Polyhedra zk light client on LayerZero

Smart Contract Security Assessment

June 26, 2023

Prepared for:

Tiancheng Xie and Abner Jia

Polyhedra

Prepared by:

Aaron Esau and Mohit Sharma

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 Polyhedra from June 13th to 16th, 2023. During this engagement, Zellic reviewed Polyhedra zk light client on LayerZero'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:

- How do the contracts validate public instances of proofs?
- Is the oracle's business logic handled correctly?
- How could a malicious actor compromise the security guarantees if admin keys were leaked?

1.2 Non-goals and Limitations

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

- Automatically generated code or MPT-parsing code
- 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.

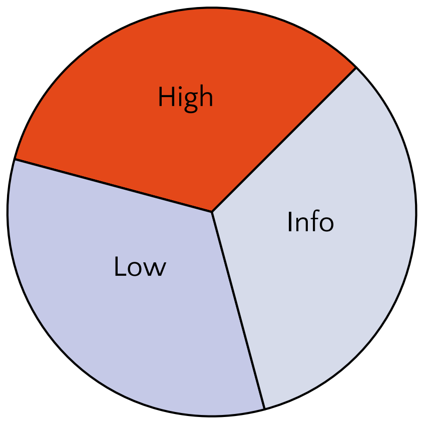
1.3 Results

During our assessment on the scoped Polyhedra zk light client on LayerZero contracts, we discovered three findings. No critical issues were found. One was of high impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Polyhedra'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	0
Low	1
Informational	1



2 Introduction

2.1 About Polyhedra zk light client on LayerZero

Polyhedra is building the next generation of infrastructure for Web3 interoperability by leveraging advanced zero-knowledge proof (ZKP) technology, a fundamental cryptographic primitive that guarantees the validity of data and computations while maintaining data confidentiality.

The Polyhedra Network team designed and developed Polyhedra zkLightClient technology, a cutting-edge solution built on LayerZero Protocol, providing secure and efficient cross-chain infrastructures for Layer 1 and Layer 2 interoperability.

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:

Polyhedra zk light client on LayerZero Contracts

Repository	https://github.com/zkBridge-integration-audit/zkBridge_review_Zellic
Version	zkBridge_review_Zellic: `29c763f7742f637751bd01acb10b203d0b6ec71` 4
Programs	• ZKMptValidator • ZkBridgeOracle
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 13, 2023	Kick-off call
June 13, 2023	Start of primary review period
June 16, 2023	End of primary review period

3 Detailed Findings

3.1 Lack of separation between ZK and traditional MPT verifiers

- **Target**: ZkBridgeOracle
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The oracle contract has the `updateMptHash` and `updateFpHash` external functions for submitting oracle proofs using the traditional MPT and ZK verifiers, respectively:

```
function updateMptHash(uint16 _sourceChainId, bytes32 _blockHash,
    bytes32 _receiptHash, address _userApplication) external {
    _updateHash(_sourceChainId, _blockHash, _receiptHash, _blockHash,
    _receiptHash, _userApplication);
}

function updateFpHash(uint16 _sourceChainId, bytes32 _blockHash,
    bytes calldata zkMptProof, address _userApplication) external {
    require(address(zkMptValidator)≠address(0),"ZkBridgeOracle:Not set
    zkMptValidator");
    IZKMptValidator.Receipt memory receipt
    = zkMptValidator.validateMPT(zkMptProof);
    _updateHash(_sourceChainId, _blockHash, receipt.receiptHash,
    receipt.logsHash, receipt.logsHash, _userApplication);
}
```

Either function can be called to submit a proof since both are external functions. The functions should generally be separated as each oracle is supposed to perform a single form of verification.

Impact

If a User Agent (UA) owner configured the Ultra Light Node (ULN) to set the oracle address to that of the deployed ZkBridgeOracle contract with the intent of using the ZK verifier, it would also be possible to submit an MPT proof—defeating the owner's intent.

This would let the attacker submit hashes using the traditional MPT verifier despite the owner expecting the ZK verifier to be used.

Recommendations

Consider separating the two verifier proof-submitting functions into separate contracts so that a ZK oracle cannot be used to submit an MPT proof and an MPT oracle cannot have a ZK proof be submitted to it.

Remediation

Polyhedra provided the following response to this finding:

> In our design context, both MPT and FP methods go through ZK verification. The function `updateMptHash` is utilized to upload `blockHash` and `receiptHash` to ULN and validates whether `blockHash` and `receiptHash` exist in our BlockUpdater. During the block update, ZK verification is involved in the BlockUpdater contract.
>
> On the other hand, `updateFpHash` is used to upload transaction logs to ULN. Initially, it goes through a ZkMPT check, then verifies the existence of the block, and eventually pushes the logs to ULN.
>
> These two functions are designed to submit different content, and one cannot be used to submit the data of the other. Given these system behaviors, we respectfully disagree with the recommendation as we don't see the need to separate these functions into different contracts in our case.

After an internal discussion, we agree that a relayer using the MPT proof library should only be interacted with via the `updateMptHash` or `batchUpdateMptHash` functions, and the FP proof library via the `updateFpHash` and `batchUpdateFpHash` functions.

However, we are still concerned about the potential possibility of type confusion attacks between the `receiptHash` and `blockHash` submitted by the functions. For example, if the `keccak256` of an FP-submitted packet happens to match a block hash, the relayer could change the validation library to MPTValidator01 and bypass oracle safety checks.

It is also possible to set the confirmation count on a `lookupHash` and `blockHash` that should be submitted through another function. Though doing this has no impact, the oracle's process for updating hashes should match the validation library for correctness reasons.

Additionally, LayerZero may release validation libraries in the future independently of Polyhedra that unintentionally enable type confusion attacks.

In general, it makes sense that one oracle has one validation method. Separating validation methods reduces the attack surface and mitigates an entire class of potential attacks.

3.2 Centralization risk in `setBlockUpdater`

- **Target**: ZkBridgeOracle
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The `setBlockUpdater` function in ZkBridgeOracle.sol allows the contract owner to modify the block updater for any source chain. The blockUpdater 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

This poses a centralization risk by enabling the admin to modify a blockUpdater to arbitrary logic and bypass the security guarantees of the contract.

Recommendations

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

Remediation

Polyhedra provided the following response to this finding:

> Regarding your concerns with the ZkBridgeOracle contract and specifically, the `setBlockUpdater` 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.

3.3 Test suite

- **Target**: ZkBridgeOracle, ZKMptValidator
- **Category**: Code Maturity
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the white paper. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Remediation

Polyhedra provided the following response to this finding:

Thank you for your meticulous review and recommendations regarding the expansion of our test suite.

We fully acknowledge the value of comprehensive testing and understand the urgency for improvement. However, developing a thorough test suite, due to the complexity of our contracts, demands careful planning and some time.

Rest assured, we are actively working to enhance our testing procedures to include all contracts and scenarios as you've outlined. We aim to implement these adjustments soon, thereby increasing code reliability, efficiency, and advancing overall code maturity.

4 Discussion

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

4.1 Batch update gas optimization via aggregation circuit

The batch update function `batchUpdateFpHash` in ZkBridgeOracle.sol expects as input a proof for each hash in the batch. All these proofs are verified separately by the verifier contract.

```
function batchUpdateFpHash(uint16[] calldata _sourceChainIds, bytes32[]
    calldata _blockHashes, bytes[] calldata zkMptProofs, address[]
    calldata _userApplications) external {
    require(address(zkMptValidator)≠address(0),"ZkBridgeOracle:Not set
    zkMptValidator");
    require(_sourceChainIds.length == _blockHashes.length,
    "ZkBridgeOracle:Parameter lengths must be the same");
    require(_sourceChainIds.length == zkMptProofs.length,
    "ZkBridgeOracle:Parameter lengths must be the same");
    require(_sourceChainIds.length == _userApplications.length,
    "ZkBridgeOracle:Parameter lengths must be the same");
    IZKMptValidator.Receipt memory receipt;
    for (uint256 i = 0; i < _sourceChainIds.length; i++) {
        receipt = zkMptValidator.validateMPT(zkMptProofs[i]);
        _updateHash(_sourceChainIds[i], _blockHashes[i],
    receipt.receiptHash, receipt.logsHash, receipt.logsHash,
    _userApplications[i]);
    }
}
```

Implementing an aggregation circuit in the future for these proofs to roll up a number of them into a single verification could help greatly reduce the cost of batch updates.

Polyhedra has acknowledged this optimization and plan to implement it in a future update.

5 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.

Our threat model was conducted on commit 29c763f7, which represents the codebase at a specific point in time. Therefore, it is possible that the absence of a particular test in our report may not reflect the current state of the test suite.

5.1 Module: ZKMptValidator.sol

Function: `validateMPT(byte[] _proof)`

Verifies an MPT proof and returns a receipt if the proof is valid

Inputs

- `_proof`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A. Assumed to be a valid proof payload.
  - **Impact**: Contains proof data to be validated by verifier.

Branches and code coverage (including function calls)

Intended branches

- Receipt is generated correctly for valid proof.
  ☐ Test Coverage

Negative behavior

- Reverts if proof is invalid.
  ☐ Test Coverage
- Reverts if `proofData.publicInputs.length` < 4.
  ☐ Test Coverage

- Reverts if `_proof` deserialization fails.
  ☐ Test Coverage

Function: `_hashInput(uint256[] _inputs)`

Concatenates the first four elements in `_inputs` and returns the hash.

Inputs

- `_inputs`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A; `_inputs` is assumed to be of length `>=` 4.
  - **Impact**: Used to compress public instances of a proof.

Branches and code coverage (including function calls)

Intended branches

- Hash computed correctly if `_inputs.length >=` 4.
  ☐ Test Coverage

Negative behavior

- Reverts if `_inputs.length <` 4.
  ☐ Test Coverage

5.2 Module: ZkBridgeOracle.sol

Function: `assignJob(uint16 _dstChainId, uint16 _proofType, uint64 _outboundBlockConfirmation, address _userApplication)`

Query price and assign jobs at the same time.

Inputs

- `_dstChainId`
  - **Control**: Fully controlled by caller.
  - **Constraints**: `supportedDstChain[_proofType][_dstChainId]` must be true; in other words, the supplied proof type must be supported for the destination ID.
  - **Impact**: The destination endpoint identifier.
- `_proofType`
  - **Control**: Fully controlled by caller.

  - **Constraints**: `supportedDstChain[_proofType][_dstChainId]` must be true; in other words, the supplied proof type must be supported for the destination ID.
  - **Impact**: Specifies the proof type to be relayed.
- `_outboundBlockConfirmation`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: Block confirmation delay before relaying blocks.
- `_userApplication`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The source sending contract address.

Branches and code coverage (including function calls)

Intended branches

- Emit `OracleNotified` if `_proofType` and `_dstChainId` are valid.
  ☐ Test Coverage

Negative behavior

- Revert if `_proofType` is not supported for `_dstChainId`.
  ☐ Test Coverage
- Revert if user application ULN is not supported.
  ☐ Test Coverage

Function: `updateMptHash(uint16 _sourceChainId, bytes32 _blockHash, bytes32 _receiptHash, address _userApplication)`

Used for submitting oracle proofs using the MPT verifiers.

Inputs

- `_sourceChainId`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: Sent to LZ as the source.
- `_blockHash`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A. Assumed to be verified against `_receiptHash` by the calling function.

    - **Impact**: Block hash to be delivered.
- `_receiptHash`
    - **Control**: Fully controlled by caller.
    - **Constraints**: Verified with `blockUpdater`.
    - **Impact**: Receipt for the block hash.
- `_userApplication`
    - **Control**: Fully controlled by caller.
    - **Constraints**: Must have a valid receiver library.
    - **Impact**: Used to get ultra light node.

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

Used to withdraw the accrued fee in ultra light node.

Inputs

- `_to`
    - **Control**: Fully controlled by caller.
    - **Constraints**: N/A.
    - **Impact**: Fee receiver.
- `_amount`
    - **Control**: Fully controlled by caller.
    - **Constraints**: Must be less than existing fee balance.
    - **Impact**: Withdrawal amount.

Branches and code coverage (including function calls)

Intended branches

- Total fee withdrawn is equal to `_amount`.
    - [ ] Test Coverage

Negative behavior

- Revert if `_to` is 0.
    - [ ] Test Coverage
- Revert if `_amount` is more than available balance.
    - [ ] Test Coverage

Function: `_updateHash(uint16 _sourceChainId, byte[32] _blockHash, byte[32] _receiptHash, byte[32] _lookupHash, byte[32] _blockData, address _userApplication)`

Delivers block data if block confirmation exists for `_blockHash` and `_receiptHash`.

Inputs

- `_sourceChainId`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: Sent to LZ as the source.
- `_blockHash`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A. Assumed to be verified against `_receiptHash` by the calling function.
  - **Impact**: Block hash to be delivered.
- `_receiptHash`
  - **Control**: Fully controlled by caller with possible additional constraints by calling function.
  - **Constraints**: Verified with `blockUpdater`.
  - **Impact**: Receipt for the block hash.
- `_lookupHash`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: Used by LZ to lookup block data.
- `_blockData`
  - **Control**: Fully controlled by caller with possible additional constraints by calling function.
  - **Constraints**: N/A. Assumes already confirmed against block hash.
  - **Impact**: Delivered to LZ.
- `_userApplication`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Must have a valid receiver library.
  - **Impact**: Used to get ultra light node.

Branches and code coverage (including function calls)

Intended branches

- Block data broadcasted successfully if receipt is valid.

    - [ ] Test Coverage

Negative behavior

- Revert if `blockUpdater` is 0.
    - [ ] Test Coverage
- Revert if receipt is invalid.
    - [ ] Test Coverage

6 Audit Results

At the time of our audit, the audited code was not deployed to Layer Zero.

During our assessment on the scoped Polyhedra zk light client on LayerZero contracts, we discovered three findings. No critical issues were found. One was of high impact, one was of low impact, and the remaining finding was informational in nature. Polyhedra 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 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.