LayerZero LZReceiveTypes

Security Assessment

Ajay Shankar Kunapareddy   d1r3wolf@osec.io

Xiang Yin   soreatu@osec.io

Robert Chen   r@osec.io

Table of Contents

01 — Executive Summary

Overview

LayerZero engaged OtterSec to assess the `lzReceiveTypes-V2` program. This assessment was conducted between July 1st and July 3rd, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 1 finding throughout this audit engagement.

In particular, we suggested the need to adhere to coding best practices (`OS-LLZ-SUG-00`).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/LayerZero-v2. This audit was performed against commit c09287a.

A brief description of the program is as follows:

Name	Description
lzReceiveTypes-V2	The scope focused on validating the integration of pre-execute and post-execute hooks in the executor program and ensuring secure, flexible metadata handling for off-chain orchestration.

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

Code Maturity   OS-LLZ-SUG-00

Description

1. The current fee validation logic utilizes unchecked addition (b `alance_after + fee_limit`), which may overflow if `fee_limit` is very large (`u64::MAX`), resulting in the check to fail. Utilize `saturating_add` to ensure the result caps at `u64::MAX` , avoiding overflow.

```solidity
>_  programs/executor/src/instructions/post_execute.rs                    SOLIDITY

/// Validates fee limits and executor account state.
fn validate_fee_limits_and_executor_state(
    executor: &Signer,
    context: &Account<ExecutionContextV1>,
) -> Result<()> {
    // Validate fee limits
    let balance_after = executor.lamports();
    let balance_before = context.initial_payer_balance;
    let fee_limit = context.fee_limit;
    require!(balance_before <= balance_after + fee_limit,
        ↪  ExecutorError::InsufficientBalance);
    [...]
}
```

2. The current `PostExecute` instruction does not verify that it utilizes the same context account initialized in `PreExecute` . To prevent misuse, it should be ensured both instructions reference the same PDA-derived context account.

3. In `get_accounts_for_clear` , the `nonce_account` is marked as writable, but the `clear` instruction only reads from it for nonce validation and does not modify it. Therefore, the `is_writable` flag is unnecessary and should be set to false.

Remediation

Implement the above-mentioned suggestions.

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.