner account is compromised, an attacker could similarly restrict access to user funds. Consider introducing safeguards against repeated extensions of the locking period.
- Assets accepted as deposits into the vaults are whitelisted and cannot be removed once added. While this ensures only approved tokens are used, it creates a risk that if a whitelisted token becomes problematic—due to exploits, depegging, or other critical issues—it cannot be disabled or removed from the vault. This could expose the protocol and its users to financial losses or operational disruptions.
- The `updateTotalAssets` function in the Vault allows the owner or operator to update the `externalAssets` value, which may include assets held off-chain. Because the actual off-chain balance cannot be verified on-chain, the `externalAssetsAmount` parameter can be manipulated within the limits set by `maxAllowedChangePerc`. This introduces a trust assumption on the owner or operator and could result in a misrepresentation of total assets for external users or integrations relying on this value. Proper off-chain reconciliation and monitoring are recommended to mitigate this risk.
- Withdrawals in the system (`claim()` or `processAllClaimsByDate()`) depend on the Vault holding a sufficient balance of reference tokens, meaning users’ ability to redeem their shares relies on admins properly managing liquidity. If admins fail to ensure enough reference tokens are available, withdrawals will not execute, creating a risk that users cannot redeem their shares even though the vault may hold sufficient assets overall.
- The system relies on oracles to price vault assets relative to the reference asset. While the vault operates with tokens such as wBTC, wETH, USDC, or USDT, it is possible that the associated oracles provide prices for the underlying assets (BTC, ETH, USD) instead. Since

wrapped and pegged assets are not always perfectly aligned with their underlying counterparts, this may introduce slight pricing inconsistencies that can affect valuations.

- The functioning of the system significantly relies on specific external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds. Precisely, Vault assets are transferred from the vaults to subAccounts, which act as intermediaries responsible for managing external yield strategies. Since the subAccounts' logic and security are out of scope, the proper handling, safeguarding, and utilization of these assets introduce a dependency on external components that may affect the safety and availability of vault funds.
- Vault assets can be transferred to accounts of type `ACCOUNT_TYPE_SUBACCOUNT`, which must implement `IAllocableSubAccount()`, or to accounts of type `ACCOUNT_TYPE_WALLET`, which may correspond to any arbitrary address. Although this function is restricted by `onlyOwnerOrOperator`, the ability to send vault funds to any address introduces a high-risk vector, as misuse or compromise of privileged accounts could result in loss or mismanagement of vault assets.
- The `emergencyWithdraw()` function allows the vault owner to retrieve all assets from the vault at any time. While intended as an emergency mechanism, this introduces a high-risk vector because misuse or compromise of the owner account—such as through key theft—could result in complete loss of user funds, making the vault highly dependent on the security and trustworthiness of the owner.
- The current asset valuation mechanism via `_fromInputAssetToReferenceAsset()`, which relies on Chainlink oracles, only supports a limited set of token pairs. While this is sufficient for the currently required tokens, not all pairs can be integrated (e.g., DAI asset / USDC reference asset). This restriction is acceptable under the present system design, since the owner controls whitelisted assets, but it introduces a limitation for future extensibility. If additional unsupported token pairs are required, the system would need to adapt the conversion logic and/or redeploy contracts.

Findings

Vulnerability Details

F-2025-12493 - Incorrect Signature Deadline Validation in permit() - High

Description:

The `permit()` function implemented in the contract `BaseLayerZeroErc20` is intended to implement signature-based approvals. As part of this process, the `deadline` parameter is used to ensure signatures cannot be used after the signed expiration date.

However, in the current implementation of the deadline check, the validation is inverted, since it will try to block signatures with a deadline larger than the current timestamp, instead of blocking the deadlines before the current time (expired):

```
if (deadline > block.timestamp) revert ExpiredDeadline();
```

This causes the function to revert when the deadline is still in the future, while allowing execution when the deadline has already expired. As a result, the function becomes unusable under normal conditions and no valid signature can be processed.

Assets:

- /core/BaseLayerZeroErc20.sol [https://github.com/fractal-protocol/august-contracts-v2]

Status: Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	High

Recommendations

Remediation: Update the deadline check in order to block signature execution when the timestamp is larger than the signature deadline, as follows:

```
if (deadline < block.timestamp) revert ExpiredDeadline();
```

Resolution: Resolved in commit `1a841f0`. The deadline check is changed to `deadline < block.timestamp`.

Evidences

Steps to Reproduce:

Reproduce:

1. Deploy the contract and obtain the current `block.timestamp` (e.g., `T`).
2. Generate a valid signature with `deadline = T + 1 hour`.
    1. Expected behavior: Signature should be accepted until one hour has passed.
    2. Actual behavior: Transaction reverts immediately with `ExpiredDeadline()`.
3. Generate a signature with `deadline = T - 1 hour`.
    1. Expected behavior: Transaction should revert because the signature is expired.
    2. Actual behavior: Signature is incorrectly accepted.

F-2025-12412 - Tokens WETH and DAI Can Not Be Reference Asset - Medium

Description:

The Vault is designed to use a reference asset, which can be any ERC20 token such as USDC, WBTC, USDT, DAI, or WETH. However, two of these tokens, **WETH** and **DAI**, have 18 decimals, which may introduce a limitation regarding which assets the Vault can accept.

Users can deposit allowed assets that may differ from the reference asset. For example, a Vault with DAI as the reference asset may also accept deposits of USDC or WBTC.

To enable a token, the `enableAsset` function must be called by the contract owner. This function includes a validation check to ensure the reference asset decimals are greater than the oracle decimals:

```
if (REFERENCE_ASSET_DECIMALS > oracleDecimals) revert InvalidDecimalPlaces();
```

Tokens like WETH and DAI have 18 decimals, while the Chainlink oracles for their pairs typically return prices with 8 decimals. Examples:

1. ETH/USD Oracle: https://arbiscan.io/address/0x639Fe6ab55C921f74e7fac1ee960C0B6293ba612#readContract
2. DAI/USD Oracle: https://arbiscan.io/address/0xc5C8E77B397E531B8EC06BFb0048328B30E9eCfB#readContract

Due to this decimal mismatch, attempting to enable these tokens fails, causing the `enableAsset` call to revert with the `InvalidDecimalPlaces` error.

Assets:

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

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

A mismatch between token decimals (e.g., 18 for WETH/DAI) and Chainlink’s 8-decimal oracle causes `enableAsset` to revert, making certain tokens like WETH and DAI impossible to be reference asset..

Severity: Medium

Recommendations

Remediation: Consider reviewing the decimal handling logic in `enableAsset` or implementing a conversion/scaling mechanism to support reference assets with more than 8 decimals to be used as a reference asset.

Resolution: Resolved in commit `1a841f0`. The `REFERENCE_ASSET_DECIMALS > oracleDecimals` check was removed, and DAI and WETH tokens can now be enabled. The token should have at least 6 decimals.

F-2025-12492 - Incorrect Calculation in Total Assets Percentage Change - Medium

Description:

The `updateTotalAssets()` function is responsible for updating the accounting of assets involved in yield strategies, tracked by the state variable `externalAssets`. To prevent abrupt and potentially malicious updates, the function enforces a threshold mechanism that limits the maximum percentage change allowed for `externalAssets`.

The threshold is computed via the helper function `getMaxAllowedChange()`:

```
    function getMaxAllowedChange() public view returns (uint256) {
        // slither-disable-next-line timestamp
        if (block.timestamp + _TIMESTAMP_MANIPULATION_WINDOW < assetsUpdatedOn)
        {
            revert InvalidTimestamp();
        }

        // (Max change per day * Time interval in seconds since last update) / (60 * 60 * 24)
        return (maxChangePercent * (block.timestamp - assetsUpdatedOn))
            / uint256(86400);
    }
```

The variable `maxChangePercent` is configured to represent a percentage value in basis points. However, the calculation does not normalize this value by dividing by its base. As a result, the computed maximum allowed change is significantly larger than intended, which undermines the protective mechanism and deviates from the system’s requirements.

Assets:

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

Status: Mitigated

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Introduce basis points normalization by dividing `maxChangePercent` by the corresponding base in the calculation of the maximum allowed change.

```
 return (maxChangePercent * (block.timestamp - assetsUpdatedOn)) / uint256(86400) / basisPoints;
```

Resolution: The development team Mitigated this finding by scaling up the returned value of `getChangePercentage()` by `100` in order to compare the same scale in `updateTotalAssets()`:

```
    function updateTotalAssets(uint256 externalAssetsAmount) external nonReentrant ifConfigured onlyOwnerOrOperator {
        uint256 perChange = getChangePercentage(externalAssetsAmount);
        uint256 maxAllowedChangePerc = getMaxAllowedChange();

        // slither-disable-next-line timestamp
        if (perChange > maxAllowedChangePerc) revert MaxAllowedChangeReached();

        ...
    }
```

Therefore, the input parameter used in `updateMaxChangePercent()` should be carefully introduced by the caller of the method (i.e. `owner` or `operator` roles) in order to keep a consistency with the basis points of `maxChangePercent` used across the system.

```
    function updateMaxChangePercent(uint256 newValue) external nonReentrant ifConfigured onlyOwnerOrOperator {
        // Build the hash of this call and attempt to consume it. The call reverts if the hash can't be consumed.
        bytes32 h = keccak256(abi.encode(
            abi.encodeWithSignature(
                "updateMaxChangePercent(uint256)",
                newValue
            )
        ));
        maxChangePercent = newValue;
        emit MaxChangePercentUpdated(newValue);

        IResourceBasedTimelockedCall(scheduledCallerAddress).consume(h);
    }
```

The development team acknowledges the risk for the correct parameters being managed for `maxChangePercent`.

F-2025-12498 - Malicious User Can Block The processAllClaimsByDate Function For a Specific Epoch - Medium

Description:

The project allows users to claim all pending requests by a specific date using the `processAllClaimsByDate` function. This function iterates over requests in LIFO (Las In First Out) order, up to a specified maximum. For each request, the withdrawal fee is deducted, and then checks are performed:

```
            (assetsAmount, assetsAfterFee) = _previewRedemption(
                _burnableAmounts[dailyCluster][receiverAddr],
                withdrawalFee);

            if (assetsAmount > maxWithdrawalAmount) revert WithdrawalLimitReached();
            if (assetsAfterFee < 1) revert AmountTooLow();
```

The fee is calculated as:

```
        if (fee > 0) {
            applicableFee = (fee * assetsAmount) / 1e4;
            assetsAfterFee = assetsAmount - applicableFee;
        }
```

Since the `requestRedeem` function only checks that the requested shares are greater than 1, users can submit very small redemption amounts. When the `applicableFee` calculation results in 0 (due to integer division by 1e4), `assetsAfterFee` may be less than 1.

This can be a problem if a user does this near the end of the epoch, as it may block the possibility of claiming all requests. The transaction will revert with `AmountTooLow`, since requests are iterates one by one in LIFO order.

Assets:

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

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate:	2/5
Exploitability:	Independent
Complexity:	Simple
	A malicious or careless user can submit extremely small redemption requests that cause `processAllClaimsByDate` to revert.
Severity:	Medium

Recommendations

Remediation: Implement a minimum threshold check for redemption requests that accounts for the withdrawal fee, ensuring that `assetsAfterFee ≥ 1` for all requests. Alternatively, or additionally, consider making the process function more flexible so that a failure in one of the requests does not compromise all of them. This will make the system more reliable to unexpected failures and more resistant to DOS.

Resolution: Resolved in commit `1a841f0`. The check for fee too low is performed in the `requestRedeem` function.

F-2025-12500 - Missing Storage Gaps in Upgradeable Base Contracts - Medium

Description: The child contracts `BridgeableReceiptToken` and `TokenizedVault` are upgradeable and inherit from multiple base contracts, such as `OperableVault` or `BaseLayerZeroERC20`. Currently, these base contracts do not include explicit storage gaps, which are a best practice in upgradeable contracts to reserve unused storage slots for future variable additions without affecting the existing storage layout.

As a consequence, any future additions of state variables—either in these base contracts or in other child contracts that inherit from them—could overwrite existing storage, potentially causing critical data corruption or unexpected behavior in the proxy.

Assets:

- /tokenized-vaults/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/BridgeableReceiptToken.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/SendersWhitelist.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: It is recommended that all upgradeable base contracts in the project introduce storage gaps (e.g., `uint256[50] private __gap;`) to ensure safe extensibility, and that child contracts carefully consider the full inheritance chain when adding new state variables. Consider also support contracts that may be deployed as upgradeable contracts, such as `ResourceBasedTimelockedCall` to be candidates.

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of `uint256` so that each element reserves a 32 byte slot. Use the name `__gap` or a name starting with `__gap_` for the array so that OpenZeppelin Upgrades will recognize the gap.

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using `upgradeProxy` or just run the validations with `validateUpgrade` (see docs for Hardhat or Truffle). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

Resolution:

Resolved in commit `1a841f0`. A `uint256[10] private __gap` storage array reserves 10 slots for each upgradeable contract.

F-2025-12518 - External Assets Valuation Fixed at SubAccount Transfer May Cause TVL Inaccuracy - Medium

Description:

The project Vaults track assets deployed to external yield managers (`subAccounts`) using the `externalAssets` state variable. This variable represents the valuation of these assets in terms of the reference asset at the moment they are transferred to `subAccounts` managers.

```
    function depositToSubaccount(
        address inputAssetAddr,
        uint256 depositAmount,
        address subAccountAddr
    ) external nonReentrant ifConfigured onlyOwnerOrOperator {
        if (depositAmount < 1) revert InvalidAmount();

        uint8 accountType = whitelistedSubAccounts[subAccountAddr];
        if (accountType < 1) revert AccountNotWhitelisted();

        // Convert the input amount to the respective amount in reference tok
ens
        uint256 amountInReferenceAssets = (inputAssetAddr == _referenceAsset)
            ? depositAmount
            : _fromInputAssetToReferenceAsset(inputAssetAddr, depositAmount);

        externalAssets += amountInReferenceAssets;

        if (accountType == ACCOUNT_TYPE_SUBACCOUNT) {
            // Deposit funds in the sub account
            SafeERC20.safeApprove(
                IERC20(inputAssetAddr), subAccountAddr, depositAmount
            );
            IAllocableSubAccount(subAccountAddr).deposit(
                inputAssetAddr, depositAmount
            );
            SafeERC20.safeApprove(IERC20(inputAssetAddr), subAccountAddr, 0);
        } else {
            // Transfer the funds to a whitelisted wallet or EOA
            SafeERC20.safeTransfer(
                IERC20(inputAssetAddr), subAccountAddr, depositAmount
            );
        }
    }
```

A subtle risk arises because subsequent market fluctuations of these external assets are not reflected in TVL calculations. Specifically:

- If the market value of an external asset increases, the vault understates its TVL, which may result in LP shares being undervalued.
    - LP share price underestimation.
    - Too low accrued fees.
- If the market value decreases significantly, the vault overstates TVL, which can lead to:
    - LP share price overestimation.
    - Too large accrued fees.
    - Withdrawal calculations exceeding the actual liquid value of the vault, potentially preventing users from withdrawing the full expected amount.

Since `_convertToAssets()` and other functions rely on TVL for deposit, withdrawal, and fee calculations, inaccuracies in `externalAssets` valuation directly affect these critical operations.

Assets:

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

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to accurately track the actual valuation of the assets held in the vault. Some possible solutions are:

- Periodically, or on-demand, revalue externalAssets using up-to-date oracle prices or other reliable pricing mechanisms.
- Track strategy balances separately and compute TVL dynamically when `_getTotalAssetsValuation()` is called, including accrued yield or losses.

Resolution: The risk for the given finding is accepted. The owner will revalue the external assets as needed by calling the `updateTotalAssets` function.

F-2025-12522 - Direct Token Donations Can Distort TVL and Share Accounting - Medium

Description:

The system Vaults accounting relies on `_getTotalAssetsValuation()`, which aggregates balances of whitelisted assets held by the vault contract and tokens allocated to external strategies.

```
    function _getTotalAssetsValuation(address vaultAddr, uint256 externalAssets)
        internal
        view
        returns (uint256)
    {
        address assetAddr;
        uint256 assetBalance;
        uint256 balanceInReferenceTokens;
        uint256 t = _whitelistedAssets.length;

        uint256 acum =
            externalAssets + IERC20(REFERENCE_ASSET).balanceOf(vaultAddr);

        for (uint256 i; i < t; i++) {
            assetAddr = _whitelistedAssets[i];
            assetBalance = IERC20(assetAddr).balanceOf(vaultAddr);
            if (assetBalance > 0) {
                balanceInReferenceTokens =
                    _fromInputAssetToReferenceAsset(assetAddr, assetBalance);
                acum += balanceInReferenceTokens;
            }
        }

        // External assets + multi assets liquidity + liquidity of the reference token
        return acum;
    }
```

However, the aforementioned ERC20 tokens can also be transferred directly to the vault by any user, whitelisted or not, without invoking its `deposit()` function. Similarly, reference tokens are ERC20 that can also be sent directly to the contract.

This creates a serious inconsistency between the two inflows of tokens:

- When a user deposits via the official `deposit()` entry point, the corresponding LP shares are also minted, based on the ratio between the assets contributed and the vault's Total Value Locked (TVL).
- If a token transfer happens directly (bypassing `deposit()`), the vault balance of that token increases, raising the TVL without minting any shares. This creates an inconsistency.

Due to the rely on ERC20 balances that will not track properly the total asset valuation (TVL) according to the actual minting of shares, several consequences arise:

- **Depositor dilution**
  Share pricing depends on the ratio between TVL and total supply of shares. If TVL is artificially increased by donations, future depositors will receive fewer shares per unit deposited, effectively getting a worse exchange rate. This dilutes their position relative to existing participants.
- **Withdrawals over-credited**
  Since share-to-asset conversion is based on the inflated TVL, existing LPs can redeem their shares for more assets than they are entitled to. This allows early LPs to "cash out" part of the donated value, leaving subsequent depositors with potential losses.
- **Fee miscalculation**
  Protocol fees (e.g., management or performance fees) are typically charged as a percentage of TVL. With inflated balances, fees may be overestimated, resulting in the protocol collecting more fees than it should, further harming users.
- **Accounting mismatch**
  The design assumes that every token counted in TVL entered through the `deposit()` function, ensuring share issuance matches assets held. Donations break this invariant, leading to inconsistencies between economic reality and accounting logic. Over time, this can complicate reconciliations and introduce systemic risk.
- **Attack surface for griefing**
  Even if attackers gain no direct profit, they can disrupt the vault by repeatedly donating small amounts of reference assets. This creates unpredictable fluctuations in share pricing, undermining trust in the vault's correctness and potentially deterring real users from depositing.

Assets:

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

Status: Accepted

Classification

Impact Rate:	4/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: In order to prevent the distortion of the total asset valuation, it is recommended to introduce an accounting mechanism in the `deposit()` function that keeps track of the tokens deposited only through this function. This accounting of deposited tokens should be used as the reference to calculate the total asset valuation of the vault, instead of relying on the token balance which can be manipulated.

Resolution: The risk for the given finding is accepted. Donating tokens to the vault is considered real PnL.

F-2025-12550 - Inaccurate Total Assets Valuation Due to Oracle and Conversion Logic Limitations - Medium

Description:

The vault calculates total asset valuation by summing the value of all whitelisted tokens using the `_getTotalAssetsValuation` function. This function iterates over each whitelisted token, queries its balance from the vault, and converts it into the reference asset using the `_fromInputAssetToReferenceAsset` function:

```
        for (uint256 i; i < t; i++) {
            assetAddr = _whitelistedAssets[i];
            assetBalance = IERC20(assetAddr).balanceOf(vaultAddr);

            if (assetBalance > 0) {
                balanceInReferenceTokens = _fromInputAssetToReferenceAsset(assetAddr, assetBalance);
                acum += balanceInReferenceTokens;
            }
        }
```

The `_fromInputAssetToReferenceAsset` function uses the following formula to convert the input amount into the corresponding amount of reference tokens:

```
        (,int256 answer,,,) = IAggregatorV3Interface(oracleAddr).latestRoundData();
        if (answer < 1) revert InvalidOraclePrice();

        uint256 a = (uint256(answer) * amount) / (10 ** tokenDecimals);
        return a / (10 ** decimalsDiff);
```

The Chainlink oracle provides a limited set of token pairs, which imposes constraints on the project. For example, WBTC/USD does not exist as a direct pair; only BTC/USD is available. While the BTC/USD pair can be used as a generalization, the retrieved price may differ slightly from the exact value. Another limitation is that Chainlink does not provide inverse pairs such as USD/WBTC or USD/BTC. These constraints significantly restrict which tokens can be used as reference assets or whitelisted tokens.

If WBTC is used as the reference token and USDC and USDT are whitelisted, a potential issues may arise. For example, suppose the vault holds 30,000 USDC and 270,000 USDT, and the BTC price is $100,000. The total value of both whitelisted tokens should equal 3 BTC. However, the current formula used for conversion produces an incorrect result. Using the BTC/USD price feed, the calculation becomes:

```
100_000e8 * 30_000e6 / 1e6 / 1 + 100_000e8 * 270_000e6 / 1e6 / 1
```

Here, `tokenDecimals` is 0 and the oracle answer is 100_000e8. The final result differs from the expected 3e8 (3 BTC), leading to inaccurate reference asset valuation.

Assets:

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

Status: Mitigated

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Incorrect total asset valuation may lead to mispriced shares and incorrect redemptions.

Severity: Medium

Recommendations

Remediation: It is recommended to introduce a direction check to increase compatibility of reference asset pairs with oracle data feeds (e.g. BTC/USD vs USD/BTC).

Resolution: Mitigated in commit `e5a91bc`. The conversion formula was changed so that it does not depend on the oracle, reference, or enabled token decimals.

This solution works for the token pairs that are currently required by the system. However, not any pair of tokens can be supported. The development team is aware of this and acknowledges the risk, since the configuration of the token pairs relies on the system `owner`. In case the system requires other pairs of tokens that are not supported, new smart contracts shall be deployed.

F-2025-12413 - Missing Stale Price Validation in Chainlink 's Agreggator latestRoundData() Call - Low

Description: In the `EnableOnlyAssetsWhitelist` contract, the protocol uses a ChainLink aggregator to fetch the `latestRoundData()`, but there is no check if the return value indicates stale data. The only check present is for the `quoteAnswer` to be `> 0`. However, this alone is not sufficient.

```
// slither-disable-next-line unused-return
(,int256 answer,,,) = IAggregatorV3Interface(oracleAddr).latestRoundData();
if (answer < 1) revert InvalidOraclePrice();
```

This could lead to stale prices according to the Chainlink documentation.

Assets:

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

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 5/5

Exploitability: Dependent

Complexity: Medium

Only checking if `answer > 0` will not prevent retrieving a stale price.

Severity: Low

Recommendations

Remediation: Add the following checks:

```
(uint80 quoteRoundID, int256 quoteAnswer,, uint256 quoteTimestamp, uint80 quoteAnsweredInRound) =
    AggregatorV3Interface(oracleAddr).latestRoundData();

if (quoteAnswer < 1) revert InvalidOraclePrice();
if (quoteAnsweredInRound < quoteRoundID) revert StalePrice();
if (quoteTimestamp == 0 ) revert RoundNotComplete();
if (block.timestamp - quoteTimestamp > VALID_TIME_PERIOD) revert InvalidTimePeriod();
```

Resolution: Resolved in commit `f09c54f`. All of the suggested checks have been added.

F-2025-12489 - Not Time Locked Vault Apply Instant Redemption Fee - Low

Description:

When `lagDuration < 1`, redemptions are executed immediately via:

```
// If the vault is not time-locked then redeem the tokens immediately.
if (lagDuration < 1) {
    _executeRedemption(shares, receiverAddr, false);
    return (type(uint256).max, 0, 0, 0);
}
```

Passing `isInstant = false` correctly results in charging the withdrawal fee for the `requestRedeem` function.
However, the `instantRedeem` function always charges the instant redemption fee, even when `lagDuration < 1`, creating inconsistent behavior and incorrect fee applying.

Assets:

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

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Inconsistent fee logic between redemption paths may charge users the wrong fee when the vault is **not time-locked** (`lagDuration < 1`).

Severity: Low

Recommendations

Remediation: Apply the withdrawal fee in both `instantRedeem` and `requestRedeem` functions when `lagDuration < 1`.

Resolution: Resolved in commit `1a841f0`. If the vault has no time-lock (`lagDuration < 1`), calling the `requestRedeem` function will revert with a `VaultNotTimelocked` error, since the user is expected to call the `instantRedeem()` function instead.

F-2025-12497 - Missing maxWithdrawalAmount Validation in Delayed Redemption Requests - Low

Description:

When a user tries to claim their funds, a check is performed to ensure the amount does not exceed the allowed `maxWithdrawalAmount`.

```
if (assetsAmount > maxWithdrawalAmount) revert WithdrawalLimitReached();
```

However, this check is applied in the wrong stage. In the current implementation, users transfer their shares into the vault when requesting a redemption via `requestRedeem()`, but the limit is only enforced later during claiming. As a result, if the requested amount exceeds the allowed maximum, the user's shares can become effectively stuck in the vault, preventing them from claiming their assets.

Assets:

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

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Users can lose their shares and be unable to claim their funds if they request an amount that is too large.

Severity: Low

Recommendations

Remediation: Ensure that the `maxWithdrawalAmount` validation is applied in the `requestRedeem` function and remove this check from the claiming functions, since the share price can change significantly.

Resolution: Resolved in commit `1a841f0`. The check for withdrawal limit is made when the user makes a redeem request, instead of in the `claim` and `processAllClaimsByDate` functions.

F-2025-12501 - enableAsset() Allows Duplicate Entries in Vault Whitelisted Assets Array - Low

Description:

The `enableAsset` function allows the same asset to be added to the vault whitelist multiple times because there is no check to prevent duplicates. Each call to `enableAsset` pushes the `assetAddr` into `_whitelistedAssets` and overwrites `_oracleOfInputAsset[assetAddr]` with the new parameters. This creates two main issues:

- **Array inflation:** `_whitelistedAssets` may contain duplicate entries, increasing gas costs and reducing efficiency in functions that iterate over the array, such as `_getTotalAssetsValuation()`.
- **Parameter overrides:** While the oracle info is updated correctly, repeated additions may overwrite previously configured values, which could lead to confusion or unexpected behavior.

```
    function enableAsset(address assetAddr, address oracleAddr)
        external
        override
        nonReentrant
        onlyOwner
    {
        if ((oracleAddr == address(0)) || (assetAddr == address(0))) {
            revert ZeroAddressError();
        }
        if (_whitelistedAssets.length > 30) revert WhitelistLimitReached();
        if (assetAddr == REFERENCE_ASSET) revert ReferenceAssetNotPermitted()
;

        _whitelistedAssets.push(assetAddr);

        uint8 tokenDecimals = ERC20(assetAddr).decimals();
        uint8 oracleDecimals = IAggregatorV3Interface(oracleAddr).decimals();
        if (REFERENCE_ASSET_DECIMALS > oracleDecimals) {
            revert InvalidDecimalPlaces();
        }

        uint8 decimalsDiff = oracleDecimals - REFERENCE_ASSET_DECIMALS;

        _oracleOfInputAsset[assetAddr] = OracleInfo({
            oracleAddress: oracleAddr,
            tokenAddress: assetAddr,
            oracleDecimals: oracleDecimals,
            tokenDecimals: tokenDecimals,
            decimalsDiff: decimalsDiff
        });
    }
```

Assets:

- /core/EnableOnlyAssetsWhitelist.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: It is recommended to introduce a check in the `enableAsset()` method to prevent duplicated entries of assets.

Resolution: Resolved in commit `1a841f0`. The check is introduced to avoid duplicating asset entries. If an already enabled asset is tried to be enabled again, the transaction will revert with `AssetAlreadyEnabled`.

F-2025-12520 - maxDepositAmount Not Enforced in Deposit Function - Low

Description:

Vaults include a `maxDepositAmount` parameter that is set during their deployment and configuration via the `configure()`function:

```
maxDepositAmount = newConfig.maxDepositAmount;
```

This variable is intended to limit the maximum amount of assets a user can deposit into the vault, helping to manage vault capacity, risk exposure, and operational requirements.

However, the current implementation of the `deposit()` function does not check `maxDepositAmount`, meaning users can deposit amounts exceeding the intended limit.

```
    function deposit(address assetIn, uint256 amountIn, address receiverAddr)
        external
        nonReentrant
        ifConfigured
        ifDepositsNotPaused
        ifSenderWhitelisted
        ifAssetWhitelisted(assetIn)
        returns (uint256)
    {
        if (amountIn < 1) revert InvalidAmount();
        if (receiverAddr == address(0) || receiverAddr == address(this)) {
            revert InvalidReceiver();
        }

        uint256 shares = previewDeposit(assetIn, amountIn);

        if (shares < 1) revert InsufficientShares();

        // Log the event
        emit Deposit(assetIn, amountIn, shares, msg.sender, receiverAddr);

        // Transfer the input tokens
        SafeERC20.safeTransferFrom(
            IERC20(assetIn), msg.sender, address(this), amountIn
        );

        // Issue (mint) LP tokens to the receiver
        IMintableBurnable(lpTokenAddress).mint(receiverAddr, shares);
        return shares;
    }
```

Assets:

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

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Add a check in the `deposit()` function to enforce `maxDepositAmount`.

Resolution: Resolved in commit `1a841f0`. The max deposit amount check has been added.

F-2025-12559 - Improper and Undocumented Handling of Performance Fees - Low

Description: The `chargePerformanceFees()` function allows the vault owner or operator to specify the `feeAmount` manually when charging performance fees, instead of deriving it from a well-defined metric such as a percentage of the vault’s TVL or profits. Additionally, there is no clear documentation explaining how this number should be determined or how it affects the vault’s performance, LP share price, or TVL. This design allows the caller to input **arbitrary values**, potentially leading to:

- Excessive or unreasonable fee withdrawals beyond what would be expected under a defined fee policy.
- Inconsistent behavior with users’ expectations, as performance fees are not automatically proportional to vault performance.
- Risk of human error or malicious action if the owner or operator sets the `feeAmount` incorrectly or intentionally.

```
function chargePerformanceFees(uint256 feeAmount)
    external
    nonReentrant
    ifConfigured
    onlyOwnerOrOperator
{
    if (block.timestamp - watermarkUpdatedOn < watermarkTimeWindow) {
        revert highWatermarkDurationError();
    }

    uint256 t = performanceFeeRecipients.length;
    uint256[] memory amounts = new uint256[](t);

    for (uint256 i; i < t; i++) {
        uint256 collectableFee =
            (performanceFeeRecipients[i].percentage * feeAmount) / 1e6;
        if (collectableFee > feeAmount) {
            revert CollectableFeesExceeded(collectableFee, feeAmount);
        }
        if (collectableFee < 1) revert FeeAmountTooLow();

        amounts[i] = collectableFee;
    }

    watermarkUpdatedOn = block.timestamp;
```

```
        uint256 currentSharePrice = _getSharePrice();
        if (currentSharePrice <= highWatermark) revert highWatermarkViolation
();

        highWatermark = currentSharePrice;

        for (uint256 i; i < t; i++) {
            SafeERC20.safeTransfer(
                IERC20(_referenceAsset),
                performanceFeeRecipients[i].collectorAddress,
                amounts[i]
            );
        }
    }
```

Assets:

- /tokenized-vaults/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: Consider deriving `feeAmount` automatically from a deterministic calculation (e.g., a percentage of accrued profits since the last fee charge, or a fraction of TVL) rather than allowing arbitrary manual input. Additionally, provide clear documentation explaining the calculation, its intended use, and its impact on vault performance and LP shares. This would improve transparency, enforce predictable fee behavior, and reduce reliance on manual governance.

Resolution: Resolved in commit `17fd0bc`. The performance fee is a percentage of the total asset increase.

F-2025-12563 - Change Percentage Bypassed for Null externalAssetsAmount in updateTotalAssets - Low

Description:

The `updateTotalAssets()` function relies on `getChangePercentage()` to calculate the percentage change of external assets and enforce a maximum allowed change threshold. However, in the current implementation, if `externalAssetsAmount == 0`, `getChangePercentage()` returns `0`, which will always be below the `maxAllowedChange` limit. As a result, the owner or operator can set `externalAssets` to zero bypassing the intended threshold protection.

```
    function updateTotalAssets(uint256 externalAssetsAmount)
        external
        nonReentrant
        ifConfigured
        onlyOwnerOrOperator
    {
        uint256 perChange = getChangePercentage(externalAssetsAmount);
        uint256 maxAllowedChangePerc = getMaxAllowedChange();

        // slither-disable-next-line timestamp
        if (perChange > maxAllowedChangePerc) revert MaxAllowedChangeReached(
);

        externalAssets = externalAssetsAmount;
        assetsUpdatedOn = block.timestamp;
    }

    function getChangePercentage(uint256 externalAssetsAmount)
        public
        view
        returns (uint256)
    {
        uint256 perChange;
        if (externalAssetsAmount < 1 || externalAssets < 1) {
            perChange = uint256(0);
        } else {
            perChange = (externalAssetsAmount > externalAssets)
                ? ((externalAssetsAmount * 100) / externalAssets) - 100
                : ((externalAssets * 100) / externalAssetsAmount) - 100;
        }

        return perChange;
    }
```

Assets:

- /tokenized-vaults/base/OraclizedMultiAssetVault.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: It is recommended to revert the transaction when `externalAssetsAmount == 0` to prevent bypassing the threshold check. Alternatively, if setting `externalAssetsAmount = 0` is a valid use case, document it clearly and adapt `getChangePercentage()` and the `updateTotalAssets()` logic to handle zero values safely without bypassing the maximum change threshold.

Resolution: Resolved in commit `1a841f0`. If the external assets amount supplied by the sender is zero then the percentage of change is set to 100%

F-2025-12336 - Redundant Errors - Info

Description: The following errors are never used in the `OraclizedMultiAssetVault` smart contract.

```
error NegativePrice();
error ForbiddenAsset();
error InvalidHolder();
error MaxDepositAmountReached();
error InsufficientAssets();
```

Also, in the `TimelockedVault` contract:

```
error InsufficientAllowance();
error VaultNotTimelocked();
```

Assets:

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

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

The errors are redundant and will never be used

Severity: Info

Recommendations

Remediation: Remove the redundant errors.

Resolution: Resolved in commit `1a841f0`. The redundant errors are removed, and `MaxDepositAmountReached` and `VaultNotTimelocked` are used now.

F-2025-12488 - Typos in Error Naming - Info

Description: The `ITokenizedVault` interface defines errors with a lowercase `h` in its name, instead of the expected uppercase `H`. This is inconsistent with other defined errors across the codebase, which follow PascalCase.

```
 error highWatermarkViolation();
error highWatermarkDurationError();
```

Assets:

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

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Inconsistent error naming reduces readability and breaks naming conventions.

Severity: Info

Recommendations

Remediation: Rename the error to follow the same naming convention as the rest of the project (use uppercase `H`).

Resolution: Resolved in commit `1a841f0`. Both errors follow the naming convention used in the rest of the project.

F-2025-12494 - 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 (`^0.8.19`, `^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:

- /tokenized-vaults/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /tokenized-vaults/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /tokenized-vaults/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /tokenized-vaults/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/BaseLayerZeroErc20.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/BridgeableReceiptToken.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/ProxyFactory.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]
- /core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]

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

Status: Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	2/5
Exploitability:	Dependent
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 0.8.19;` (or desired version).

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: The risk for the given finding is accepted. The source code is planned to be compatible with any EVM chain supporting Paris or higher (Solidity 0.8.20+), with the appropriate version selected at deployment for each target network.

F-2025-12502 - Unnecessary nonReentrant Modifier - Info

Description: The `enableSender` and `disableSender` functions have a `nonReentrant` modifier, which prevents reentrant calls.

```
    /**
     * @notice Enables the address specified.
     * @param addr The address to enable.
     */
    function enableSender(address addr) external override nonReentrant onlyOw
ner {
        if (_addresses[addr]) revert AlreadyApproved();
        _addresses[addr] = true;
    }

    /**
     * @notice Disables the address specified.
     * @param addr The address to disable.
     */
    function disableSender(address addr) external override nonReentrant onlyO
wner {
        _addresses[addr] = false;
    }
```

Currently, reentrant calls to these functions are not possible, as no external calls are made during the enabling or disabling of a sender.

Assets:

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

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

The use of the `nonReentrant` modifier in the `enableSender` and `disableSender` functions is intended to prevent reentrancy calls, which are currently not possible. This only adds unnecessary complexity to the code.

Severity: Info

Recommendations

Remediation: Remove the `nonReentrant` modifier from the functions.

Resolution: Resolved in commit `1a841f0`. The unnecessary modifier is removed.

F-2025-12505 - Lack of Fee-on-Transfer Token Compatibility Corrupts Vault TVL and Accounting - Info

Description:

The Vault does not support fee-on-transfer (FoT) tokens. The `deposit()` function calculates shares to mint based on the `amountIn` parameter before transferring tokens to the vault. For FoT tokens, the actual amount received is lower than `amountIn`, but the shares to mint are calculated as if the full amount was received.

```
function deposit(address assetIn, uint256 amountIn, address receiverAddr)
    external
    nonReentrant
    ifConfigured
    ifDepositsNotPaused
    ifSenderWhitelisted
    ifAssetWhitelisted(assetIn)
    returns (uint256)
{
    if (amountIn < 1) revert InvalidAmount();
    if (receiverAddr == address(0) || receiverAddr == address(this)) {
        revert InvalidReceiver();
    }

    uint256 shares = previewDeposit(assetIn, amountIn);

    if (shares < 1) revert InsufficientShares();

    // Log the event
    emit Deposit(assetIn, amountIn, shares, msg.sender, receiverAddr);

    // Transfer the input tokens
    SafeERC20.safeTransferFrom(
        IERC20(assetIn), msg.sender, address(this), amountIn
    );

    // Issue (mint) LP tokens to the receiver
    IMintableBurnable(lpTokenAddress).mint(receiverAddr, shares);

    return shares;
}
```

This lack of compatibility with FoT tokens is also present in the `depositToSubaccount()` logic: when assets are transferred to strategy `subAccounts`, the value of the transferred assets is taken as the full input amount and recorded into `externalAssets`, without accounting for the transfer fee. As a result, the `externalAssets` value is overstated, leading to an incorrect Total Value Locked (TVL) for the vault.

```
  function depositToSubaccount(
    address inputAssetAddr,
    uint256 depositAmount,
    address subAccountAddr
  ) external nonReentrant ifConfigured onlyOwnerOrOperator {
    if (depositAmount < 1) revert InvalidAmount();

    uint8 accountType = whitelistedSubAccounts[subAccountAddr];
    if (accountType < 1) revert AccountNotWhitelisted();

    // Convert the input amount to the respective amount in reference tok
ens
    uint256 amountInReferenceAssets = (inputAssetAddr == _referenceAsset)
      ? depositAmount
      : _fromInputAssetToReferenceAsset(inputAssetAddr, depositAmount);
    externalAssets += amountInReferenceAssets;

    if (accountType == ACCOUNT_TYPE_SUBACCOUNT) {
      // Deposit funds in the sub account
      SafeERC20.safeApprove(
        IERC20(inputAssetAddr), subAccountAddr, depositAmount
      );
      IAllocableSubAccount(subAccountAddr).deposit(
        inputAssetAddr, depositAmount
      );
      SafeERC20.safeApprove(IERC20(inputAssetAddr), subAccountAddr, 0);
    } else {
      // Transfer the funds to a whitelisted wallet or EOA
      SafeERC20.safeTransfer(
        IERC20(inputAssetAddr), subAccountAddr, depositAmount
      );
    }
  }
```

Since the TVL of the vault is a key component of the system, the aforementioned mis-calculation of the accounting for fee-on-transfer tokens have a severe impact in the vault accounting (minting of shares, withdrawal amount of reference assets, charging of fees, etc). Although the system already includes a whitelisting of assets, meaning only those tokens previously accepted by the admin team will be allowed in the vault, if any of them includes a transfer fee, the consequences are severe for the system.

Assets:

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

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is recommended to make the system compatible with fee-on-transfer tokens. This can be done by comparing the token balance before and after the transfer, in order to account only the actual amount of tokens being transferred.

Resolution: The risk for the given finding is accepted. The vault will support well-known tokens for deposits and withdrawals, such as USDC, USDT, DAI, WBTC, cbBTC, and WETH. The client is not planning to support any fee-on-transfer tokens.

F-2025-12540 - Unused Variable - Info

Description: Within the `processAllClaimsByDate` function, the variable `assetsToSend` is declared and updated but never used, returned, or otherwise contributing to the function's logic. This results in redundant state updates, unnecessary gas consumption, and reduced code clarity. While this does not pose a direct security risk, it negatively impacts efficiency and code quality.

Assets:

- /tokenized-vaults/base/TimelockedVault.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: Remove the unused `assetsToSend` variable and related updates to streamline execution and improve maintainability.

Resolution: Resolved in commit `1a841f0`. The redundant variable is removed.

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
Initial Commit	a1f8599e73796c75d0a62ea79dbff78fb97f0b98
Remediation Commit	95c8cb1f3cb27e513b4bb20424690fbaefb2fdbf
2nd Remediation Commit	e5a91bbceecb5439943bb443a4ed3dc1b277356e
Whitepaper	N/A
Requirements	https://docs.upshift.finance/architecture/vault-architecture
Technical Requirements	https://docs.upshift.finance/architecture/vault-architecture

Asset	Type
/core/BaseLayerZeroErc20.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/BaseReentrancy.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/BridgeableReceiptToken.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/DateUtils.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/ProxyFactory.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/tokenized-vaults/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/tokenized-vaults/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
/tokenized-vaults/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract

Asset	Type
/tokenized-vaults/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of **Upshift Finance**, Hacken followed its methodology by performing fuzz-testing on the project's main functions. Foundry, a tool used for testing, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Echidna fuzzing suite was prepared for this task, and throughout the assessment, **15** invariants were tested over **10,000** runs. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant ID	Test Case	Description	Test Result
INV-01	`test_fuzz_deposit_asset`	Shares are successfully minted to users and assets transferred to the vault	Passed
INV-02	`test_fuzz_deposit_asset_and_charge_management_fee`	Management fee is successfully charged as time passes	Passed
INV-03	`test_fuzz_deposit_and_withdraw_to_subAccount`	Assets are correctly deposited and withdrawn to and from subAccounts	Passed
INV-04	`test_emergency_withdraw`	Emergency withdraw successfully retrieves the tokens from the vault	Passed
INV-05	`test_instant_redeem`	Users successfully exchange shares with assets via instant redeem	Passed
INV-06	`test_request_redeem_and_claim`	Users successfully redeem shares for reference assets via request and claim	Passed
INV-07	`test_total_assets_consistency`	Total vault assets must equal reference asset balance + external assets + whitelisted asset valuations across all operations	Passed
INV-08	`test_share_price_consistency`	Share price calculation maintains consistency: total assets / total supply ratio preserved during deposits, withdrawals, and fee operations	Passed
INV-09	`test_high_watermark_monotonic_and_timed`	High watermark never decreases and only updates after configured time window	Passed

Invariant ID	Test Case	Description	Test Result
INV-10	`test_timelock_cannot_execute_early`	Timelocked operations never execute before scheduled timestamp	Passed
INV-11	`test_only_resource_can_consume`	Only the RESOURCE can consume scheduled hashes; owner/operator cannot consume	Passed
INV-12	`test_hash_integrity_and_immutability`	Scheduled hashes maintain integrity until cancel/consume; zero-hash and duplicates rejected	Passed
INV-13	`test_schedule_and_cancel_authorization`	Scheduling and cancellation restricted to authorized senders (resource/owner/operator) only	Passed
INV-14	`test_reference_asset_immutable_and_not_whitelisted`	Reference asset cannot be added to whitelist and remains immutable across operations	Passed
INV-15	`test_max_withdrawal_limit_enforced`	Withdrawal requests not exceed maximum withdrawal amount limit	Passed

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.