LayerZero OVault

Security Assessment

Nicholas R. Putra   nicholas@osec.io

Zhenghang Xiao   kiprey@osec.io

Table of Contents

01 — Executive Summary

Overview

LayerZero engaged OtterSec to assess the `ovault` program. This assessment was conducted between July 14th and July 15th, 2025. We further conducted a follow-up audit between October 20th and October 21st, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 1 finding throughout this audit engagement.

We made a recommendation to ensure the correct functionality is utilized and to provide a more appropriate error message for improved clarity (OS-OVT-SUG-00).

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#1584. The follow-up audit was performed against 7610a16 and PR#1778.

A brief description of the program is as follows:

Name	Description
ovault	It defines a cross-chain ERC4626-compatible vault system utilizing LayerZero, allowing users to deposit or redeem assets on a hub chain and receive tokens on a target chain

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-OVT-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Code Maturity   OS-OVT-SUG-00

Description

1. The current compose function incorrectly reverts with `InsufficientMsgValue` when a `msg.value` is sent to the hub chain, even though the issue does not concern underpayment. Utilizing a more specific error will clarify intent and reduce confusion during debugging.

2. In `quoteSend`, since `_oft` is set to `ASSET_OFT`, the function should utilize `previewRedeem` instead of `previewDeposit`.

3. The current slippage check does not account for potential slippage due to dust removal in the `OFT`, as `minAmountLD` is nullified after the slippage check is performed.

4. The comment in the `VaultComposerSyncNative` stated that the `WETH` is a share token, but the correct comment is that it is an asset token.

Patch

1. Issue #1 and #2 were addressed in Pr#1639.

2. The LayerZero team acknowledged Issue #3, stating that slippage is correctly applied to the vault action but is ignored for the `OFT` transfer.

3. Issue #4 was addressed in PR#1766.

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 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.