Zellic

Prepared for
Tiancheng Xie
Abner Jia
Polyhedra Network

Prepared by
Aaron Esau
Vlad Toie
Zellic

February 5, 2024

Polyhedra DVN
Smart Contract Security Assessment

Contents

About Zellic **4**

1. Executive Summary **4**

1.1. Goals of the Assessment 5

1.2. Non-goals and Limitations 5

1.3. Results 5

2. Introduction **6**

2.1. About Polyhedra DVN 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Verification procedure lacks minimum confirmations check 11

3.2. Malicious libraries may not be removable 13

3.3. Idempotency checks are not performed 16

3.4. Centralization risk in `setBlockUpdater` 17

4. Discussion **18**

4.1. Message library not removed from the `lzMessageLibs` array 19

4.2. Use a large number of confirmations 19

4.3. Missing `AlreadySet` check 20

5. Threat Model **20**

5.1. Module: ZkBridgeOracleV2.sol 21

6. Assessment Results **29**

6.1. Disclaimer 30

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. Executive Summary

Zellic conducted a security assessment for Polyhedra Network from January 29th to February 5th, 2024. During this engagement, Zellic reviewed Polyhedra DVN'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:

- Are all relevant packet fields included in the verification?
- Is the DVN implemented to LayerZero's specification?
- Is the MPT verification code sound?

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
- Key custody

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 Polyhedra DVN contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Polyhedra Network's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Polyhedra DVN

Polyhedra DVN is an oracle built for LayerZero V2 that provides transaction proofs for LayerZero using zero-knowledge proofs, making the LayerZero system more decentralized and improving its security.

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:

Polyhedra DVN Contracts

Repository	https://github.com/zkBridge-integration/layerzero-oracle-audit ↗
Version	layerzero-oracle-audit: `d6253b3c9cb4156f1be88e25268968a5c0fdcede`
Programs	• oracle/ZkBridgeOracleV2.sol • mpt/ZKMptValidatorV2.sol • libraries/LzV2PacketCodec.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight person-days. The assessment was conducted over the course of six 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:

Aaron Esau
Engineer
aaron@zellic.io

Vlad Toie
Engineer
vlad@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 29, 2024	Start of primary review period
February 5, 2024	End of primary review period

3. Detailed Findings

3.1. Verification procedure lacks minimum confirmations check

Target	ZkBridgeOracleV2		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `_verify` function is an essential component of the ZKBridgeOracleV2 contract. It ensures that the parameters of each particular packet that has been transmitted cross-chain are legitimate, and it determines whether those packets can be forwarded to the Receive Ultra-Light Node (ULN).

```
function _verify(
    bytes32 _blockHash,
    bytes32 _receiptHash,
    uint32 _srcEid,
    address _receiver,
    bytes memory _packetHeader,
    bytes32 _payloadHash
) internal {
    IBlockUpdater blockUpdater = blockUpdaters[_srcEid];
    if (address(blockUpdater) == address(0))
    revert UnsupportedUpdater(_srcEid);
    (bool exist, uint256 blockConfirmation)
    = blockUpdater.checkBlockConfirmation(_blockHash, _receiptHash);
    if (!exist) revert BlockNotSet();
    (address receiverLib,)
    = layerZeroEndpointV2.getReceiveLibrary(_receiver, _srcEid);
    IReceiveUlnE2(receiverLib).verify(_packetHeader, _payloadHash,
    uint64(blockConfirmation));
}
```

In the current implementation, the function checks whether the given `_blockHash` is valid for the `_receiptHash`, and if so, it forwards the specific packet on to the Ultra-Light Node.

As per the official LayerZero DVN documentation ↗ (point 4) requirements, however, prior to the Ultra-Light Node verification, the `_verify` function must ensure that the packet has accrued enough block confirmations.

Impact

Lack of an adequate verification procedure might lead to the exclusion of the DVN from the LayerZero Verification layer, as it might propose to verifications packets that have not yet passed enough block confirmations.

Recommendations

We recommend implementing the following checks that ensure all LayerZero standards are respected adequately:

```
function _verify(
    bytes32 _blockHash,
    bytes32 _receiptHash,
    uint32 _srcEid,
    address _receiver,
    bytes memory _packetHeader,
    bytes32 _payloadHash
) internal {
    // ...
    (address receiverLib,)
    = layerZeroEndpointV2.getReceiveLibrary(_receiver, _srcEid);

    UlnConfig memory ulnConfig = IReceiveUlnE2(receiverLib).getUlnConfig
        (
        _receiver,
        _srcEid
    );

    require(
        blockConfirmations >= ulnConfig.confirmations,
        "ZkBridgeOracleV2: packet not verified"
    );

    IReceiveUlnE2(receiverLib).verify(_packetHeader, _payloadHash,
    uint64(blockConfirmation));
}
```

Remediation

This issue has been acknowledged by Polyhedra Network, and a fix was implemented in commit 17df5bda ↗.

3.2. Malicious libraries may not be removable

Target	ZkBridgeOracleV2		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Messaging libraries are responsible for notifying DVNs of messages being sent ("send library") on the source chain and receiving verifications from DVNs ("receive library") on the destination chain.

They verify the payload of each packet, committing the verified payload hash to the endpoint after the extrinsic security requirement (e.g., DVN threshold) is fulfilled.

Any send library in ZkBridgeOracleV2's `messageLibLookup` with `enabled` set to `true` may call the `assignJob` function. If the message's `dstEid` is the `localEid`, the DVN skips the off-chain component and shortcuts to verify with the legitimate receive library:

```
function assignJob(AssignJobParam calldata _param,
    bytes calldata /*_options*/ )
external
payable
returns (uint256 fee)
{
    if (!supportedDstChain[_param.dstEid])
    revert UnsupportedChain(_param.dstEid);
    if (!isSupportedMessageLib(msg.sender)) revert UnsupportedSendLib();
    fee = chainFeeLookup[_param.dstEid];
    emit OracleNotified(_param.dstEid, _param.confirmations, _param.sender,
    fee);

    if (localEid == _param.dstEid) {
        (address receiverLib,) = layerZeroEndpointV2.getReceiveLibrary(
            _param.packetHeader.receiver(), localEid);
        IReceiveUlnE2(receiverLib).verify(_param.packetHeader, _param.
            payloadHash, _param.confirmations);
    }
}
```

If a malicious library were to be added (e.g., if an upgradable library were to be added whose owner's keys were compromised), it is possible for the library to prevent itself from being unconfigured from the DVN because the removal function makes external calls to the malicious library where it could revert:

```
function removeLzMessageLib(address _messageLib) external onlyOwner {
    if (!messageLibLookup[_messageLib].enabled)
    revert MessageLibAlreadyDeleted();

    messageLibLookup[_messageLib].enabled = false;
    uint256 fee = ISendLib(_messageLib).fees(address(this));
    if (fee > 0) {
        ISendLib(_messageLib).withdrawFee(payable(owner()), fee);
        emit WithdrawFee(_messageLib, owner(), fee);
    }
}
```

Impact

A malicious library can prevent itself from being removed, which would permanently impact the integrity of the DVN.

Recommendations

Ensuring legitimacy of libraries

First, to reduce centralization risk by ensuring both Polyhedra Network and LayerZero Labs agree on the legitimacy and security of a messaging library, we recommend querying the LayerZero endpoint to ensure the `_messageLib` being added is a registered library:

```
function addLzMessageLib(address _messageLib) external onlyOwner {
    require(
        layerZeroEndpointV2.isRegisteredLibrary(_messageLib),
        "message library not registered with LayerZero"
    );
    // ...
}
```

Ensuring malicious libraries can be removed

Additionally, to ensure malicious send libraries can be removed, we recommend removing the fee-withdrawal logic from the `removeLzMessageLib` function; a function to withdraw fees already exists. It is named `withdrawFee`. So, having the logic in `removeLzMessageLib` only adds risk.

Alternatively, consider adapting the `removeLzMessageLib` to use a try-catch such that even in the case that any of the `_messageLib` functions were to revert, the `messageLibLookup` entry would still be removable.

Remediation

Polyhedra Network provided the following response to this finding:

> MessageLib is added via our multi-signature wallet in this function. During the addition process, we conduct thorough verification and review of the contract source code to ensure that the messageLib meets our requirements. Subsequently, we will upgrade the code to add support for v1 301. The Endpoint in v1 does not have this function for verification and is not compatible with the v2 Endpoint. Therefore, the audit and verification of adding messageLib should occur off-chain.

3.3. Idempotency checks are not performed

Target	ZkBridgeOracleV2		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Note that idempotency checks do not exist on-chain.

Impact

Lack of an idempotency check allows for reverification of the same packet on chain.

Recommendations

We recommend performing the idempotency check as per the official LayerZero DVN documentation ↗ (point 5).

Remediation

The Polyhedra team has acknowledged this finding and provided the following clarification as response:

> We perform the idempotency check off-chain before calling the verify function to ensure that the validation has not been submitted to LayerZero yet.

3.4. Centralization risk in `setBlockUpdater`

Target	ZkBridgeOracleV2		
Category	Protocol Risks	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `setBlockUpdater` function in ZkBridgeOracleV2 allows the contract owner to modify the block updater for any source chain. The block updater is responsible for validating receipt hashes against a given block hash.

```
function setBlockUpdater(uint16 _sourceChainId, address _blockUpdater)
    external onlyOwner {
    require(_blockUpdater != address(0), "ZkBridgeOracle:Zero address");
    emit ModBlockUpdater(_sourceChainId,
    address(blockUpdaters[_sourceChainId]), _blockUpdater);
    blockUpdaters[_sourceChainId] = IBlockUpdater(_blockUpdater);
}
```

Impact

The owner can potentially change the block updater to an arbitrary contract and thereby bypass the security guarantees of the contract.

Recommendations

We recommend integrating a governance for the owner or a timelock mechanism to remediate arbitrary updates to `blockUpdaters`.

Remediation

Polyhedra Network provided the following response to this finding:

> Regarding your concerns with the ZkBridgeOracle contract and specifically, the `setBlock-Updater` function, please be aware that the contract currently possesses the ability to change the `blockUpdater` due to the hi-tech and intricate nature of the Zero-Knowledge

Proofs. This upgradability allows us to swiftly deal with and settle any unforeseen issues, thereby enhancing the security and overall continuity of our services.

However, this arrangement is not permanent. Our ultimate plan is to transition towards a non-adjustable contract when the operations of the ZkBridgeOracle proves to be steady and reliable. At that point, we expect an enhancement in the solidity and safety of the contract. We foresee this transition to be implemented within a month, and we greatly appreciate your understanding on this matter.

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. Message library not removed from the `lzMessageLibs` array

The `removeLzMessageLib` function performs the removal of a `_messageLib` from the `messageLibLookup` mapping. However, the function does not remove the `_messageLib` from the `lzMessageLibs` array. This can lead to inconsistencies in the state of the contract, as the `lzMessageLibs` array is used to iterate over all message libraries across the contract.

4.2. Use a large number of confirmations

In order to avoid messages from being stuck in cases when the BlockUpdater contract malfunctions for any reasons, the number of verifications in the call to the Ultra Light Node could be set to huge number, to ensure its success.

```
function _verify(
    bytes32 _blockHash,
    bytes32 _receiptHash,
    uint32 _srcEid,
    address _receiver,
    bytes memory _packetHeader,
    bytes32 _payloadHash
) internal {
    // ...
    IReceiveUlnE2(receiverLib).verify(
        _packetHeader,
        _payloadHash,
-       uint64(blockConfirmation)
+       type(uint64).max
    );
}
```

Note that this particular solution (or any solution of this sort) is not the safest, as it goes against the verification of the number of confirmations, as recommended in 3.1. ↗ and should only be considered should the BlockUpdater be impaired for any reason.

Polyhedra Network provided the following response:

As you highlighted, the proposed solution of utilizing an exceptionally large number of confirmations, while intended as a workaround for scenarios where the BlockUpdater may malfunction, is fundamentally not secure. It is essential to address and rectify any issues with the BlockUpdater directly.

Relying on Oracles to push transactions with the maximum number of confirmations in the event of a BlockUpdater failure circumvents the essential verification process. This approach compromises the integrity of the transactions on the sending chain by not duly verifying them.

Therefore, we have decided not to implement the suggested modification. Ensuring the integrity of the verification process and the reliability of the BlockUpdater takes precedence, and we will focus on addressing the root causes of any issues with the BlockUpdater directly.

4.3. Missing `AlreadySet` check

There is no security impact of this, but note that `setReceiveView` is inconsistent in that it does not check whether the new value is equal to the old value like other setters in the contract. Consider adding the following:

```
if (receiveLibToView[receiveLib_] == receiveLibView_) revert AlreadySet();
```

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: ZkBridgeOracleV2.sol

Function: `addLzMessageLib(address _messageLib)`

Adds a new message library to the list of supported message libraries. Only callable by the owner.

Inputs

- `_messageLib`
    - **Control**: Full.
    - **Constraints**: Must not be already added. Must return a valid message library type when `messageLibType()` is called on it.
    - **Impact**: The address of the message library to add.

Branches and code coverage

Intended branches

- Successfully adds the message library to the list of supported message libraries.
    - ☑ Test coverage

Negative behavior

- Caller is not the owner.
    - ☑ Negative test
- Message library was already added.
    - ☑ Negative test
- Message library type is not valid.
    - ☑ Negative test

Function: `assignJob(AssignJobParam _param, bytes)`

Called by the send library to notify the oracle of a new job. Shortcuts if the source and destination chains are the same to call verify.

Only callable by enabled messaging libraries.

Inputs

- `_param`
    - **Control**: Full.
    - **Constraints**: The `_param.dstEid` must be an enabled destination chain.
    - **Impact**: Contains the details of the packet to notify the oracle about.

Branches and code coverage

Intended branches

- Successfully notifies oracle of job.
    - ☑ Test coverage
- Shortcuts when the destination chain is the same as the source chain EIDs.
    - ☑ Test coverage

Negative behavior

- Unsupported destination chain.
    - ☑ Negative test
- Unsupported send library.
    - ☑ Negative test

Function: `batchVerify(byte[32][] _blockHashs, byte[][] _encodedPayloads, byte[][] _zkMptProof)`

This function is the bulk version of `verify(byte[32] _blockHash, byte[] _encodedPayload, byte[] _zkMptProof)`. This sign-off applies to both functions.

Performs batch verification of the zkMPT proofs for the given block hashes and encoded payloads.

Internally calls `_verify`, which ensures the `blockHash` exists, and submits the packet to the receiver library.

Inputs

- `_blockHashs`
    - **Control**: Full.
    - **Constraints**: Must be the same length as `_encodedPayloads` and `_zkMptProof`. Must contain the block hash matching the receipt root for each.
    - **Impact**: The block hashes containing the receipt roots for the encoded payloads.
- `_encodedPayloads`
    - **Control**: Full.

    - **Constraints**: Must be the same length as `_blockHashs` and `_zkMptProof`. Must contain the encoded payloads for each.
    - **Impact**: The `EncodedPayload` structs to verify.
- `_zkMptProof`
    - **Control**: Full.
    - **Constraints**: Must be the same length as `_blockHashs` and `_encodedPayloads`. Must contain valid ZK proofs.
    - **Impact**: The ZK proofs to validate.

Branches and code coverage

Intended branches

- Verifies a valid zkMPT proof.
    - ☑ Test coverage

Negative behavior

- Invalid zkMPT proof provided.
    - ☑ Negative test
- `zkMptValidator` not configured.
    - ☑ Negative test
- `blockUpdater` not configured.
    - ☑ Negative test
- Nonexisting `blockHash` provided.
    - ☑ Negative test
- Lengths of the arrays are mismatched.
    - ☐ Negative test

Function: `removeLzMessageLib(address _messageLib)`

Removes a message library from the list of supported message libraries. Only callable by the owner.

Inputs

- `_messageLib`
    - **Control**: Full.
    - **Constraints**: Must be an enabled message library.
    - **Impact**: Specifies which messaging library to withdraw fees and mark as disabled.

Branches and code coverage

Intended branches

- Successfully disables the message library and withdraws fees.
  - ☑ Test coverage
- Removes the message library from the `lzMessageLibs` array.
  - ☐ Test coverage

Negative behavior

- Caller is not owner.
  - ☑ Negative test
- Messaging library is not enabled.
  - ☑ Negative test

Function: `setBlockUpdater(uint32 _srcEid, address _newBlockUpdater)`

Sets the block-updater contract. Callable by the owner only.

Inputs

- `_srcEid`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The source EID to configure the block updater for.
- `_newBlockUpdater`
  - **Control**: Full.
  - **Constraints**: Cannot be zero and cannot be equal to the currently stored block-updater address.
  - **Impact**: The new address to use as the block updater.

Branches and code coverage

Intended branches

- Updates the block-updater configuration.
  - ☑ Test coverage

Negative behavior

- Caller is not the owner.
  - ☑ Negative test
- Reverts if the `_newBlockUpdater` is zero.
  - ☑ Negative test

- The `_newBlockUpdater` is the same as the stored block-updater address.
  - ☑ Negative test

Function: `setDstChain(uint32 _dstEid, bool enabled)`

Configures which destination EIDs are enabled for sending messages. Only callable by the owner.

Inputs

- `_dstEid`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The destination endpoint ID to configure.
- `enabled`
  - **Control**: Full.
  - **Constraints**: Must not be the current state.
  - **Impact**: The new state to configure for the destination chain.

Branches and code coverage

Intended branches

- Successfully sets the destination EID enabled status.
  - ☑ Test coverage

Negative behavior

- Value equals the current state.
  - ☑ Negative test
- Caller is not the owner.
  - ☑ Negative test

Function: `setFeeManager(address feeManager_, bool enabled_)`

Enables and disables the ability of certain callers to call `setFee`. Callable by the owner only.

Inputs

- `feeManager_`
  - **Control**: Full.
  - **Constraints**: None.

    - **Impact**: The address to change the configuration of.
- `enabled_`
    - **Control**: Full.
    - **Constraints**: Must not be equal to the current state for the fee-manager address.
    - **Impact**: The new configuration value to use.

Branches and code coverage

Intended branches

- Successfully modifies the fee-manager configuration.
    - ✓ Test coverage

Negative behavior

- The `feeManager_` is a zero address.
    - ✓ Negative test
- The `enabled_` equals the current state for that address.
    - ✓ Negative test

Function: `setFee(uint32 _dstEid, uint256 _price)`

Configures the fee for a given destination chain. Only callable by the fee manager.

Inputs

- `_dstEid`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The destination endpoint ID to configure the fee for.
- `_price`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The fee to configure for the destination chain.

Branches and code coverage

Intended branches

- Successfully sets the fee for the destination chain.
    - ✓ Test coverage

Negative behavior

- Caller must be the fee manager.
  - ☑ Negative test

Function: `setReceiveView(address receiveLib_, address receiveLibView_)`

Configures the view receive library for the given receive library address.

Inputs

- `receiveLib_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The receive library to use as the key.
- `receiveLibView_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The view receive library to configure.

Branches and code coverage

Intended branches

- Successfully sets the view receive library for the given receive library address.
  - ☐ Test coverage

Negative behavior

- Addresses cannot be zero.
  - ☐ Negative test

Function: `setZKMptValidator(address _newZkMptValidator)`

Changes the zkMPT-validator contract address configuration. Callable by the owner only.

Inputs

- `_newZkMptValidator`
  - **Control**: Full.
  - **Constraints**: Cannot be zero or equal to the currently stored address.
  - **Impact**: The zkMPT validator contract to use.

Branches and code coverage

Intended branches

- Updates the zkMPT address configuration.
  - ✓ Test coverage

Negative behavior

- Caller is not the owner.
  - ✓ Negative test
- Reverts if the `_newZkMptValidator` is zero.
  - ✓ Negative test
- The `_newZkMptValidator` is the same as the stored zkMPT contract address.
  - ✓ Negative test

Function: `withdrawFeeAll(address payable _to)`

Withdraws all fees from all messaging libraries. Callable by the owner only.

Inputs

- `_to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The address to withdraw the fees to.

Branches and code coverage

Intended branches

- Successfully withdraws all fees from all messaging libraries.
  - ✓ Test coverage

Negative behavior

- No fees to withdraw.
  - ✓ Negative test
- Caller is not the owner.
  - ✓ Negative test

Function: `withdrawFee(address _messageLib, address payable _to)`

Withdraws fees for a single messaging library. Callable by the owner only.

Inputs

- `_messageLib`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The address to try to withdraw the fees from (not necessarily an enabled messaging library).
- `_to`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The address to withdraw the funds to.

Branches and code coverage

Intended branches

- Successfully withdraws fees from the messaging library at the address.
    - ☑ Test coverage

Negative behavior

- Caller is not the owner.
    - ☑ Negative test
- No fees to claim from the library.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was already deployed to several chains.

During our assessment on the scoped Polyhedra DVN contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature. Polyhedra Network acknowledged all findings and implemented fixes.

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.