LayerZero Examples

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Shiva Shankar sh1v@osec.io

Contents

01 Executive Summary **2**

Overview . . . 2

Key Findings . . . 2

02 Scope **3**

03 Findings **4**

04 General Findings **5**

OS-LZR-SUG-00 | Reorganize Payload Fields . . . 6

OS-LZR-SUG-01 | Fee Constraints Edge Case . . . 7

Appendices

A Vulnerability Rating Scale **8**

B Procedure **9**

01 | Executive Summary

Overview

LayerZero engaged OtterSec to perform an assessment of various LayerZero programs under our retainer. This is an ongoing engagement, starting November 11th. For more information on our auditing methodology, see Appendix B.

As part of this engagement, we reviewed LayerZero's OFT v2 upgrade.

Key Findings

Over the course of this audit engagement, we produced 2 findings total.

In particular, we found a minor edge case with fee constraint validation (OS-LZR-SUG-01). Additionally, we made recommendations around reorganizing payload fields (OS-LZR-SUG-00).

Overall, we commend the LayerZero team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/LayerZero-Labs/solidity-examples . This audit was performed against commit `9d9bf12`.

A brief description of the programs is as follows.

Name	Description
solidity-examples	Solidity code examples building on top of LayerZero.

03 | Findings

Overall, we report 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 don't have an immediate impact but will help mitigate future vulnerabilities.

Severity	Count
Critical	0
High	0
Medium	0
Low	0
Informational	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 antipatterns and could lead to security issues in the future.

ID	Description
OS-LZR-SUG-00	Reorder OFT payload fields to mitigate the impact of variable length decoding.
OS-LZR-SUG-01	Potential edge case for fee constraints

OS-LZR-SUG-00 | Reorganize Payload Fields

Description

Currently, the OFT payload is encoded via

```solidity
          return abi.encodePacked(
              PT_SEND_AND_CALL,
              _toAddress,
              _amountSD,
              _addressToBytes32(_from),
              _dstGasForCall,
              _payload
          );tc
```

Note that the security of this encoding depends on the correct size of `_toAddress`. Otherwise, the critical field `_amountSD` could get confused with bytes of the `_from` field.

Remediation

As a defense-in-depth measure, it could make sense to reorder the fields. More specifically, putting `_amountSD` first in the encoding will likely mitigate future variable length encoding attacks.

OS-LZR-SUG-01 | Fee Constraints Edge Case

Description

In the `Fee.sol` contract, while setting the `defaultFeeBp` via `setDefaultFeeBp` and `setFeeBp` it should ensure that the fee is less than and **not equal to** the `BP_DENOMINATOR`. If the `defaultFeeBp` is equal to the `BP_DENOMIMATOR` then the transactions will fail.

Remediation

Update the check to ensure that the `_feeBp` is only less than `BP_DENOMINATOR`.

Fee.sol SOLIDITY

```
   function setDefaultFeeBp(uint16 _feeBp) public virtual onlyOwner {
        require(_feeBp < BP_DENOMINATOR, "Fee: fee bp must be <
    ↪   BP_DENOMINATOR");
        defaultFeeBp = _feeBp;
        emit SetDefaultFeeBp(defaultFeeBp);
    }

    function setFeeBp(uint16 _dstChainId, bool _enabled, uint16 _feeBp)
    ↪   public virtual onlyOwner {
        require(_feeBp < BP_DENOMINATOR, "Fee: fee bp must be <
    ↪   BP_DENOMINATOR");
        chainIdToFeeBps[_dstChainId] = FeeConfig(_feeBp, _enabled);
        emit SetFeeBp(_dstChainId, _enabled, _feeBp);
    }
```

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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 other 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.