BURRA SEC

LayerZeroV2 Security Review

Reviewed by: windhustler

LayerZeroV2 Security Review Report

Burra Security

Feb 22, 2024

Introduction

A time-boxed security review of the **LayerZeroV2** protocol was done by **Burra Security** team, focusing on the security aspects of the smart contracts.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Burra Security

Burra Sec offers security auditing and advisory services with a special focus on cross-chain and inter-operability protocols and their integrations.

About LayerZeroV2

LayerZero is an open-source, immutable messaging protocol designed to facilitate the creation of omnichain, interoperable applications.

Using smart contracts deployed on each chain, in combination with **Decentralized Verifier Networks (DVNs)** and **Executors**, LayerZero enables different blockchains to seamlessly interact with one another.

In **LayerZero V2**, message verification and execution have been separated into two distinct phases, providing developers with more control over their application's security configuration and independent execution.

Combined with **improved handling**, **message throughput**, **programmability**, and other contract specific improvements, **LayerZero V2** provides a more flexible, performant, and **future-proof messaging protocol**.

Severity classification

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

***review commit hash* - 4a522bac575d18314cb16781dd3bd97875483a2a**

No fixes implemented.

Scope

The following smart contracts were in the scope of the audit:

- `packages/layerzero-v2/evm/protocol/contracts/**`
- `packages/layerzero-v2/evm/messagelib/contracts/**`

Findings Summary

ID	Title	Severity	Status
[L-01]	Support for the `PUSH0` opcode	Low	-
[L-02]	Possible DoS attack vector with certain app designs	Low	Ack
[L-03]	Possible DoS attack vector by setting high gas limit	Low	Ack
[I-01]	Replace `Ownable` with `Ownable2Step`	Info	-
[I-02]	Follow best practices for `__gap` in upgradeable contracts	Info	-
[I-03]	Pass `guid` and `message` encoded as calldata to `lzReceive`	Info	-

Detailed Findings

[L-01] Support for the PUSH0 opcode

Context

- foundry.toml
- hardhat.config.ts

Description

On Arbitrum and Base the `PUSH0` opcode is not supported yet. Since the project is using a solidity version higher than `0.8.20` it can only be used with evm version lower than the default `shanghai`, e.g. `paris`.

Make sure to check the support for the aforementioned opcode on all the chains you are planning to deploy the contracts on.

Here are the instructions to set evm version configuration parameter with `Foundry` and `Hardhat`.

Recommendation

One of the options is to add the following to the `foundry.toml` file:

```
evm_version = "paris"   # to prevent usage of PUSH0, which is not
    supported on all chains
```

Discussion

No comments yet.

[L-02] Possible DoS attack vector with certain app designs

Context

- EndpointV2.sol

Description

The design of `EndpointV2` separates the delivery and execution of messages on the receiving chain. On the sending side, the user is paying for his transaction to be delivered with a specific gas limit and `msg.value`. The design is such that after the message is delivered the `EndpointV2:lzReceive()` can be invoked by anyone. It is the responsibility of the app to check the sender and other parameters that are encoded in the message.

Let's imagine the following application design:

- An application allows the user to lock in tokenA on ChainA and send a message through LayerZero to ChainB with encoded data to swap tokens. When the message lands on ChainB it is supposed to swap an amount of some tokenB through a DEX aggregator (0x, 1Inch).
- As the message is delivered to ChainB, when trying to execute the message the swapping path has become more complex which requires more gas than the user has paid for. The executor waits for 30 minutes.
- After 30 minutes the swapping path is simpler, and the gas paid for is enough to execute the message, but the transaction is reverting due to excessive slippage.
- To ensure the state is handled correctly, the swap is wrapped in **`try/catch`** and in cases of failure the user can send the message back to ChainA to unlock his tokens.

- Although a reasonable app design it now opens a DoS attack vector whereby the griever can front-run the execution with a smaller gas limit so the revert always occurs in the `try`/`catch` block.
- This is grieving, the attacker has no specific motive but denies the app functionality(swap) to the user.

The design space is vast here and can't be covered within single issue. The main takeaway is that the application developers need to take special care in designing the receiving logic and allowing only certain actors to execute the message.

This is especially crucial if the receiving logic is complex and fluctuates concerning gas usage and other parameters.

Recommendation

Describe such cases in the documentation for integrators.

Discussion

Acknowledged with the following comments:

> It is app level logic and cross-chain apps do require careful design of the state and execution (e.g. make sure the state can finalized at the source side like OFT burn/mint).

> If it can not do it, it can of course encode those important params in their message and assert it in the app scope. but it external to the messaging layer.

Also, the following comment was made concerning the fact anyone can invoke `EndpointV2:lzReceive()` / `EndpointV2:lzCompose()` functions with an arbitrary `gasLimit` and `msg.value`:

> Yes they can front run it. But if executor is passed to the app in `lzReceive()` so if the apps do not want front running they can whitelist the caller in their apps.

[L-03] Possible DoS attack vector by setting high gas limit

Context

- EndpointV2.sol

Description

I'm going to illustrate this with an example.

Let's say an application wants to use the default Executor and enforce ordered nonce at the application level. The `OmniCounter:_acceptNonce` function showcases how it can be done. This design however opens a DoS attack vector if the app is not configured properly.

A griever can send a message from ChainA -> ChainB and specify the gas limit near the block gas limit. The message will increment the `outboundNonce` on the sending chain, but it will take the Executor a long time to get this transaction included in the block on the receiving chain. During this time all the following messages will become pending and the application loses liveness.

This type of issue belongs to the same category as [L-02] and the application developers are advised to thoroughly test their application and make sure all the edge cases are covered.

Recommendation

Describe such cases in the documentation for integrators.

Discussion

Acknowledged with the following comments:

> The options are offchain agreements and handshakes. If the sender requested 1 trillion gas, the attacker would pay for that at the source chain but executor would only do the block gas limit.

> The case someone submitting a message with the gas limit near the block gas limit is considered bad configuration of the app and not a protocol level security issue.

[I-01] Replace `Ownable` with `Ownable2Step`

Context

- ExecutorFeelib.sol
- PriceFeed.sol
- SendLibBase.sol
- Treasury.sol
- UlnBase.sol

- DVNFeeLib.sol
- AddressSizeConfig.sol
- ProxyAdmin.sol
- MessageLibManager.sol

Description

Using the `Ownable.sol` contract with its `onlyOwner` modifier is one of the most common patterns in Solidity. The biggest shortcoming with the Openzeppelin ownable implementation is that it allows the transfer of ownership to a non-existent or mistyped address.

`Ownable2Step` is much safer than `Ownable` since rather than directly transferring to the new owner, the transfer only completes after the new owner has accepted the ownership. The implementation from Openzeppelin can be found: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.5/contracts/access/Ownable2Step.sol.

Recommendation

Consider replacing `Ownable` with `Ownable2Step` in contracts that are more likely to change owner in the future.

Discussion

No comments yet.

[I-02] Follow best practices for __gap in upgradeable contracts

Context

- WorkerUpgradeable.sol

Description

Storage gaps are a pattern used in upgradeable contracts that allows to freely add new storage variables without compromising the storage compatibility with existing deployments. This pattern was introduced by Openzeppelin and is widely used in their contracts. It is described in more detail in their docs: https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps.

The convention is that the size of the `__gap` array and the amount of storage used by the contract always add up to 50 storage slots.

In your `WorkerUpgradeable.sol` contract the `__gap` array is of size 49, while storage occupies 3 slots, which breaks the convention.

Recommendation

You should implement the following simple change:

```
-    uint256[49] private __gap;
+    uint256[47] private __gap;
```

Discussion

No comments yet.

[I-03] Pass `guid` and `message` encoded as calldata to `lzReceive`

Context

- EndpointV2.sol

Description

Inside the `lzReceive` function the first line invokes the `clearPayload` internal function:

```
// clear the payload first to prevent reentrancy, and then execute the message
_clearPayload(_receiver, _origin.srcEid, _origin.sender, _origin.nonce, abi.encodePacked(_guid, _message));
```

`Guid` is a 32-bytes value while `message` can be of arbitrary length. As `abi.encodePacked` copies the data to memory, this can be expensive and with big messages can be as high as 4-5k gas.

Recommendation

Consider having the `guid` and `message` encoded together as calldata. Passing it as such to `_clearPayload` while slicing the individual values for the `lzReceive` function. This should reduce the gas cost as `lzReceive` is called frequently.

Discussion

No comments yet.