Withdrawable OFT Adapter Audit Report

Prepared by: InAllHonesty

Table of Contents

Protocol Summary

The Withdrawable OFT Adapter is a modified LayerZero OFTAdapter implementation that enables permissioned withdrawal of locked funds for migration or recovery purposes, addressing vendor lock-in concerns with LayerZero's default lockbox system. The protocol allows projects to opt-out of the OFT system and migrate to alternative cross-chain solutions like Chainlink CCIP, while maintaining standard OFT functionality with additional emergency withdrawal capabilities and LayerZero disabling mechanisms.

Disclaimer

We made every effort to find as many vulnerabilities in the code in the given time period of 72 hours. We hold no responsibilities for the findings provided in this document. A security audit 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 Solidity implementation of the contracts.

Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

High Impact:

- Funds are directly or nearly directly at risk.
- There's a severe disruption of protocol functionality or availability.

Medium Impact:

- Funds are indirectly at risk.
- There's some level of disruption to the protocol's functionality or availability.

Low Impact:

- Funds are not at risk.
- However, a function might be incorrect, state might not be handled appropriately, etc.

How to evaluate the Likelihood of exploitation

Likelihood represents the probability of the impact occurring due to the vulnerability.

High Likelihood:

Highly probable to happen. For instance, a hacker can call a function directly and extract money.

Medium Likelihood:

It might occur under specific conditions. For example, a peculiar ERC20 token is used on the platform.

Low Likelihood:

Unlikely to occur. An example might be if a hard-to-change variable is set to a unique value on a very specific block.

Audit Details

The findings described below correspond to the following commit hash:

```
b5f627e22c83fd5d215b8c83cd9d0a76c5d50df4
```

Scope

The following contracts:

- DisableableOFT.sol
- DisableableOFTAdapter.sol
- DisableableOFTCore.sol
- WithdrawableOFTAdapter.sol

Executive Summary

The audit was carried out over 72 hours. The protocol was checked for vulnerabilities using manual review.

Issues found

Severity	**Number of issues found**
High	0
Medium	1
Low	1
Informational	1
Gas	0

Findings

[M-1] Emergency withdrawal can be permanently blocked by user withdrawals during delay period

Description

The emergency withdrawal mechanism uses a fixed amount set at initialization, but allows users to continue withdrawing during the delay period. This creates a scenario where `WithdrawableOFTAdapter::emergencyWithdraw()` will revert if the contract balance falls below `WithdrawableOFTAdapter::s_pendingWithdrawAmount`.

When the owner calls `WithdrawableOFTAdapter::initializeEmergencyWithdraw(address _to, uint256 _amount)`, the `_amount` is fixed. During the `WithdrawableOFTAdapter::i_emergencyWithdrawDelay` period, users can still bridge their tokens out, reducing the contract balance. If the balance drops below the requested amount, `WithdrawableOFTAdapter::emergencyWithdraw()` will revert with insufficient balance.
This forces the owner into a cycle of:

- Cancel the current withdrawal
- Re-initialize with a lower amount
- Wait another full delay period
- Risk the balance dropping again

This can happen through normal user activity (users legitimately bridging their tokens when they see an emergency withdrawal initiated).

To prevent this, the owner could disable send and receive functionality, but this would undermine the purpose of having an emergency delay period.

Risk

Likelihood: Medium

- Any emergency withdrawal announcement will likely trigger users to bridge their tokens out as a precautionary measure
- This is expected user behavior when they see emergency functions being activated
- Even without malicious intent, normal user activity can cause this issue

Impact: Medium

- No permanent loss of funds
- Owner can eventually withdraw remaining balance after multiple attempts
- However, this creates significant operational friction and delays in emergency scenarios and could prevent timely migration or emergency response

Recommended Mitigation

Modify `WithdrawableOFTAdapter::emergencyWithdraw()` to withdraw the minimum of the requested amount and current balance:

Add `import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";` in the import section of `WithdrawableOFTAdapter` contract.

```
    function emergencyWithdraw() external onlyOwner {
        if (
            s_emergencyWithdrawInitializedAt == 0
                || block.timestamp <= s_emergencyWithdrawInitializedAt +
i_emergencyWithdrawDelay
        ) {
            revert EmergencyWithdrawDelayNotPassed();
        }
        address to = s_pendingWithdrawTo;
--      uint256 amount = s_pendingWithdrawAmount;
++      uint256 requestedAmount = s_pendingWithdrawAmount;
++      uint256 actualAmount = Math.min(requestedAmount,
innerToken.balanceOf(address(this)));

        s_pendingWithdrawTo = address(0);
        s_pendingWithdrawAmount = 0;
        s_emergencyWithdrawInitializedAt = 0;

--      emit TokensWithdrawn(to, amount);
--      innerToken.safeTransfer(to, amount);

++      emit TokensWithdrawn(to, actualAmount);
++      innerToken.safeTransfer(to, actualAmount);

    }
```

This ensures the withdrawal always succeeds with whatever balance remains, preventing accidental griefing while maintaining the delay period's user protection intent.

[L-1] `WithdrawableOFTAdapter` is not fully compatible with rebasing tokens

Description

The emergency withdrawal mechanism records a fixed `WithdrawableOFTAdapter::s_pendingWithdrawAmount` at initialization time, but rebasing tokens can change the contract's balance during the delay period without any transfers occurring.

Positive Rebasing Impact:
When the token supply increases (positive rebase), the contract's balance grows beyond the recorded amount. After withdrawal, the excess tokens remain permanently locked in the contract with no recovery mechanism. Each subsequent withdrawal attempt leaves additional tokens stuck.

Negative Rebasing Impact:
When the token supply decreases (negative rebase), the contract's balance may fall below `WithdrawableOFTAdapter::s_pendingWithdrawAmount`. This causes `WithdrawableOFTAdapter::emergencyWithdraw()` to revert due to insufficient balance, forcing the owner to cancel and restart the entire delay period with a lower amount. Multiple negative rebases during successive attempts could make withdrawal impossible.

Risk

Likelihood: Low

- Requires the owner to set a reasonably high `i_emergencyWithdrawDelay`
- Requires a rebasing token

Impact: Low

- Positive rebase: Tokens stuck but contract remains functional
- Negative rebase: Withdrawal delayed but eventually possible with reduced amount
- No permanent fund loss, but operational inefficiency

Recommended Mitigation

- Document the behavior
- Adjust the current functions providing the choice between withdraw modes, one for a fixed amount (existing), and one for a full balance where the contract balance computed inside `emergencyWithdraw` is withdrawn. Thus the owner would use the full balance mode in case of rebasing tokens.

Example implementation:

```
//WithdrawableOFTAdapter.sol

++enum WithdrawMode { FIXED_AMOUNT, FULL_BALANCE }

++WithdrawMode s_withdrawMode;

--    function initializeEmergencyWithdraw(address _to, uint256 _amount)
external onlyOwner {
++    function initializeEmergencyWithdraw(address _to, uint256 _amount,
WithdrawMode _withdrawMode) external onlyOwner {
    // [...]
++      s_withdrawMode = _withdrawMode;
    // [...]
    }

    function emergencyWithdraw() external onlyOwner {
        // [...]
++      uint256 amount;
++        if (s_withdrawMode == WithdrawMode.FULL_BALANCE) {
++            amount = innerToken.balanceOf(address(this));
++      } else {
++          amount = s_pendingWithdrawAmount;
        }
        // [...]
    }
```

[I-1] Stuck transfers possible if `lzReceive` is disabled during delivery

Description

Cross-chain token transfers via LayerZero are asynchronous. When a user initiates a transfer, tokens are immediately locked/burned on the source chain, and a message is sent to the destination chain to unlock/mint the tokens. This creates a time window where tokens exist in neither chain - they're in transit.

If the owner calls `DisableableOFTCore::setOFTReceive(false)` while messages are in-flight, the `lzReceive` function will revert with `DisableableOFTCore::OFTReceiveDisabled` when the LayerZero endpoint attempts delivery. The endpoint then stores the failed message payload, leaving tokens in limbo.

As a result, the LayerZero endpoint marks the payload as failed and stores it.
Tokens are no longer available on the source chain but are also not credited on the destination chain until the message is retried.
Without retry, funds remain stuck indefinitely.

Risk

Likelihood: Low

- Requires owner action at the precise moment messages are in transit
- Most likely during emergency response or migration scenarios

Impact: Info

- No permanent loss of funds (message can be retried)
- However, users experience stuck transfers and poor UX

Recommended Mitigation

- Document the recovery flow.