Zellic

Omnichain Governance Executor

Smart Contract Security Assessment

March 20, 2023

Prepared for:

Ryan Zarick and Isaac Zhang

LayerZero Labs

Prepared by:

Katerina Belotskaia and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for LayerZero Labs from March 13th to March 16th, 2023. During this engagement, Zellic reviewed Omnichain Governance Executor's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the sending and execution of proposals done in a secure manner?
- Is the logic of the proposal execution correct, and is the execution done in a secure manner?
- Can only privileged users participate in the decision-making process regarding the writing/execution of proposals?
- Is the overall architecture of the developed system properly integrated with the underlying LayerZero architecture?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project, such as the overall integrity of the LayerZero components
- Key custody of the owner of the project
- Any other implications of possible code changes on top of the reviewed codebase

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Omnichain Governance Executor contracts, there were no security vulnerabilities discovered.

2 Introduction

2.1 About Omnichain Governance Executor

Omnichain Governance Executor extends existing governance protocols (e.g., Compound GovernorBravo) with Layer Zero omnichain messaging protocol to enable cross-chain proposal execution. The two main contracts are OmnichainProposalSender and OmnichainGovernanceExecutor.

OmnichainProposalSender allows sending proposal actions to other chains for execution. It has to be deployed on the main chain where voting is taking place (e.g., Ethereum). When used with GovernorBravo, the owner of the OmnichainProposalSender must be set to the Timelock contract.

OmnichainGovernanceExecutor executes the proposal actions on other chains. It must be deployed on each chain that needs to support the remote proposal execution. The OmnichainGovernanceExecutor contract derives from NonBlockingLzApp and provides nonblocking message delivery, meaning failed messages will not block the message queue from the main chain. For the blocking behavior, inherit the contract from LzApp.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities,

time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Omnichain Governance Executor Contracts

Repository	https://github.com/LayerZero-Labs/omnichain-governance-executor
Version	omnichain-governance-executor: `7f7f213c149506dfeb4ce4666f261d8a1d6991d6`
Programs	• OmnichainGovernanceExecutor • OmnichainProposalSender

Type	Solidity
Platform	EVM

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of one person-week. The assessment was conducted over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 13, 2022	Start of primary review period
March 16, 2022	End of primary review period

3 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

3.1 NonblockingLzApp may block when insufficiect gas is supplied

NonblockingLzApp is designed by leveraging the 63/64th gas rule. If the external call to `_nonblockingLzReceive` runs out of gas, there will be still be 1/64th gas left to perform the write to `failedMessages` (and perform an emit), keeping the LayerZero queue unblocked. This can be coupled with passing in a `minGas` argument in `adapterParams` when sending a cross-chain message to prevent the execution from running out of gas.

This is fine in general case as the gas sent by relayer should be enough to cover both the write and the emit. But an edge case scenario may arise if 1/64th of the gas is not enough to cover the following operations, causing a revert. This can block future messages from being delivered by halting the LayerZero queue.

This can easily be recovered from by just performing a retry with a higher gas limit, but nonetheless can cause undue friction for the application and user.

This was fixed in the commit 049e9886 by holding back a minimum of 30,000 gas units to perform the store and emit after the external call.

3.2 Limited testing suite

To ensure the correctness and reliability of smart contracts, it is essential to strive for 100% code coverage during development. Comprehensive testing is a critical part of the software development lifecycle as untested code is always susceptible to bugs, regardless of its simplicity. In this regard, we recommend enhancing the testing suite, including both negative and positive tests, to ensure that the contract functions as intended and can withstand all potential circumstances, particularly those imposed by the LayerZero infrastructure, such as timeouts or failures.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1 Module: OmnichainGovernanceExecutor.sol

Function: `_executeTransaction(address target, uint value, string memory signature, bytes memory data)`

The actual transaction execution.

Inputs

- `target`
  - **Validation**: Full control from the calling function. The target is expected to be a contract address.
  - **Impact**: The target address where the transaction will be executed. Highly important to check that the payload is properly formed and that the caller is the LayerZero Endpoint.
- `value`
  - **Validation**: Full control from the calling function. The value is expected to be a `uint`.
  - **Impact**: The value to be transferred to `target`, if any. Highly important to check that the payload is properly formed and that the caller is the LayerZero Endpoint, as well as having reentrancy protection.
- `signature`
- **Validation**: Full control from the calling function. The signature is expected to be a string.
  - **Impact**: Highly important to check that the payload is properly formed and that the caller is the LayerZero Endpoint.
- `data`
  - **Validation**: Full control from the calling function. The data is expected to be bytes and stands for the calldata.

- **Impact**: Represents the calldata of the call. Highly important to check that the payload is properly formed and that the caller is the LayerZero Endpoint.

Branches and code coverage (including function calls)

Intended branches

- ☑ Assure the transactions are executed, reverting if any of them fails.
- ☑ Should not block if the transaction fails. This should be handled in the calling public function.

Negative behavior

- ☐ Should not be callable by anyone else other than the LayerZero Endpoint.

Function: `_nonblockingLzReceive(uint16, bytes memory, uint64, bytes memory _payload)`

To be called by the LayerZero Endpoint when a message from the source is received.

Inputs

- `_payload`
  - **Validation**: Full control from the calling function. The payload is expected to be a tuple of `(address[], uint[], string[], bytes[])`.
  - **Impact**: Should form multiple transactions that will be executed via `call`. Highly important to check that the payload is properly formed and that the caller is the LayerZero Endpoint.

Branches and code coverage (including function calls)

Intended branches

- ☑ Should not block if the transaction fails.
- ☑ Should emit `ProposalExecuted` event.
- ☐ Assumes `_payload` is properly constructed beforehand.

Negative behavior

- ☐ Should not be callable by anyone else other than the LayerZero Endpoint.

4.2 Module: OmnichainProposalSender.sol

Function: `execute(uint16 remoteChainId, byte[] payload, byte[] adapterParams)`

Allows owner of contract to send the message to a remote chain where the payload is `abi.encode(targets, values, signatures, calldatas)`. The OmnichainGovernanceExecutor contract will be the receiver of this message and will execute the proposal. In case of failed `lzEndpoint.send` call, the payload will be saved inside `storedExecutionHashes` for resending.

Inputs

- `remoteChainId`
  - **Validation**: The `trustedRemoteLookup` should contain the address for corresponding `remoteChainId`.
  - **Impact**: The lz ID of the remote chain.
- `payload`
  - **Validation**: There are not any checks.
  - **Impact**: The trusted payload that will be sent to the remote chain.
- `adapterParams`
  - **Validation**: There are not any checks.
  - **Impact**: The custom amount of gas required for transferring.

Branches and code coverage (including function calls)

Intended branches

- ☑ The payload will be saved for resending after failed `lzEndpoint.send` call.
- ☑ Successful `lzEndpoint.send` function call.

Negative behavior

- ☐ A caller is not owner.
- ☐ Wrong `remoteChainId`.

Function call analysis

- `lzEndpoint.send{value: msg.value}(remoteChainId, trustedRemote, payload, payable(tx.origin), address(0), adapterParams)`
  - **External/Internal?** External.
  - **Argument control?** `msg.value`, `remoteChainId`, `payload`, and `adapterParams`.
  - **Impact**: The lzEndpoint contract will call the corresponding ULN contract previously selected by the owner. The ULN contract will manage the transfer of the payload to the remote chain.

Function: `retryExecute(uint64 nonce, uint16 remoteChainId, byte[] payload, byte[] adapterParams)`

Allows any caller to resend a trusted message that for some reason was not successful in the past. Initially, the message should be sent by the contract owner using the function `execute`. If the call fails, the message will be saved inside the `storedExecutionHashes` for resending. The `msg.sender` will pay all fees.

Inputs

- `nonce`
  - **Validation**: The `storedExecutionHashes` should contains the nonzero hash for the `nonce`.
  - **Impact**: The nonce of last stored payload.
- `remoteChainId`
  - **Validation**: `hash == abi.encode(remoteChainId, payload, adapterParams)`.
  - **Impact**: The lz ID of the remote chain.
- `payload`
  - **Validation**: `hash == abi.encode(remoteChainId, payload, adapterParams)`.
  - **Impact**: The trusted payload that will be sent to the remote chain.
- `adapterParams`
  - **Validation**: `hash == abi.encode(remoteChainId, payload, adapterParams)`.
  - **Impact**: The custom amount of gas required for transferring.

Branches and code coverage (including function calls)

Intended branches

☑ Successful `lzEndpoint.send` function call.

Negative behavior

☐ The call fails again.

Function call analysis

- `lzEndpoint.send{value: msg.value}(remoteChainId, trustedRemoteLookup[remoteChainId], payload, payable(msg.sender), address(0), adapterParams);`
  - **External/Internal?** External.
  - **Argument control?** `msg.value`.
  - **Impact**: The lzEndpoint contract will call the corresponding ULN contract previously selected by the owner. The ULN contract will manage the transfer of the payload to the remote chain.

Function: `setConfig(uint16 version, uint16 chainId, uint256 configType, byte[] config)`

Allows owner of contract to set configurations of ULN contract. The address of ULN will correspond to the `version` selected by caller, but it should be valid version.

Inputs

- `version`
  - **Validation**: There are not any checks. But there are checks inside the lzEndpoint contract `_version ≤ latestVersion || _version == BLOCK_VERSION`.
  - **Impact**: If version is DEFAULT_VERSION, then the default ULN contract will be configured, otherwise selected by caller.
- `chainId`
  - **Validation**: There are not any checks.
  - **Impact**: Refers to the remote `chainId` for which the configuration will be set.
- `configType`
  - **Validation**: There are not any checks.
  - **Impact**: The `configType` determines which configuration settings need to be set.
- `config`
  - **Validation**: There are not any checks.
  - **Impact**: Contains additional configuration information.

Function call analysis

- `lzEndpoint.setConfig(version, chainId, configType, config)`
  - **External/Internal?** External.
  - **Argument control?** `version`, `chainId`, `configType`, and `config`.
  - **Impact**: This function calls the ULN contract address corresponding to the passed `version` and transmits the configuration settings data there, which will be used to the cross-chain communication.

Function: `setSendVersion(uint16 version)`

Allows owner of contract to set the version of ULN that will handle message transferring. The `version` cannot be greater than the last approved version inside the lzEndpoint contract; it can be also `BLOCK_VERSION` equal to `65535` and `DEFAULT_VERSION` equal to `0`. The `DEFAULT_VERSION` is determined by the owner of the lzEndpoint contract. If `BLOCK_VERSION` was selected, than no one will be able to send messages.

Inputs

- `version`
  - **Validation**: There are not any checks. But there are checks inside the lzEndpoint contract `_version ≤ latestVersion || _version == BLOCK_VERSION`.
  - **Impact**: The corresponding approved version of the ULN will be used to the cross-chain communication.

Branches and code coverage (including function calls)

Intended branches

- ☐ Set `BLOCK_VERSION`.
- ☐ Set `DEFAULT_VERSION`.
- ☐ Set any valid version.

Negative behavior

- ☐ Caller is not owner.
- ☐ Set an invalid version.

Function call analysis

- `lzEndpoint.setSendVersion(version);`
  - **External/Internal?** External.
  - **Argument control?** `version`.
  - **Impact**: Allows to set the `version` of ULN contract that will be used to transfer messages from this contract. The transaction will be reverted if `version` is not valid.

Function: `setTrustedRemoteAddress(uint16 remoteChainId, byte[] remoteAddress)`

Allows owner of contract to set the trusted receiver address for the cross-chain communication.

Inputs

- `remoteChainId`
  - **Validation** There are not any checks.
  - **Impact** The remote chain ID for which the receiver's address will be set.
- `remoteAddress`
  - **Validation**: There are not any checks.

- **Impact**: The remote message receiver address.

5 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

There were no findings during our audit.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.