Layer Zero RateLimiter Security Review

Pashov Audit Group

Conducted by: Juan, r0bert, Alex Murphy

September 18th - September 19th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **LayerZero-Labs/devtools/packages/oapp-evm** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About RateLimiter

Layer Zero RateLimiter is designed to control the flow of operations by enforcing rate limits based on a configurable limit and time window. It tracks inflight transactions, updates the available capacity over time, and reverts any action that exceeds the allowed rate for a specific destination endpoint.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 64aeeb59bb649fe7756b3c884057a76237c9e5de

fixes review commit hash - 16acbdcd0c8357be2821ea506a713bb6904d2ad8

Scope

The following smart contracts were in scope of the audit:

- `RateLimiter`

7. Executive Summary

Over the course of the security review, Juan, r0bert, Alex Murphy engaged with Layer Zero to review RateLimiter. In this period of time a total of **3** issues were uncovered.

Protocol Summary

Protocol Name	RateLimiter
Repository	https://github.com/LayerZero-Labs/devtools
Date	September 18th - September 19th
Protocol Type	Messaging protocol

Findings Count

Severity	**Amount**
Low	3
Total Findings	**3**

Summary of Findings

ID	**Title**	**Severity**	**Status**
[L-01]	In flights can get reset to zero	Low	Resolved
[L-02]	Precision loss in decay calculation	Low	Acknowledged
[L-03]	Rate limits are not applied	Low	Acknowledged

8. Findings

8.1. Low Findings

[L-01] In flights can get reset to zero

The current implementation of the RateLimiter will reset the in-flight amount to zero when `timeSinceLastDeposit` is greater than or equal to the window, this is handled in an `if` statement:

```
if (timeSinceLastDeposit >= _window) {
    currentAmountInFlight = 0;
    amountCanBeSent = _limit;
}
```

However, the code does not check if the `timeSinceLastDeposit` is enough time to have decayed the `currentAmountInFlight` to zero. To showcase the issue take the following example:

State 1:

- Window = 100
- Limit = 100
- AmountInFlight = 100

At that moment a configuration change happens and the window and limit are changed to:

- Window = 60
- Limit = 40
- AmountInFlight = 100 (remains the same)

After 60 seconds from the last update, the amount in flight should be (100 - 40) = 60, but the code will reset it to zero.

Take into consideration this scenario before resetting the `currentAmountInFlight` to zero and `amountCanBeSent` to the limit. If this is the desired behavior then the documentation should warn about this possibility.

[L-02] Precision loss in decay calculation

In the `RateLimiter` contract, the decay of `amountInFlight` overtime is calculated using the following line in the `_amountCanBeSent()` function:

```
uint256 decay = (_limit * timeSinceLastDeposit) / _window;
```

This calculation is intended to model a linear decay of the `amountInFlight` over the specified rate limiting window. However, due to Solidity's integer division, this approach suffers from precision loss, especially when the `_window` period is way higher than the `_limit` or when dealing with small `timeSinceLastDeposit` values relative to `_window`.

Suppose `_limit` = 1000, `_window` = 1000000, and `timeSinceLastDeposit` = 1.

```
decay = (1000 * 1) / 1000000 = 0
```

Here, `decay` is calculated as 0 due to integer division, even though some time has passed and there should be a fractional decay.

To address the precision loss in the `decay` calculation, it is recommended to implement fixed-point arithmetic by introducing a scaling factor.

Layer Zero comments

Tracking as ten-thousandth, even just internally, is complicated and does not provide much benefit to our intended use case. Additionally, we plan to leverage RateLimiter with existing _inflow(...) and _outflow(...) implementations which uses units of counting numbers, not ten-thousandths. As such, we see documentation as the best resolution to this issue.

LayerZero-Labs/devtools#905

[L-03] Rate limits are not applied

The `_inflow()` function is designed to reduce the `amountInFlight` by the `_amount` passed to it (`inflowAmount`), with a lower bound of zero:

```
function _inflow(uint32 _srcEid, uint256 _amount) internal virtual {
    RateLimit storage rl = rateLimits[_srcEid];

        rl.amountInFlight = _amount >= rl.amountInFlight ? 0 : rl.amountInFlight - _a
}
```

However, let's imagine the following two scenarios:

- `RateLimiter.RateLimitConfig(1337, 1000, 100);`

Scenario 1

- Initially, the system starts with 0 units in flight and 1000 units available to be sent.
- In step 1, an `outflow(500)` occurs, reducing the available units to 500, with 500 units now in flight.
- In step 2, an `inflow(1000)` clears the inflight amount, bringing it back to 0, and restoring 1000 units available to be sent.
- Finally, in step 3, another `outflow(500)` is processed, leaving 500 units in flight and 500 units that can still be sent.

Scenario 2

- The initial state has 0 units in flight and 1000 units available to send.
- In step 1, an `outflow(500)` occurs, reducing the available amount to 500, with 500 units now in flight.
- In step 2, another `outflow(500)` fully utilizes the available amount, leaving 1000 units in flight and 0 units available.
- In step 3, an `inflow(1000)` clears the inflight amount, resetting it to 0 and making 1000 units available again. Despite both scenarios involving a total of 1000 units of outflow and 1000 units of inflow processed in the same timestamp, the remaining rate limit differs based on the order of operations. This inconsistency proves that the rate limiter behaves differently when the sequence of inflow and outflow operations changes, even though the net effect is the same.

Users performing the same net actions will experience different rate limits based solely on the sequence of their operations. Moreover, it is expected that users will deliberately arrange operations to maximize their rate limits. **A rate limiter should apply limits consistently, regardless of the order of operations, as long as the net effect is the same.**

Consider updating the contract's logic to ensure that the order of operations within the same timestamp does not alter the remaining rate limit.

Layer Zero comments

Interesting. We have taken the approach that transaction ordering within a given block matters.

The current RateLimiter fixes the range of amountCanBeSent = [0, MAX]. Per timestamp (block), we could track excessive inflows in a separate variable (as there is no Inbound Rate Limit), and prioritize subtracting from that first. However, we cannot do the opposite and allow excessive outflows (as there is an Outbound Rate Limit) that might be compensated for later. The asymmetry of this solution is confusing, and costs additional storage reads/writes.

Given a sufficiently large limit, this shouldn't really matter. As such, we are choosing to just add documentation explaining the limitation.

LayerZero-Labs/devtools#904