BITVAULT
Security Review

SEPTEMBER, 2025

www.chaindefenders.xyz
https://x.com/DefendersAudits

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

An ERC-7540 compliant vault giving capital managers maximum flexibility and ease to focus on their bread and butter, optimised returns.

Disclaimer

The Chain Defenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

PR

PR 205

Scope

Id	Files in scope
1	AsyncVault.sol
2	OracleVault.sol
3	BaseControlledAsyncRedeem.sol
4	BaseERC7540.sol

Roles

Id	Roles
1	Owner
2	Pauser
3	User

Executive Summary

Issues found

Severity	Count	Description
High	1	Critical vulnerabilities
Medium	1	Significant risks
Low	10	Minor issues with low impact
Informational	-	Best practices or suggestions
Gas	-	Optimization opportunities

Findings

High

High 01 Incorrect Share Transfer For Withdrawal Incentive Fee When Operator != Owner

Location

AsyncVault.sol:300

Description

In the `AsyncVault` contract, the `requestRedeem` function allows an operator (`msg.sender`) to initiate a redemption on behalf of a share owner. This function calculates a `withdrawalIncentive` fee in shares (`feeShares`) and calls `handleWithdrawalIncentive` to transfer these fee shares to the `feeRecipient`.

The vulnerability lies within the `handleWithdrawalIncentive` function. It incorrectly uses `msg.sender` as the source of the shares for the fee transfer. In a delegated redemption scenario where an operator calls `requestRedeem` for a different owner, `msg.sender` is the operator, not the owner of the shares being redeemed.

This leads to two incorrect outcomes:

1. If the operator does not have enough shares to cover the fee, the entire `requestRedeem` transaction will revert, preventing the owner's legitimate redemption request from being processed.
2. If the operator has sufficient shares, the fee is deducted from the operator's personal share balance instead of the owner's, effectively stealing shares from the operator. The owner's redemption proceeds, but the fee is not correctly paid from their redeemed amount.

Proof of Concept

Example scenario:

1. Alice (owner) approves Bob (operator) to request a redeem of 1000 shares (via allowance to Bob).
2. Bob calls `requestRedeem(1000, Alice, Alice)` (controller = owner for simplicity).
3. The function calculates `feeShares` (e.g., 50 if `withdrawalIncentive` = 5e16).
4. `AsyncVault`::`handleWithdrawalIncentive` attempts to transfer 50 shares from Bob to the fee recipient. If Bob has fewer than 50 shares, the transfer reverts. If Bob has 50+ shares, it transfers Bob's shares instead of Alice's, effectively stealing from Bob. Meanwhile, the subsequent `_requestRedeem(950, Alice, Alice)` proceeds (assuming it transfers from Alice), but the fee was not properly deducted from Alice's redeem amount.
5. This prevents legitimate delegated redeems or causes unintended transfers.

Recommendation

To resolve this, the `handleWithdrawalIncentive` function must be made aware of the actual share owner. This can be achieved by passing the `owner` address from `requestRedeem` to `handleWithdrawalIncentive` and using it as the source for the share transfer.

Apply the following changes to `AsyncVault.sol`:

```
//  ... existing code ...

        // Send the withdrawal incentive fee to the fee recipient
        handleWithdrawalIncentive(feeShares, fees_.feeRecipient, owner);

        // process request
        return _requestRedeem(shares - feeShares, controller, owner);

//  ... existing code ...

    function handleWithdrawalIncentive(
        uint256 feeShares,
        address feeRecipient,
        address owner
    ) internal virtual {
        if (feeShares > 0) {
            // Transfer feeShares from owner to feeRecipient

//  ... existing code ...

            SafeTransferLib.safeTransferFrom(
                this,
                owner,
                feeRecipient,
                feeShares
            );
        }
    }

//  ... existing code ...
```

Status

Fixed

Medium

Mid 01 Inaccurate Management Fee Calculation

Location

AsyncVault.sol:391

Description

The `_accruedManagementFee` function in `AsyncVault.sol` calculates the management fee based on the vault's current `totalAssets()`. This implementation is inconsistent with its own NatSpec documentation, which states that the fee should be calculated using the average Assets Under Management (AUM) over the fee period, similar to the trapezoid rule.

By using only the current `totalAssets()`, the calculation becomes inaccurate. If the vault's assets have increased since the last fee collection, the management fee will be overestimated, leading to an excessive fee being minted and unfairly diluting the shares of other investors. Conversely, if the assets have decreased, the fee will be underestimated, disadvantaging the fee recipient. The current implementation fails to correctly account for the AUM fluctuations over the fee period as intended.

Proof of Concept

1. Add the following public function to the `AsyncVault.sol` contract:

```
function accruedManagementFee() public view returns (uint256) {
    Fees memory fees_ = fees;

    return _accruedManagementFee(fees_);
}
```

2. Add the following test to the `AsyncVault.t.sol` file:

```
function testManagementFeeOverstated() public virtual {
    // Set management fee to 5%
    Fees memory newFees = Fees({
        performanceFee: 0.15e18, // 15%
        managementFee: 0.05e18, // 5%
```

```
            withdrawalIncentive: 0,
            feesUpdatedAt: uint64(block.timestamp),
            feeRecipient: feeRecipient,
            highWaterMark: ONE
        });

        vm.prank(owner);
        asyncVault.setFees(newFees);

        // Test management fee over one year
        vm.warp(block.timestamp + 365 days);

        uint256 managementFeeBefore = asyncVault.accruedManagementFee();

        // Double total assets to showcase overstated management fee
        asset.mint(address(asyncVault), 100e18);

        uint256 managementFeeAfter = asyncVault.accruedManagementFee();

        // Management fee is greater than the one before
        assertGt(managementFeeAfter, managementFeeBefore);
    }
```

Recommendation

To align the implementation with the documented intent and ensure fair fee calculation, the contract should be modified to track `totalAssets` at the last fee collection point and use it to compute the average AUM.

1. Add a `lastTotalAssets` field to the `Fees` struct to store the asset balance at the last fee checkpoint.
2. Update the `_setFees` function to initialize `lastTotalAssets` with the current `totalAssets()` when fees are first set.
3. Modify the `_takeFees` function to update `lastTotalAssets` to the current `totalAssets()` after fees are collected.
4. Adjust the `_accruedManagementFee` function to use the average AUM, calculated as `(totalAssets() + fees_.lastTotalAssets) / 2`, for the fee computation.

Status

Acknowledged

Low

Low 01 `_takeFees` Called Even When Vault Is Paused When Requesting A Redeem

Location

AsyncVault.sol:230

Description

The `AsyncVault` contract includes a `paused` state intended to halt deposits and fee collection, as a safety mechanism. Functions like `deposit`, `beforeWithdraw`, and `takeFees` correctly respect this state by preventing execution when the vault is paused.

However, the `requestRedeem` function unconditionally calls `_takeFees()` at the beginning of its execution. This means that even when the vault is paused, a user can initiate a redeem request and trigger the fee collection logic. This action will update fee-related state variables like `highWaterMark` and `feesUpdatedAt`, and potentially mint new shares for the fee recipient. This behavior is inconsistent with the intended purpose of the paused state, which should freeze all non-essential state changes and operations.

Recommendation

To ensure consistent behavior and enforce the purpose of the `paused` state, the call to `_takeFees()` within the `requestRedeem` function should be conditional and only execute if the vault is not paused.

```
// ... existing code ...
    ) public override returns (uint256 requestId) {
        require(shares >= limits.minAmount, "ERC7540Vault/min-amount");
        require(shares > 0, "ZERO_SHARES");

        // Take fees
        if (!paused) {
            _takeFees();
        }
```

```
        // Calculate the withdrawal incentive fee from the assets
        Fees memory fees_ = fees;
        uint256 feeShares = shares.mulDivDown(
// ...existing code...
```

Status

Fixed

Low 02 Removed Cancel Redeem Request Functionality

Location

AsyncVault.sol

Description

The `AsyncVault` contract implements an asynchronous redemption model where users first call `requestRedeem` to signal their intent to withdraw. This action transfers their shares to the contract and records a pending request. Then the vault's `owner` calls `fulfillMultipleRedeems` to fulfil the request and after that the user can actually withdraw/redeem.

The vulnerability arises because the contract lacks a function for users to cancel their pending redemption requests. If the vault owner becomes malicious, is compromised, or is otherwise unable or unwilling to call `fulfillMultipleRedeems` for a specific user's request, that user's shares are effectively trapped within the contract. Without a cancellation mechanism, the user has no way to reclaim their pending shares and is entirely dependent on the owner's action to access their funds.

Recommendation

To mitigate this risk and provide users with an essential safety measure, it is recommended to reintroduce a `cancelRedeemRequest` function. This function would allow a user (or an approved operator) to reverse their redemption request, transferring the pending shares held by the contract back to their wallet.

You can add the following functions to `AsyncVault.sol`:

```
// ... existing code ...
        // process request
        return _requestRedeem(shares - feeShares, controller, owner);
    }

    /**
     * @notice Cancels a redeem request for the controller
     * @param controller The controller to cancel the request for
     * @param receiver The address to send the shares to
     */
    function cancelRedeemRequest(
        address controller,
        address receiver
    ) public virtual {
        return _cancelRedeemRequest(controller, receiver);
    }

    /// @dev Internal function to cancel a redeem request
    function _cancelRedeemRequest(
        address controller,
        address receiver
    ) internal virtual {
        require(
            controller == msg.sender || isOperator[controller][msg.sender],
            "ERC7540Vault/invalid-caller"
        );

        // Get the pending shares
        RequestBalance storage currentBalance = requestBalances[controller];
        uint256 shares = currentBalance.pendingShares;

        require(shares > 0, "ERC7540Vault/no-pending-request");

        // Transfer the pending shares back to the receiver
        SafeTransferLib.safeTransfer(ERC20(address(this)), receiver, shares);

        // Update the controller's requestBalance
        currentBalance.pendingShares = 0;
        currentBalance.requestTime = 0;

```

```
        emit RedeemRequestCanceled(controller, receiver, shares);
    }

// ... existing code ...
```

Status

Acknowledged

Low 03 Uninitialized Values In `beforeDeposit` Hook Calls

Location

BaseControlledAsyncRedeem.sol:52

BaseControlledAsyncRedeem.sol:86

Description

In the `BaseControlledAsyncRedeem` contract, the `deposit` and `mint` functions both call a `beforeDeposit` virtual hook. This hook is intended to allow inheriting contracts to execute custom logic before a deposit or mint operation proceeds.

The vulnerability lies in the order of operations within these functions.

1. In the `deposit` function, the `beforeDeposit(assets, shares)` hook is called before the `shares` variable is calculated. As a result, the hook is always executed with `shares` being zero.
2. Similarly, in the `mint` function, the `beforeDeposit(assets, shares)` hook is called before the `assets` variable is calculated, meaning the hook is always executed with `assets` being zero.

While this does not cause issues in the current implementation because the only override of `beforeDeposit` in `AsyncVault` does not use the values, it creates a latent bug. Any future contract that inherits from `BaseControlledAsyncRedeem` and implements logic within the `beforeDeposit` hook would operate on incorrect data, potentially leading to failed checks, incorrect state changes, or other unintended behaviors.

Recommendation

To ensure the `beforeDeposit` hook provides complete and correct information to inheriting contracts, the variable calculations should be performed before the hook is called.

Apply the following changes to `BaseControlledAsyncRedeem.sol`:

```
// ... existing code ...
    function deposit(
        uint256 assets,
        address receiver
    ) public override whenNotPaused returns (uint256 shares) {
        // Check for rounding error since we round down in previewDeposit.
        require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

        // Additional logic for inheriting contracts
        beforeDeposit(assets, shares);

        // Need to transfer before minting or ERC777s could reenter.
        SafeTransferLib.safeTransferFrom(
            asset,
// ... existing code ...
    function mint(
        uint256 shares,
        address receiver
    ) public override whenNotPaused returns (uint256 assets) {
        require(shares != 0, "ZERO_SHARES");

        assets = previewMint(shares); // No need to check for rounding error, previewMint rounds up.

        // Additional logic for inheriting contracts
        beforeDeposit(assets, shares);

        // Need to transfer before minting or ERC777s could reenter.
        SafeTransferLib.safeTransferFrom(
            asset,
// ... existing code ...
```

Alternatively document this behaviour so future developers are aware of this situation in the code.

Status

Acknowledged

Low 04 PAUSER_ROLE Can Unpause Contract

Location

BaseERC7540.sol:98

Description

In the `BaseERC7540` contract, the `unpause()` function is intended to resume contract operations after they have been halted. The function's NatSpec comment explicitly states, `Caller must be owner`, indicating that only the contract owner should have the authority to unpause.

However, the function's implementation uses the `onlyRoleOrOwner(PAUSER_ROLE)` modifier. This allows any account assigned the `PAUSER_ROLE`, in addition to the owner, to call `unpause()`. This discrepancy between the documented intent and the actual implementation creates a security risk. If an account with `PAUSER_ROLE` is compromised, an attacker could prematurely resume contract operations against the owner's wishes. This could re-expose the system to a vulnerability that the pause was meant to contain, undermining the effectiveness of the emergency stop mechanism.

Recommendation

To enhance security and align the implementation with the documented design, the access control for the `unpause()` function should be restricted to only the contract owner.

```
// ... existing code ...

    /// @notice Unpause Deposits. Caller must be owner
    function unpause() external override onlyOwner {
        _unpause();
    }
```

```

// ... existing code ...
```

Status

Fixed

Low 05 Assets Rounding Down To Zero Can Cause A Situation With Zero Pending Assets But Non Zero Pending Shares

Location

BaseControlledAsyncRedeem.sol:401

Description

In the `_requestRedeem` function of the `BaseControlledAsyncRedeem` contract, the amount of `assets` corresponding to the `shares` being redeemed is calculated using `convertToAssets(shares)`. This conversion involves division and may round down.

The vulnerability occurs when a user requests to redeem a very small number of shares. Due to the downward rounding, the calculated `assets` value can become zero. When this happens, the user's shares are transferred to the vault and their `pendingShares` balance is increased, but the corresponding `pendingAssets` is not. This leads to a state where the user has locked their shares in a redemption request but this request cannot be fulfilled.

While the inheriting `AsyncVault` contract mitigates this issue by enforcing a minimum redemption amount, the base `BaseControlledAsyncRedeem` contract does not have this protection. This creates a latent bug for any future contracts that might inherit from it without implementing a similar minimum amount check.

Proof of Concept

Add the following test to the `BaseControlledAsyncRedeem.t.sol` file:

```
/// bridgeMint inflates totalSupply so convertToAssets(smallShare) == 0
/// => pendingShares > 0 && pendingAssets == 0 and fulfillRedeem reverts.
function test_Revert_FulfillRedeem_WhenPendingAssetsZero() public {
    // Sanity: Alice already deposited INITIAL_DEPOSIT in setUp()

    // Grant BRIDGE_ROLE to owner and mint a huge amount of shares to inflate totalSupply
    vm.startPrank(owner);
    baseVault.updateRole(baseVault.BRIDGE_ROLE(), owner, true);
    vm.stopPrank();

    uint256 hugeMint = 1e30; // big number to make totalSupply > totalAssets
    vm.prank(owner);
    baseVault.bridgeMint(owner, hugeMint);

    // Ensure totalSupply is much larger than totalAssets
    assertTrue(baseVault.totalSupply() > baseVault.totalAssets());

    // Alice requests a very small redeem (1 wei of shares)
    uint256 tinyShares = 1;
    vm.startPrank(alice);
    baseVault.approve(address(baseVault), tinyShares);
    baseVault.requestRedeem(tinyShares, alice, alice);
    vm.stopPrank();

    // After request, pendingShares should be 1 but pendingAssets should be 0 due to rounding
    RequestBalance memory rb = baseVault.getRequestBalance(alice);
    assertEq(rb.pendingShares, tinyShares, "pendingShares mismatch");
    assertEq(
        rb.pendingAssets,
        0,
        "expected pendingAssets == 0 (rounding to zero) - demonstrates bug"
    );

    // Owner trying to fulfill the tiny request will compute assets == 0 and revert in _fulfillRedeem
    vm.prank(owner);
    vm.expectRevert("ZERO_SHARES");
    baseVault.fulfillRedeem(tinyShares, alice);

    // Ensure request balance unchanged after revert
    RequestBalance memory rbAfter = baseVault.getRequestBalance(alice)
```

```
    ;
    assertEq(rbAfter.pendingShares, tinyShares, "pendingShares should
    remain");
    assertEq(rbAfter.pendingAssets, 0, "pendingAssets should remain
    zero");
}
```

Recommendation

To prevent this issue at the base contract level, a check should be added to the `_requestRedeem` function to ensure that the calculated `assets` are greater than zero before proceeding with the redemption request.

```
// ... existing code ...

        uint256 assets = convertToAssets(shares);
        require(assets > 0, "ZERO_ASSETS");

// ... existing code ...
```

Status

Fixed

Low 06 Incompatibility With Standard Burn-Mint Token Mechanism

Location

BaseControlledAsyncRedeem.sol

Description

The `BaseControlledAsyncRedeem` contract implements `bridgeMint` and `bridgeBurn` functions intended to allow a trusted bridge role to mint and burn vault shares, facilitating cross-chain functionality. However, these functions do not adhere to the function signature requirements specified by Chainlink's Cross-Chain Interoperability Protocol (CCIP) for the standard "Burn-Mint" token mechanism.

According to the Chainlink documentation, a CCIP-compliant Burn-Mint token must implement:

- `mint(address account, uint256 amount)`
- `burn(uint256 amount)` or `burnFrom(address account, uint256 amount)`

The contract provides `bridgeMint(address to, uint256 shares)` and `bridgeBurn(address from, uint256 shares)`. These non-standard signatures mean that Chainlink's standard `BurnMintTokenPool` contract will be unable to interact with the vault, as its attempts to call the expected `mint` and `burn` functions will fail. This forces the project to develop a custom token pool contract to accommodate the unique function names and parameters. This will mean an additional contract which will increase the complexity of the protocol.

Specifications can be found here

Recommendation

To ensure compatibility with Chainlink CCIP's standard `BurnMintTokenPool` and simplify cross-chain integration, it is recommended to refactor the `bridgeMint` and `bridgeBurn` functions to match the required signatures.

```
// ... existing code ...
    bytes32 public constant BRIDGE_ROLE = keccak256("BRIDGE_ROLE");

    function mint(
        address account,
        uint256 amount
    ) external {
        require(hasRole[BRIDGE_ROLE][msg.sender], "BaseERC7540/not-authorized");
        require(account != address(this), "BaseERC7540/invalid-account");

        _mint(account, amount);
    }

    function burnFrom(
        address account,
        uint256 amount
    ) external {
        require(hasRole[BRIDGE_ROLE][msg.sender], "BaseERC7540/not-authorized");
        require(account != address(this), "BaseERC7540/invalid-account");
```

```

        _burn(account, amount);
    }
// ... existing code ...
```

Status

Acknowledged

Response: Custom pool will be created to handle the difference in signatures.

Low 07 Bridging Vault Owned Shares Will Lead To Loss Of Funds Or Corrupted Share Accounting

Location

BaseControlledAsyncRedeem.sol

Description

The `BaseControlledAsyncRedeem` contract includes `bridgeMint` and `bridgeBurn` functions, which allow a privileged `BRIDGE_ROLE` to create and destroy vault shares for cross-chain operations. The contract itself (`address(this)`) acts as a temporary custodian for shares that users have submitted for redemption/withdrawal via the `requestRedeem` function. These shares are held in a pending/claimable state until the redemption/withdrawal is finalized.

The vulnerability is that the `bridgeMint` and `bridgeBurn` functions do not prevent the vault's own address from being used as the target (`to`) or source (`from`). This creates a situation of a manipulation of the shares held by the vault.

1. Minting to the vault: Calling `bridgeMint(address(this), amount)` would create new shares and assign them to the vault. These shares are not tied to any user's request, which corrupts the internal accounting.
2. Burning from the vault: Calling `bridgeBurn(address(this), amount)` would destroy shares that belong to users awaiting redemption/withdrawal. This will lead to a direct loss of user funds as their shares are burned without them receiving the corresponding assets.

Recommendation

To protect the integrity of the asynchronous redemption process and prevent the loss of user funds, the `bridgeMint` and `bridgeBurn` functions should be modified to explicitly disallow the vault's contract address from being used as the destination for minting or the source for burning.

```
// ... existing code ...
    function bridgeMint(
        address to,
        uint256 shares
    ) external {
        require(hasRole[BRIDGE_ROLE][msg.sender], "BaseERC7540/not-authorized");
        require(to != address(this), "BaseERC7540/invalid-account");

        _mint(to, shares);
    }

    function bridgeBurn(
        address from,
        uint256 shares
    ) external {
        require(hasRole[BRIDGE_ROLE][msg.sender], "BaseERC7540/not-authorized");
        require(from != address(this), "BaseERC7540/invalid-account");

        _burn(from, shares);
    }
// ... existing code ...
```

Status

Acknowledged

Low 08 Multi-chain Share Value Arbitrage

Location

BaseControlledAsyncRedeem.sol

AsyncVault.sol

Description

The `BaseControlledAsyncRedeem` and `AsyncVault` contracts provide `bridgeMint` and `bridgeBurn` functions to enable cross-chain transfers of vault shares. The value of each share is determined by the formula `totalAssets() / totalSupply()`. In a multi-chain deployment, the `totalAssets()` held by the vault on each chain can diverge. This leads to the same vault share having different underlying asset values on different chains.

This discrepancy creates an economic arbitrage opportunity. A malicious user can exploit this by:

1. Identifying two chains where the vault share has a different price.
2. Burning their shares on the chain where the share price is lower.
3. Using a bridge to mint the same number of shares on the chain where the share price is higher.
4. Redeeming these newly minted shares for a larger amount of the underlying asset than they were worth on the original chain.

This effectively allows the user to drain value from the vault on the destination chain, causing a loss for the other liquidity providers in that specific deployment. While the `OracleVault` implementation mitigates this risk by using a price oracle to determine `totalAssets`, which helps standardize the share price, the fundamental vulnerability exists in the base contracts.

Recommendation

This behavior should be explicitly documented in the `BaseControlledAsyncRedeem` contract to warn future developers and integrators of the potential risk. The NatSpec comments for the bridging functions should highlight the danger of share price arbitrage in multi-chain environments and recommend implementing a mechanism to ensure value consistency.

Status

Fixed

Response: Bridging functionality was moved to OracleVault where there is no such risk.

Low 09 Missing Emergency Switch For Oracle And Safe In `OracleVault`

Location

OracleVault.sol

Description

The `OracleVault` contract implements a two-step, time-locked process for changing its critical `safe` and `oracle` addresses. An owner must first `propose` a new address and then wait for a mandatory 3-day delay before they can `accept` the change. While this time-lock is a valuable security feature for planned administrative changes, as it gives users time to review and react, it is dangerously slow in an emergency.

The vulnerability lies in the absence of a mechanism for immediate, emergency intervention. If the `oracle` contract becomes compromised and starts providing malicious prices, or if it consistently reverts (causing a denial-of-service for the `totalAssets` function), the vault's operations would be severely impacted. Similarly, if the signers for the `safe` are compromised, all assets held by the vault are at immediate risk of theft. In these critical scenarios, waiting 3 days is not a viable option and would likely lead to significant financial loss for the vault and its users.

Recommendation

To address this, the contract should include emergency functions that allow a privileged role (e.g., a new emergency role) to bypass the time-lock and change the `safe` or `oracle` address instantly. This provides a necessary "fast track" to mitigate critical threats, such as switching to a new secure safe or a fallback oracle.

Status

Acknowledged

Low 10 Missing Proposal Expiry For Oracle And Safe

Location

OracleVault.sol

Description

The `OracleVault` contract allows the owner to change the `safe` and `oracle` addresses through a two-step process: first proposing a new address and then accepting it after a 3-day time delay. This is implemented via the `proposeSafe` /`acceptSafe` and `proposeOracle`/`acceptOracle` function pairs.

The vulnerability is that these proposals, once made, never expire. A proposal for a new safe or oracle can be accepted weeks, months, or even years after it was initially made. This creates a risk of stale proposals being accepted long after they are relevant. For example, an owner might propose a new address, but then decide against it for some reason. If the proposal is not explicitly overwritten, it remains pending indefinitely and could be accidentally accepted later, leading to the vault being configured with an unintended or outdated address. This could cause operational disruption or security issues if the old proposed address is no longer secure.

Recommendation

To mitigate this risk, a fixed expiration period should be added to all proposals. When accepting a proposal, the contract should check not only that the minimum delay has passed but also that the proposal has not expired. This ensures that only recent, relevant proposals can be activated.

Status

Fixed