Native OFT Adapter

Security Assessment

September 19th, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Table of Contents

Appendices

© 2024 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Layerzero engaged OtterSec to assess the `nativeOFTAdapter` program. This assessment was conducted between August 28th and August 30th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We made a recommendation to optimize staking efficiency by restaking the entire remaining balance of `ApeCoin` after handling token crediting and withdrawals (OS-OFT-SUG-00), and advised to provide proper documentation for certain fee handling functions which are stubs, to prevent misunderstandings when extending the contract (OS-OFT-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/devtools. This audit was performed against PR#838.

A brief description of the programs is as follows:

Name	Description
nativeOFTAdapter	It extends the functionality of OFTCore (Omnichain Fungible Token) to handle native currency transfers between chains utilizing LayerZero.

© 2024 Otter Audits LLC. All Rights Reserved.

02 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-OFT-SUG-00	Replace `_removeDust` with `_debitView` in `send` as it provides a more semantically correct way to determine the required value.
OS-OFT-SUG-01	In `NativeOFTAdapter`, `_debit` and `_payNative` are stubbed out, rendering it difficult to extend and insecure if these functions are called elsewhere without proper fee handling.

© 2024 Otter Audits LLC. All Rights Reserved.

Improving Debit Calculation Semantics OS-OFT-SUG-00

Description

For improved semantic correctness within `NativeOFTAdapter` it is appropriate to utilize `_debitView` instead of `_removeDust` in `send` because `_debitView` better represents the intended operation of calculating the exact amount of value (in native tokens) that needs to be debited.

>_ packages/oft-evm/contracts/NativeOFTAdapter.sol SOLIDITY

```
function send(
    SendParam calldata _sendParam,
    MessagingFee calldata _fee,
    address _refundAddress
    ) public payable virtual override returns (MessagingReceipt memory msgReceipt, OFTReceipt
        ↪  memory oftReceipt)
{
    // @dev Ensure the native funds in msg.value are exactly enough to cover the fees and amount
        ↪  to send(with dust removed).
    // @dev This will revert if the _sendParam.amountLD contains any dust
    uint256 requiredMsgValue = _fee.nativeFee + _removeDust(_sendParam.amountLD);
    if (msg.value != requiredMsgValue) {
        revert IncorrectMessageValue(msg.value, requiredMsgValue);
    }
    return _send(_sendParam, _fee, _refundAddress);
}
```

Remediation

Utilize `_debitView` instead of `_removeDust` in `send`.

© 2024 Otter Audits LLC. All Rights Reserved.

Code Maturity OS-OFT-SUG-01

Description

In the current implementation of `NativeOFTAdapter`, `_payNative` and `_debit` are overridden in a somewhat minimalistic way and lack any significant functionality, which may be misleading, potentially resulting in fragility and insecurity in the contract, particularly if it is extended or utilized in future upgrades. `_debit` is overridden in a way that it does not actually handle the fees nor directly manage the locking of native tokens, returning only the results of `_debitView`. Similarly, `_payNative` is also stubbed out, where it simply returns the `_nativeFee` parameter without actually performing any operation to pay the fee. Thus, there is a risk of incorrect accounting and fee handling.

>_ packages/oft-evm/contracts/NativeOFTAdapter.sol SOLIDITY

```
function _debit(
    address /*_from*/,
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 _dstEid
) internal virtual override returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    // @dev Native funds sent with msg.value are locked into this contract higher up on the
        ↪  overriddensend() function
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
}

function _payNative(uint256 _nativeFee) internal pure override returns (uint256 nativeFee) {
    return _nativeFee;
}
```

Remediation

Make it clear in the documentation that `_debit` and `_payNative` are stubs, and no actual handling of native tokens and fees is done in them. This will help prevent misunderstandings when extending the contract.

© 2024 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2024 Otter Audits LLC. All Rights Reserved.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2024 Otter Audits LLC. All Rights Reserved.