Zellic

Prepared for
Barbara Hubert
LayerZero Labs

Prepared by
Junghoon Cho
Sunwoo Hwang
Zellic

October 25, 2024

LZ Read

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for LayerZero Labs from October 15th to October 21st, 2024. During this engagement, Zellic reviewed LZ Read's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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 LZ Read secure against common smart contract vulnerabilities?
- Does the codec correctly encode and decode the command packet?
- Are there issues in the protocol math or logic that lead to loss of funds?
- Is the payment for the protocol's contributors, including DVN, executor, and worker, being made accurately?
- Does LZ Read have any centralization risks?

1.3. Non-goals and Limitations

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

- Off-chain DVNs
- Front-end components
- Infrastructure relating to the project
- Key custody

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

1.4. Results

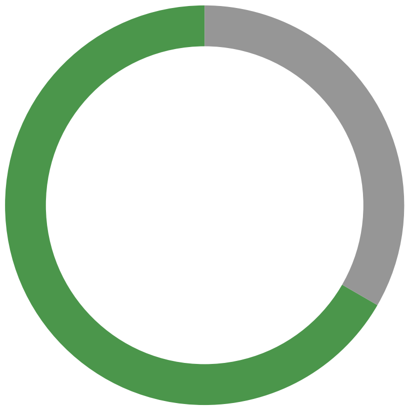
During our assessment on the scoped LZ Read contracts, we discovered three findings. No critical issues were found. Two findings were of low impact and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

LayerZero Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	2
⬜ Informational	1



2. Introduction

2.1. About LZ Read

LayerZero Labs contributed the following description of LZ Read:

> LayerZero Read and Compute enhances the way developers interact with on-chain data. Allowing a developer to call data from any time period with pin-point block accuracy on any supported chain.

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 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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

LZ Read Contracts

Repository	https://github.com/LayerZero-Labs/monorepo ↗
Versions	monorepo: `e88bb176ead2f631dcab411bc631212389deb294` monorepo: `56cc7e7ecbc5beb469d6b494e0c5e44de348e5ab`
Programs	• messagelib/contracts/uln/readlib/ReadLib1002.sol • messagelib/contracts/uln/readlib/ReadLibBase.sol • messagelib/contracts/uln/interfaces/ILayerZeroReadDVN.sol • messagelib/contracts/interfaces/ILayerZeroReadExecutor.sol • messagelib/contracts/uln/dvn/MultiSig.sol • messagelib/contracts/uln/dvn/DVN.sol • oapp/contracts/oapp/libs/ReadCmdCodecV1.sol • oapp/contracts/oapp/OAppRead.sol • oapp/contracts/oapp/interfaces/IOAppComputer.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one and a half person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Junghoon Cho
Engineer
junghoon@zellic.io ↗

Sunwoo Hwang
Engineer
sunwoo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 15, 2024	Start of primary review period
October 18, 2024	Kick-off call
October 21, 2024	End of primary review period
November 1, 2024	Assessment version updated to `56cc7e7e` ↗

3. Detailed Findings

3.1. The `workerFeeLib` variable can remain uninitialized

Target	DVN.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The DVN contract interacts with various functions of the DVNFeeLib contract through the `workerFeeLib` variable to calculate the fees for workers within the protocol.

```
function getFee(
    address _sender,
    bytes calldata /*_packetHeader*/,
    bytes calldata _cmd,
    bytes calldata _options
) external view onlyAcl(_sender) returns (uint256 fee) {
    IDVNFeeLib.FeeParamsForRead memory feeParams
    = IDVNFeeLib.FeeParamsForRead(
        priceFeed,
        _sender,
        quorum,
        defaultMultiplierBps
    );
    fee = IDVNFeeLib(workerFeeLib).getFee(feeParams, dstConfig[localEidV2],
    _cmd, _options);
}
```

However, `workerFeeLib` is not initialized in the constructor and must be assigned later with the address of the DVNFeeLib contract using the `setWorkerFeeLib` method. The current version of the DVN contract does not include any logic to check whether the `workerFeeLib` has been initialized.

Impact

All transactions in which DVN calls methods from `workerFeeLib` may revert, potentially causing the overall functionality of the protocol to fail.

Recommendations

Add logic to check whether `workerFeeLib` has been initialized.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.2. Zero address can be set as a signer

Target	Multisig.sol		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

The `_setSigner` function is used to register and remove signers who can participate in multi-sig verification.

```
function _setSigner(address _signer, bool _active) internal {
    if (_active) {
        if (!signerSet.add(_signer)) {
            revert MultiSig_StateAlreadySet(_signer, _active);
        }
    } else {
        if (!signerSet.remove(_signer)) {
            revert MultiSig_StateNotSet(_signer, _active);
        }
    }

    uint64 _signerSize = uint64(signerSet.length());
    uint64 _quorum = quorum;
    if (_signerSize < _quorum) {
        revert MultiSig_SignersSizeIsLessThanQuorum(_signerSize, _quorum);
    }
    emit UpdateSigner(_signer, _active);
}
```

However, this function does not check whether `_signer` is a zero address.

Impact

Allowing the zero address to be added as a signer disrupts quorum calculations because it counts towards the total number of signers but cannot provide valid signatures, making it impossible to meet the required quorum.

```
bytes32 messageDigest = _getEthSignedMessageHash(_hash);
```

```
address lastSigner = address(0); // There cannot be a signer with address 0.
for (uint256 i = 0; i < quorum; i++) {
```

Additionally, since the code assumes the zero address is not a signer (e.g., `verifySignatures`), including it violates these assumptions, leading to potential logic errors. Meanwhile, a registered signer can be revoked at any time through `_setSigner`, so it does not cause permanent disruption of functionality.

Recommendations

Add logic to check whether `_signer` is a zero address or not.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `4de76da2` ↗.

3.3. The `_shouldCheckHash` function has omitted `ReadLib1002.verify` from the whitelist

Target	DVN.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

For gas efficiency, the `execute` function does not check the calldata hash for several functions that cannot alter the state through replay attacks.

```
function _shouldCheckHash(bytes4 _functionSig) internal pure returns (bool) {
    // never check for these selectors to save gas
    return
        _functionSig != IReceiveUlnE2.verify.selector && // 0x0223536e,
    replaying won't change the state
        _functionSig != ILayerZeroUltraLightNodeV2.updateHash.selector; //
    0x704316e5, replaying will be revert at uln
}
```

The list of such functions is defined in `_shouldCheckHash`, but in the current implementation, the `verify(bytes calldata _packetHeader, bytes32 _cmdHash, bytes32 _payloadHash)` function from ReadLib1002 is missing from this list.

Impact

The `_shouldCheckHash` function may fail to achieve its intended gas efficiency.

Recommendations

Modify the `_shouldCheckHash` function to return false for the selector of the `verify` function in the ReadLib1002 contract as well.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 4de76da2 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Code maturity

The overall code quality of the codebase is excellent, but here we list some miscellaneous code-maturity issues.

Inconsistent coding style of `getFee` functions

There are several variations of the `getFee` function defined in the DVN contract to support compatibility for other DVNs. In the current version, only the one that supports ReadLib assigns a variable to the fee before returning it, while the others do not. For consistency and readability, we recommend unifying the style by assigning the return value to a variable across all `getFee` functions.

```
/// @dev to support ULNv2
function getFee(
    uint16 _dstEid, uint16 /*_outboundProofType*/, uint64 _confirmations,
    address _sender
) public view onlyAcl(_sender) returns (uint256 fee) {
[...]
    return IDVNFeeLib(workerFeeLib).getFee(params, dstConfig[_dstEid],
    bytes(""));
}
/// @dev to support ReadLib
function getFee(
    address _sender, bytes calldata /*_packetHeader*/, bytes calldata _cmd,
    bytes calldata _options
) external view onlyAcl(_sender) returns (uint256 fee) {
[...]
    fee = IDVNFeeLib(workerFeeLib).getFee(feeParams, dstConfig[localEidV2],
    _cmd, _options);
}
```

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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.

5.1. Module: DVN.sol

Function: `execute(ExecuteParam[] _params)`

This function executes arbitrary transactions. It allows the admin to execute transactions if each transaction is signed by the required number of signers. The function does not revert if a transaction fails, ensuring that the contract can continue executing the remaining transactions.

Inputs

- `_params`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array of the `ExecuteParam` struct that contains the `vid`, `target`, `calldata`, `expiration`, and `signatures`.

Branches and code coverage

Intended branches

- Execute the transactions with the given parameters.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the role `ADMIN_ROLE`.
    - ☑ Negative test

Function: `grantRole(byte[32] _role, address _account)`

This function grants a role to an account.

Inputs

- `_role`
    - **Control**: Arbitrary.

    - **Constraints**: None.
    - **Impact**: The role to grant.
- `_account`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The account to grant the role to.

Branches and code coverage

Intended branches

- Grant the role to the account.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the role `ADMIN_ROLE` or is not the contract itself.
    - ☑ Negative test

Function: `quorumChangeAdmin(ExecuteParam _param)`

This function grants the role `ADMIN_ROLE` to the new admin. It allows the signers to add a new admin to the contract.

Inputs

- `_param`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The `ExecuteParam` struct that contains the `vid`, `target`, `calldata`, `expiration`, and `signatures`.

Branches and code coverage

Intended branches

- Verify the signatures and grant the role `ADMIN_ROLE` to the new admin.
    - ☑ Test coverage

Negative behavior

- Revert if the `expiration` is less than or equal to the current block timestamp.
    - ☑ Negative test
- Revert if the `target` is not the contract itself.

- ☑ Negative test
- Revert if the `vid` is not same as the `vid` in the contract.
  - ☑ Negative test
- Revert if the signatures are not valid.
  - ☐ Negative test
- Revert if the hash of the params is already used.
  - ☐ Negative test

Function: `revokeRole(byte[32] _role, address _account)`

This function revokes a role from an account.

Inputs

- `_role`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The role to revoke.
- `_account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The account to revoke the role from.

Branches and code coverage

Intended branches

- Revoke the role from the account.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the role `ADMIN_ROLE` or is not the contract itself.
  - ☑ Negative test

Function: `setDstConfig(DstConfigParam[] _params)`

This function sets the configuration for the destination chain. The configuration includes the `gas`, `multiplierBps`, and `floorMarginUSD` for each destination chain.

Inputs

- `_params`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array or the `DstConfig` struct that contains the `gas`, `multiplierBps`, and `floorMarginUSD`.

Branches and code coverage

Intended branches

- Set the configurations for each destination chain.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the role `ADMIN_ROLE`.
    - ☑ Negative test

Function: `setQuorum(uint64 _quorum)`

This function sets the quorum for the `signerSet`. The quorum represents the number of signers required to execute a transaction, so it must be less than the size of the `signerSet` and greater than zero.

Inputs

- `_quorum`
    - **Control**: Arbitrary.
    - **Constraints**: Must be less than the size of the `signerSet` and greater than zero.
    - **Impact**: The new quorum value.

Branches and code coverage

Intended branches

- Set the quorum to the new value.
    - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract itself.
    - ☑ Negative test
- Reverts if `_quorum` is zero or greater than the size of the `signerSet`.

- ☐ Negative test

Function: `setSigner(address _signer, bool _active)`

This function sets the status of a signer. It adds a signer to the `signerSet` if `_active` is `true` and removes a signer from the `signerSet` if `_active` is `false`. Additionally, the size of the `signerSet` must always be greater than the `quorum`.

Inputs

- `_signer`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The address of the signer.
- `_active`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The status of the signer.

Branches and code coverage

Intended branches

- Add or remove the signer from the `signerSet`.
    - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract itself.
    - ☑ Negative test

Function: `withdrawFeeFromUlnV2(address _lib, address payable _to, uint256 _amount)`

This function is for supporting the ULNv2 contract. It withdraws the fee from the ULNv2 contract and sends it to the specified address.

Inputs

- `_lib`
    - **Control**: Arbitrary.
    - **Constraints**: Must have the role `MESSAGE_LIB_ROLE`.

    - **Impact**: The ULN contract that will be used to withdraw the fee.
- `_to`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The address to which the fee will be sent.
- `_amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The amount of the fee to be withdrawn.

Branches and code coverage

Intended branches

- Withdraw the fee from the `_lib` contract and send it to the `_to` address.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the role `ADMIN_ROLE`.
    - ☑ Negative test
- Revert if the `_lib` does not have the role `MESSAGE_LIB_ROLE`.
    - ☑ Negative test

Function call analysis

- `ILayerZeroUltraLightNodeV2(_lib).withdrawNative(_to, _amount)`
    - **What is controllable?** All.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.2. Module: MultiSig.sol

Function: `verifySignatures(byte[32] _hash, bytes _signatures)`

The `verifySignatures` function checks if the number of signatures is equal to the `quorum` and if they are valid. It ensures that the signers of each signature are in the `signerSet`. If all conditions are met, the function returns `true`; otherwise, it returns `false`.

Inputs

- `_hash`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: The hash of the message that the signers signed.

- `_signatures`
  - **Control**: Arbitrary.
  - **Constraints**: The length of `_signatures` must be `65 * quorum`.
  - **Impact**: The signatures of each signer.

Branches and code coverage

Intended branches

- Verify the signatures.
  - ☑ Test coverage

Negative behavior

- Return false if the length of `_signatures` is not `65 * quorum`.
  - ☐ Negative test
- Return false if one of the signatures is invalid.
  - ☐ Negative test
- Return false if one of the signers is duplicated.
  - ☐ Negative test
- Return false if one of the signers is not in the `signerSet`.
  - ☐ Negative test

5.3. Module: ReadLib1002.sol

Function: `commitVerification(bytes _packetHeader, byte[32] _cmdHash, byte[32] _payloadHash)`

This function verifies a packet on the same chain where it was sent.

Inputs

- `_packetHeader`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be exactly 81 bytes in length. Must have a version matching `PacketV1Codec.PACKET_VERSION`. The `dstEid` within the packet header must equal `localEid`.
  - **Impact**: Essential packet metadata required for verification.
- `_cmdHash`
  - **Control**: Controlled by the caller.

- **Constraints**: Must match the stored command hash in `cmdHashLookup` for the given `receiver`, `srcEid`, and `nonce`.
- **Impact**: Used to verify the integrity of the command associated with the packet.

- `_payloadHash`
    - **Control**: Controlled by the caller.
    - **Constraints**: Must match a hash of the payload.
    - **Impact**: Passed to `_verifyAndReclaimStorage` and `ILayerZeroEndpointV2.verify` functions for verification.

Branches and code coverage

Intended branches

- `cmdHash` is checked, verification is processed in `ILayerZeroEndpointV2.verify`, and a corresponding event is emitted.
    - ☑ Test coverage

Negative behavior

- Revert if the `_packetHeader` length is not 81 bytes.
    - ☑ Negative test
- Revert if the `_packetHeader` version does not match `PacketV1Codec.PACKET_VERSION`.
    - ☑ Negative test
- Revert if the `_packetHeader`'s `dstEid` does not match `localEid`.
    - ☑ Negative test
- Revert if `cmdHashLookup[receiver][srcEid][nonce]` does not equal `_cmdHash`.
    - ☑ Negative test

Function: `send(Packet _packet, bytes _options, bool _payInLzToken)`

This function is called by the endpoint contract to send a packet.

Inputs

- `_packet`
    - **Control**: Controlled by the caller.
    - **Constraints**: The `sender` and `receiver` addresses in the packet must match.
    - **Impact**: Packet to be sent, including message data, sender, receiver, and so on.
- `_options`
    - **Control**: Controlled by the caller.
    - **Constraints**: Must be decoded to valid `executorOptions` and `dvnOptions`.

    - **Impact**: Several parameters indicating payment preference.
- `_payInLzToken`
    - **Control**: Controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Boolean value indicating payment preference.

Branches and code coverage

Intended branches

- The payment to workers/treasury is processed, and the hash of the message is stored in `cmdHashLookup` when called by the endpoint with valid inputs.
    - [✓] Test coverage

Negative behavior

- Revert if not called by the endpoint.
    - [✓] Negative test
- Revert if the `sender` and `receiver` in the packet do not match.
    - [✓] Negative test

Function: `setConfig(address _oapp, SetConfigParam[] _params)`

This function allows the endpoint to set configurations for a specific OApp.

Inputs

- `_oapp`
    - **Control**: Controlled by the caller.
    - **Constraints**: Must be a valid address representing the OApp.
    - **Impact**: The OApp address for which the configurations are being set.
- `_params`
    - **Control**: Controlled by the caller.
    - **Constraints**: Each parameter must have a supported `eid` and a valid `configType`.
    - **Impact**: The configuration parameters to be applied to the OApp for each specified `eid`.

Branches and code coverage

Intended branches

- Valid configurations are set when called by the endpoint.

    - ☑ Test coverage

Negative behavior

- Revert if not called by the endpoint.
    - ☑ Negative test
- Revert if the given `eid` is not supported.
    - ☑ Negative.test
- Revert if the given `configType` is not `CONFIG_TYPE_CMD_LID_CONFIG`.
    - ☑ Negative.test

Function: `setTreasuryNativeFeeCap(uint256 _newTreasuryNativeFeeCap)`

This function allows the contract owner to decrease the `treasuryNativeFeeCap` value.

Inputs

- `_newTreasuryNativeFeeCap`
    - **Control**: Controlled by the caller.
    - **Constraints**: Must be less than or equal to the previous value.
    - **Impact**: Updated cap on the treasury's native fee.

Branches and code coverage

Intended branches

- The cap of the treasury's native fee is set when called by the owner with a valid value of `treasuryNativeFeeCap`.
    - ☑ Test coverage

Negative behavior

- Revert if not called by the contract owner.
    - ☑ Negative test
- Revert if `_newTreasuryNativeFeeCap` is greater than the current `treasuryNativeFeeCap`.
    - ☑ Negative test

Function: `setTreasury(address _treasury)`

This function allows the contract owner to set the address of the treasury.

Inputs

- `_treasury`

- **Control**: Controlled by the caller.
- **Constraints**: None.
- **Impact**: Updated treasury address.

Branches and code coverage

Intended branches

- Treasury address is set when called by the owner.
  - ☑ Test coverage

Negative behavior

- Revert if not called by the contract owner.
  - ☑ Negative test

Function: `withdrawFee(address _to, uint256 _amount)`

This function allows a DVN/executor/treasury to withdraw a specified amount of their accumulated fees to a designated address.

Inputs

- `_to`
  - **Control**: Controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The destination address where the withdrawn fees will be sent.
- `_amount`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be less than or equal to the caller's available fees.
  - **Impact**: The amount of fees the caller intends to withdraw.

Branches and code coverage

Intended branches

- Successful withdrawal when `_amount` is less than or equal to the caller's available fees.
  - ☑ Test coverage

Negative behavior

- Revert if `_amount` exceeds available fees.
  - ☑ Negative test

Function: `withdrawLzTokenFee(address _lzToken, address _to, uint256 _amount)`

This function allows the treasury to withdraw fees accumulated by the contract in LZ token.

Inputs

- `_lzToken`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must not be the same as the native token.
  - **Impact**: The address of the LZ token to withdraw.
- `_to`
  - **Control**: Controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The destination address where the LZ tokens will be sent.
- `_amount`
  - **Control**: Controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of LZ tokens to withdraw.

Branches and code coverage

Intended branches

- Successful withdrawal when called by the treasury with an available amount and valid `_lzToken`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the treasury.
  - ☑ Negative test
- Revert if `_lzToken` is the same as the native token.
  - ☑ Negative test

5.4. Module: ReadLibBase.sol

Function: `setDefaultReadLibConfigs(SetDefaultReadLibConfigParam[] _params)`

This function allows the contract owner to set the default ReadLib configurations for multiple eIDs.

Inputs

- `_params`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Each configuration must not use `NIL_DVN_COUNT` for `requiredDVNCount` and `optionalDVNCount` and must require at least one DVN, and `executor` must not be the zero address.
  - **Impact**: ReadLib configurations for the specified eIDs.

Branches and code coverage

Intended branches

- Valid configurations are set when called by the owner.
  - ☑ Test coverage

Negative behavior

- Revert if not called by the contract owner.
  - ☑ Negative test
- Revert if `param.config.requiredDVNCount` is `NIL_DVN_COUNT`.
  - ☑ Negative test
- Revert if `param.config.requiredDVNCount` is `NIL_DVN_COUNT`.
  - ☑ Negative test
- Revert if the configuration does not require at least one DVN.
  - ☑ Negative test
- Revert if the `executor` is the zero address.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet.

During our assessment on the scoped LZ Read contracts, we discovered three findings. No critical issues were found. Two findings were of low impact and the remaining finding was informational in nature.

6.1. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.