Orbit Adapter

Security Assessment

June 27th, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Layerzero engaged OtterSec to assess the `orbit-adapter` programs. This assessment was conducted between June 3rd and June 23rd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We recommended isolating the handling of native tokens (`ETH`) to manage native fee semantics independently, thereby avoiding potential conflicts with future changes (OS-OAP-SUG-00). Additionally, we suggested adding a check in the constructor of `OrbitERC20OFTAdapter` to ensure proper setup for handling the `innerToken` (OS-OAP-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/orbit-adapter. This audit was performed against commit b20f782.

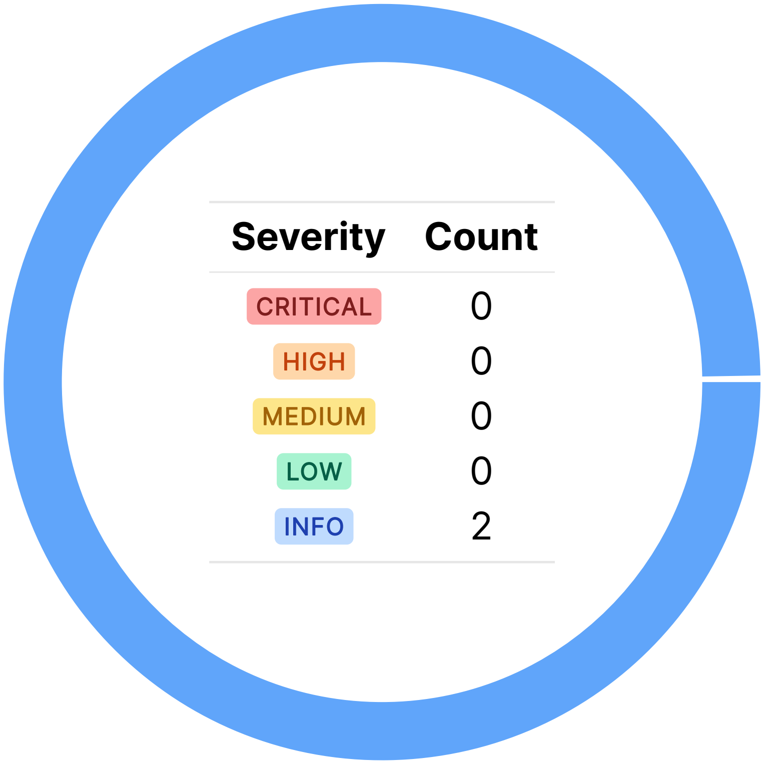
A brief description of the programs is as follows:

Name	Description
orbit-adapter	Integrates the Omnichain Fungible Token (OFT) directly into the existing Arbitrum native bridge, allowing seamless transactions between Layer 3 (L3) networks and Arbitrum (L2). This integration enables users to move assets utilizing either the Arbitrum native bridge or the OFT interchangeably.

03 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	0
INFO	2

04 — 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-OAP-SUG-00	`send` in `NativeOFTAdapterMsgValueTransfer` alters the semantics of `msg.value`, which may pose a security risk if other parts of the contract rely on `msg.value` to validate fees.
OS-OAP-SUG-01	Suggestion to ensure that each `OrbitERC20OFTAdapter` is correctly configured with its corresponding bridge.

Native Fee Handling Risk

OS-OAP-SUG-00

Description

In the current implementation of `send`, `msg.value` is utilized to cover the difference when the sender's token balance is insufficient to meet the required transfer amount. This utilization of `msg.value` is intertwined with the minting of additional tokens and the adjustment of the messaging fee. If other parts of the contract or future upgrades assume that `msg.value` is only utilized for fees and not for minting tokens, it may result in inconsistencies.

>_ *vsrc/L3/NativeOFTAdapterMsgValueTransfer.sol* solidity

```solidity
function send(
    SendParam calldata _sendParam,
    MessagingFee calldata _fee,
    address _refundAddress
) external payable override returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
    ↪  oftReceipt) {
    [...]
    if (msgSenderBalance < _sendParam.amountLD) {
        require(msgSenderBalance + msg.value >= _sendParam.amountLD,
            ↪  "NativeOFTAdapterMsgValueTransfer: Insufficient msg.value");

        // user can cover difference with additional msg.value ie. wrapping
        uint mintAmount = _sendParam.amountLD - msgSenderBalance;
        _mint(address(msg.sender), mintAmount);

        // update the messageFee to take out mintAmount
        feeWithExtraAmount.nativeFee = msg.value - mintAmount;
    }
    [...]
}
```

Remediation

Utilize `this.call` to handle the native fee semantics more explicitly. This approach keeps the handling of native tokens (`ETH`) more isolated and avoids potential conflicts with future changes.

Bridge-Token Configuration Verification OS-OAP-SUG-01

Description

Each `OrbitERC20OFTAdapter` is associated with a specific `ERC-20` token (referred to as `innerToken` in the `OFTAdapter`). This token must be correctly managed by the bridge to ensure it is locked, transferred, and released accurately during cross-chain operations. By verifying that the bridge is configured with the `nativeToken` matching the `innerToken`, misconfigurations that could lead to incorrect token transfers or loss of tokens are prevented. This verification ensures that each adapter works with its designated bridge, maintaining the integrity and security of the token transfer operations.

Remediation

Add a check in the constructor of `OrbitERC20OFTAdapter` to ensure that the bridge is correctly set up to handle the `innerToken`.

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.

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 the 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.