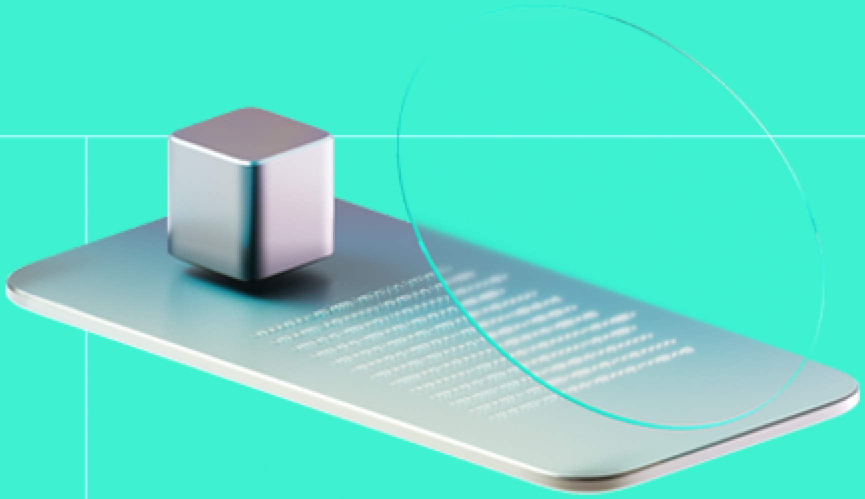
Smart Contract Code Review And Security Analysis Report



Customer: Upshift Finance

Date: 18/12/2025

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
Approved By	Ivan Bondar
Website	https://www.upshift.finance/
Changelog	16/12/2025 - Preliminary Report; 18/12/2025 - Final Report;
Platform	Any EVM-compatible chain
Language	Solidity
Tags	ERC4626; Upgradable; 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	6fbcaf5

Audit Summary

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

9	0	6	3
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	4
Low	2

Vulnerability	Severity	Status
F-2025-14332 - Inaccurate Total Assets Valuation Due to Oracle and Conversion Logic Limitations	Medium	Mitigated
F-2025-14333 - Direct Token Donations Can Distort TVL and Share Accounting	Medium	Accepted
F-2025-14334 - External Assets Valuation Fixed at SubAccount Transfer May Cause TVL Inaccuracy	Medium	Accepted
F-2025-14335 - Incorrect Calculation in Total Assets Percentage Change	Medium	Mitigated
F-2025-14309 - Custom Signature Recovery May Reject Valid permit() Signatures	Low	Mitigated
F-2025-14312 - Vault Owner Can Set Fees Outside Safe Limits	Low	Accepted
F-2025-14311 - Missing Events for Key State Changes	Info	Accepted
F-2025-14315 - Missing Safeguards When Assigning Vault Ownership	Info	Accepted
F-2025-14336 - Lack of Fee-on-Transfer Token Compatibility Corrupts Vault TVL and Accounting	Info	Accepted

Documentation quality

- Functional requirements are limited.
- Technical description is 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 **0%** (branch coverage).

- Tests are not properly configured, leading to running errors.

Table of Contents

System Overview **6**
Privileged Roles 6
Potential Risks **8**
Findings **10**
Vulnerability Details 10
Disclaimers 31
Appendix 1. Definitions **32**
Severities 32
Potential Risks 32
Appendix 2. Scope **33**
Appendix 3. Additional Valuables **35**

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.
- The `updatePerformanceFee()` and `updateInstantRedemptionFee()` functions allow the Vault owner to modify fee parameters at any time, which may result in users experiencing changes to costs after engaging with the Vault. This could affect user expectations and influence participation or liquidity in the protocol.
- The `updateLimits()` function allows the Vault owner to modify `maxWithdrawalAmount` at any time. Users who have already engaged with the Vault may be subject to new withdrawal limits, potentially restricting their ability to exit the system as anticipated and affecting user experience and confidence in the protocol.

Findings

Vulnerability Details

F-2025-14332 - 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(as
setAddr, 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).latestRoundD
ata();
        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.

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-2025-14333 - 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 externalAsse
ts)
        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 refere
nce 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.

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-2025-14334 - 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.

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-2025-14335 - 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 < assetsUpdatedO
n)
        {
            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.

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(864
00) / 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 nonReen
trant 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 i
fConfigured onlyOwnerOrOperator {
        // Build the hash of this call and attempt to consume it. The call re
verts 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-2025-14309 - Custom Signature Recovery May Reject Valid permit() Signatures - Low

Description: The `permit()` function in the `BaseLayerZeroErc20` abstract contract performs signature recovery using a custom wrapper around the EVM `ecrecover()` primitive. The implementation includes common best practices such as enforcing `s` requirements, using nonces for replay protection, and applying EIP-712 domain separation.

```
     function permit(...) external override nonReentrant {
        ...

        // Attempt to recover the signature
        address signer = _recover(h, v, r, s);
        if (signer != holderAddr) revert InvalidSigner();

        ...
    }

    function _recover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) internal pure returns (address) {
        ...
        if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
            revert InvalidSignatureComponentS();
        }

        // If the signature is valid (and not malleable), return the signer address
        address signer = ecrecover(hash, v, r, s);
        if (signer == address(0)) revert InvalidSignature();

        return signer;
    }
```

However, the current signature recovery mechanism can introduce subtle compatibility differences with signatures produced by widely used off-chain signing libraries, particularly regarding the encoding of the `v` component. In such cases, signatures that are otherwise valid may not be recovered as expected.

This behavior may lead to some `permit()` calls failing under specific signing configurations, potentially affecting user experience and integrations that rely on standardized signature handling.

Assets:

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

Status: Mitigated

Classification

Impact Rate:	2/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Medium
Severity:	Low

Recommendations

Remediation: Replace the `_recover()` logic with OpenZeppelin's `ECDSA.recover()` in `permit()`:

```
function permit(...) external override nonReentrant {
    ...

    // Attempt to recover the signature
    address signer = ECDSA.recover(digest, v, r, s);
    if (signer != holderAddr) revert InvalidSigner();

    ...
}
```

Resolution: Mitigated by the team, provided the following feedback about their signature management:

> We privilege security and best practices rather than backwards compatibility with old legacy EIP standards (example: EIP-2 and others). As a result, we privilege a clearly defined, strongly typed signature scheme (vrs) as opposed to a bunch of malleable signatures (65 bytes or more).

F-2025-14312 - Vault Owner Can Set Fees Outside Safe Limits - Low

Description: The `updatePerformanceFee()` and `updateInstantRedemptionFee()` functions allow the Vault `owner` to set the respective fee values without any restrictions or validation on the input range.

- TokenizedVault

```
    function updatePerformanceFee(uint256 newValue) external nonReentrant onl
yOwner {
        performanceFeeRate = newValue;
    }
```

- TimelockedVault

```
    function updateInstantRedemptionFee(
        uint256 newValue,
        bool pKeepFeeInVault
    ) external nonReentrant onlyOwner {
        instantRedemptionFee = newValue;
        keepFeeInVault = pKeepFeeInVault;
    }
```

As a result, fees can be set to unreasonably high values after users have already engaged with the protocol’s Vaults, allowing abusive behavior and unexpected losses for users. Since these parameters directly affect the cost of using the system, unrestricted updates may undermine trust, user experience, and participation, and could have a negative impact on protocol TVL and reputation.

Assets:

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

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability:	Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: It is recommended to introduce a validation of the input fee arguments in order to enforce reasonable bounds on fee parameters. For example, ensure that `performanceFeeRate` and `instantRedemptionFee` cannot exceed a predefined maximum (e.g., 20%); or validate both upper and lower bounds (e.g. 5% to 10%).

Resolution: Accepted by the development team, providing the following feedback:

> Any arbitrary maximum limit would have to be case specific and opens the door for an edge case where the instantRedemptionFee/performanceFeeRate would have to be much higher for a business use case. In terms of trade-offs, it is easier to refund a vault for which fees were over-charged than to be constrained by an arbitrary max value in perpetuity.

F-2025-14311 - Missing Events for Key State Changes - Info

Description: The `updatePerformanceFee()` and `updateInstantRedemptionFee()` functions modify key protocol parameters but do not emit events. Without events, off-chain monitoring tools and users cannot reliably track changes in real time. This reduces transparency and may negatively impact user perception of fairness or trustworthiness.

- TokenizedVault

```
    function updatePerformanceFee(uint256 newValue) external nonReentrant onl
yOwner {
        performanceFeeRate = newValue
    }
```

- TimelockedVault

```
    function updateInstantRedemptionFee(
        uint256 newValue,
        bool pKeepFeeInVault
    ) external nonReentrant onlyOwner {
        instantRedemptionFee = newValue;
        keepFeeInVault = pKeepFeeInVault;
    }
```

Assets:

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

Status: Accepted

Classification

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

Recommendations

Remediation: It is recommended to emit events in the reported methods `updatePerformanceFee()` and `updateInstantRedemptionFee()`.

Resolution: Accepted by the development team, providing the following feedback:

> Events consumption is a responsibility of the consumer rather than the publisher

F-2025-14315 - Missing Safeguards When Assigning Vault Ownership - Info

Description: The `configure()` function in `TokenizedVault` assigns the Vault `_owner` address without validating that it `newConfig.futureOwnerAddress` is non-zero. If the zero address is mistakenly provided, ownership of the Vault would be irreversibly assigned to `address(0)`, effectively disabling owner-only functionality and preventing further administrative actions.

```
  function configure(
      ConfigInfo memory newConfig
  ) public virtual override nonReentrant onlyOwner ifNotConfigured {
      ...
      _owner = newConfig.futureOwnerAddress;


      ...
  }
```

Assets:

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

Status: Accepted

Classification

Impact Rate:	3/5
Likelihood Rate:	1/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Consider using one of the following approaches:

- Implement a two-step ownership assignment mechanism (e.g., `Ownable2Step` pattern), where a pending owner is first set and must explicitly accept ownership. This significantly reduces the risk of accidental misconfiguration and irreversible loss of ownership, or

- Enforce a zero-address validation when setting the owner to ensure ownership is always assigned to a valid address:

```
require(newConfig.futureOwnerAddress != address(0), "Owner cannot be zero add
ress");
```

Resolution: Accepted by the development team, providing the following feedback:

> The caller/deployer is responsible for providing the right address of the owner at deployment time

F-2025-14336 - 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.

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.

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	6fbcaf5
Whitepaper	N/A
Requirements	https://docs.upshift.finance/architecture/vault-architecture
Technical Requirements	https://docs.upshift.finance/architecture/vault-architecture

Asset	Type
src/core/BaseLayerZeroErc20.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/BaseReentrancy.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/BridgeableGovernanceToken.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/BridgeableReceiptToken.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/DateUtils.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/EnableOnlyAssetsWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/GuardedProxyOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/Ownable2StepsGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/OwnableGuarded.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/ProxyAdminOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/ProxyFactory.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/ProxyFactoryOwnable2Steps.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/ResourceBasedTimelockedCall.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/core/SendersWhitelist.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/base/OperableVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract

Asset	Type
src/tokenized-vaults/base/OraclizedMultiAssetVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/base/TimelockedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/deployment/Parameters.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/ITokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/MasterDeployer.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract
src/tokenized-vaults/TokenizedVault.sol [https://github.com/fractal-protocol/august-contracts-v2]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

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

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

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

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

Frameworks and Methodologies

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