LayerZero Protocol

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Shiva Shankar sh1v@osec.io

Contents

Appendices

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 a Solidity version upgrade for Layerzero's zksync implementation. We also reviewed a multisig oracle implementation.

Key Findings

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

In particular, we found issues with failing calculations caused by checked operations due to a Solidity compiler version upgrade (OS-LZR-ADV-00). We also made suggestions around clarifying critical security invariants (OS-LZR-SUG-00).

02 | Scope

The source code was delivered to us in a git repository at https://github.com/LayerZero-Labs/layerzero-internal/. This audit was performed against commit `f1e06e3` respectively.

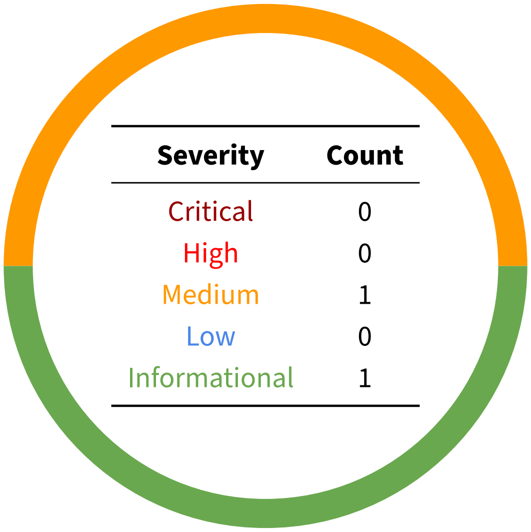
A brief description of the programs is as follows.

Name	Description
LayerZero	EVM contracts for LayerZero, an omnichain interoperability protocol.

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	1
Low	0
Informational	1

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-LZR-ADV-00	Medium	Resolved	Intended integer overflows fail due to checked math in solidity 0.8.0.

OS-LZR-ADV-00 [med] [resolved] | Intended Overflows Aborts

Description

In the solidity version 0.8.X, checked arithmetic operations are used by default. Operations that may result in integer overflows or underflows will fail. In certain circumstances however, these overflows are intended, such as when calculating integer masks.

Remediation

Use unchecked arithmetic operations for calculations when there is an intended integer overflow/underflow.

DIFF

```diff
--- a/contracts/proof/utility/Buffer.sol
+++ b/contracts/proof/utility/Buffer.sol
@@ -145,12 +145,15 @@ library Buffer {
         }

         // Copy remaining bytes
-        uint mask = 256**(32 - len) - 1;
-        assembly {
-            let srcpart := and(mload(src), not(mask))
-            let destpart := and(mload(dest), mask)
-            mstore(dest, or(destpart, srcpart))
+        unchecked {
+            uint mask = 256**(32 - len) - 1;
+            assembly {
+                let srcpart := and(mload(src), not(mask))
+                let destpart := and(mload(dest), mask)
+                mstore(dest, or(destpart, srcpart))
+            }
         }
+

         return buf;
     }
```

Patch

Resolved in f1e06e3.

05 | 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	Packed encoding is unsafe when using with multiple dynamic arguments.

OS-LZR-SUG-00 | Potentially Unsafe Encoding

Description

In the MultiSigOracle.sol contract, the hashCallData function uses abi.encodePacked to encode the call data. This might be unsafe if multiple arguments to call data are dynamic, allowing for confusion between different payloads.

MultiSigOracle.sol SOLIDITY

```
        function hashCallData(address _target, bytes calldata _callData,
      ↪  uint _expiration) public pure returns (bytes32) {
         return keccak256(abi.encodePacked(_target, _expiration,
      ↪  _callData));
    }
}
```

Remediation

Either document this behavior or change abi.encodePacked to abi.encode.

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.