Zellic

LayerZero Core

Smart Contract Security Assessment

April 15, 2022

Prepared for:

Ryan Zarick and Isaac Zhang

LayerZero Labs

Prepared by:

Katerina Belotskaia and Aaron Esau

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About LayerZero Core

LayerZero is an Omnichain Interoperability Protocol designed for lightweight message passing across chains. LayerZero provides authentic and guaranteed message delivery with configurable trustlessness. The protocol is implemented as a set of gas-efficient, non-upgradable smart contracts.

LayerZero Core refers to the core contracts behind the LayerZero omnichain network.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

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

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than 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 numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

LayerZero Core Contracts

Repository	GitHub
Versions	`43ab0aed0fbcd123bcac3d089e74898e25b86c0a`
Type	Solidity
Platform	EVM-compatible

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 3 person-week. The assessment was conducted over the course of 2 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

April 4, 2022	Kick-off call
April 4, 2022	Start of primary review period
April 14, 2022	End of primary review period
TODO	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for LayerZero Labs from April 4th to April 15th, 2022 on the scoped contracts and discovered 3 findings. Fortunately, no critical issues were found. We applaud LayerZero Labs for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of LayerZero Core.

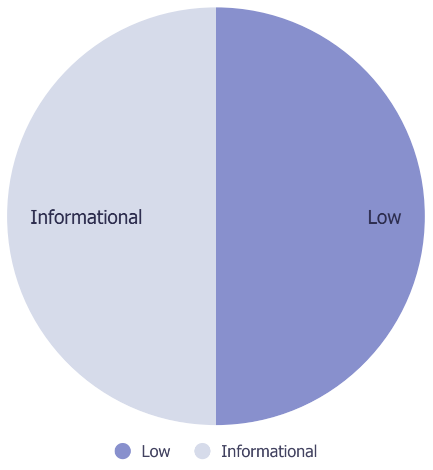
Of the 4 findings, 2 were of low severity, and the remaining findings were informational in nature. Additionally, Zellic recorded its notes and observations from the audit for LayerZero Labs's benefit at the end of the document.

Zellic thoroughly reviewed the LayerZero Core codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account LayerZeros' threat model, we focused heavily on issues that would break core invariants such as executing payloads without the agreement of both parties (oracle and relayer), or executing them out of order causing desynchronization between source and destination chains.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	2



3 Detailed Findings

3.1 Using non-contract address as destination blocks future messages

- **Target**: Endpoint
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

An improperly-configured user application (UA) can permanently block itself from communicating with an endpoint by simply sending a message to a UA address that is not a contract.

If a UA sends a message with a destination UA address that is not a contract, the following try/catch statement does not catch the exception (as the control structure only catches failures in an external call) causing a revert on the destination chain:

```
try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(_srcChainId
    , _srcAddress, _nonce, _payload) {
    // success, do nothing, end of the message delivery
} catch (bytes memory reason) {
    // revert nonce if any uncaught errors/exceptions if the ua chooses
    the blocking mode
    storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64(
    _payload.length), _dstAddress, keccak256(_payload));
    emit PayloadStored(_srcChainId, _srcAddress, _dstAddress, _nonce,
    _payload, reason);
}
```

If the destination chain reverts, the source chain's nonce remains incremented by 1 while the destination chain's nonce is unchanged.

Impact

When the nonces are desynchronized, no messages can be sent to any destination UA address because the destination endpoint assumes the messages are out of order.

Endpoints key the nonce map with the source chain ID and source UA address—

meaning this issue can only be exploited as self-denial-of-service.

Recommendation

Add a check to ensure the destination UA is a valid contract address before attempting to execute its `lzReceive` function. If the contract address is invalid, the endpoint should route the message to a default contract address that discards the message to keep the nonces synchronized.

Remediation

The issue was also discovered in parallel by LayerZero and a fix will be released with UltraLightNode version 2.

3.2 Out-of-bounds read in __getPrices

- **Target**: Relayer
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `__getPrices` function uses the `MLOAD` instruction to read `dstNativeAmt` from `_adapterParameters+66` when `txType == 2`:

```
if (txType == 2) {
    uint dstNativeAmt;
    assembly {
        dstNativeAmt := mload(add(_adapterParameters, 66))
    }
    require(dstConfig.dstNativeAmtCap ≥ dstNativeAmt, “Relayer:
    dstNativeAmt too large”);
    totalRemoteToken = totalRemoteToken.add(dstNativeAmt);
}
```

At the start of the function, it checks that the size of `_adapterParameters` is either 34 bytes or greater than 66 bytes:

```
require(_adapterParameters.length == 34 || _adapterParameters.length >
    66, “Relayer: wrong _adapterParameters
    size”);
```

Because the assertion allows an `_adapterParameters` of a size smaller than the offset added to the size of the memory read, the read could potentially be out of bounds.

Impact

There is no direct security impact of this instance of out-of-bounds read. However, this code pattern allows undefined behavior and is potentially dangerous. In the past, even low-level vulnerabilities have been chained with other bugs to achieve critical security compromises.

Recommendation

The size of a `uint` (which is internally a `uint256`) is 32 bytes. So, the branch that uses the `MLOAD` instruction should require that the size of `_adapterParameters` is greater than or equal to the read size added to offset, or 98 bytes (32+66).

Remediation

The issue has been acknowledged by LayerZero.

3.3 Messaging library provides a function to renounce ownership

- **Target**: UltraLightNode
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The messaging library, UltraLightNode (ULN), implements Ownable which provides a method named `renounceOwnership` that removes the current owner (Reference). This is likely not a desired feature of the ULN.

Impact

If `renounceOwnership` were called, the contract would be left without an owner.

Recommendation

Override the `renounceOwnership` function:

```
function renounceOwnership() public {
    revert(“This feature is not available.”);
}
```

Remediation

The issue has been acknowledged by LayerZero.

3.4 Race condition may enable bypass of library address check

- **Target**: Endpoint
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: High

Description

An obscure scenario may allow a malicious actor to bypass the ULN library address check in `receivePayload`:

```
// authentication to prevent cross-version message validation
// protects against a malicious library from passing arbitrary data
if (uaConfig.receiveVersion == DEFAULT_VERSION) {
    require(defaultReceiveLibraryAddress == msg.sender, "LayerZero:
    invalid default library");
} else {
    require(uaConfig.receiveLibraryAddress == msg.sender, "LayerZero:
    invalid library");
}
```

This behavior can only happen in a very specific situation:

1. The originally-configured ULN (hereinafter referred to as the "untrusted ULN") must send a message (the "untrusted message") to the Endpoint that passes all checks in the `receivePayload` function but fails for any reason—causing the endpoint to store it.

2. A new ULN must be configured (the "trusted ULN"). This may happen if, for example, the untrusted ULN was found to be malicious or vulnerable.

3. Even though the trusted ULN is now configured—and the untrusted ULN cannot send further messages—the malicious ULN can bypass the ULN address checks and re-send the untrusted message using the `retryPayload` function.

It is important to understand that the untrusted message must be sent *before* the configured ULN address is changed; that is, this behavior is unlikely to be exploited unless it is advantageous to a malicious actor to wait before sending a message (e.g. if the attacker wants the message to only be sent after the configured ULN address is changed, or only after some time has passed).

Because the scenario is incredibly specific and obscure, Zellic considers the likelihood to be low.

Impact

This behavior may lead to the endpoint forwarding an "untrusted" message (i.e. a message that the currently-configured ULN considers trustworthy) to the user application. The impact varies depending on the user application; however, because it is possible the behavior could lead to a negative financial impact, Zellic considers the impact to be high.

Recommendation

A possible solution to this flaw is to clear the stored message entries in `storedPayload` whenever the `uaConfig`'s library address or the `defaultReceiveLibraryAddress` (whichever address should be used, depending on the value of `uaConfig.receiveVersion`) is changed. The `forceResumeReceive` function clears only one entry; however, all entries should be considered untrusted and removed.

However, this solution introduces a potential issue where stored messages are not delivered during an upgrade (i.e. when the "untrusted ULN" is actually a legitimate, but old library).

Instead, the library upgrade process could clear `storedPayload` (e.g. by calling `forceResumeReceive`) only if the upgrader determines the old library is untrustworthy. The `storedPayload` should be cleared immediately after changing the configured ULN address (within the same transaction) to avoid leaving a window where a malicious actor could call `retryPayload`.

Remediation

The issue has been acknowledged by LayerZero and a fix may be implemented in the future.

4 Discussion

The purpose of this section is to document miscellaneous observations the Zellic team made during the assessment.

4.1 UltraLightNode Version 2

Please note that as of the time of this writing, LayerZero Labs plans to release a ULNv2 with the following changes:

Interface Changes

- `ILayerZeroRelayerV2` and `ILayerZeroOracleV2`:
  - Merges `getPrice` and `notifyRelayer/notifyOracle` interfaces into an `assignJob` interface. This change saves gas by avoiding one contract call per notify function.
  - Adds a new `getPrice` function.
  - Adds a withdraw fee function.
- `ILayerZeroUltraLightNodeV2`:
  - Simplifies the withdraw NativeFee interface.

Miscellaneous Changes

- Trims the BlockData size from 2 `bytes32` to 1. Saves gas in oracle's `updateHash`.
- Redefines the `Packet` event to include new fields in the `encodedPacket` such as `PACKET_VERSION` to make the packet translation safer and more efficient.
- Adds more assertions to make the ULNv2 safer in general.
- Fixes the issue where cross-chain contract calls to non-contract addresses cause the nonce to be out of sync, blocking further messages. See Zellic's finding 3.1 for more information.

Others

Zellic performed an informal review of the changes included in ULNv2; however, the scope of this report only includes ULNv1 and its associated contracts.

4.2 Notes on in-scope contracts

Variable naming standards inconsistency

In `Endpoint.sol`, the variables `_send_entered_state` and `_receive_entered_state` should be named `_sendEnteredState` and `_receiveEnteredState` (i.e. camel case), respectively. These variables, though internal, are not constant.

Warning about pragma locked versions

Many of the files lock pragma to Solidity versions that do not check math operations by default. So, it is critical that when future developers write code that performs math operations, they use the SafeMath library, write the code in a way that is safe from overflows and underflows, or be audited for secure operation. At the time of this assessment by Zellic, all math operations are safe from overflows and underflows.

Additionally, consider locking the pragma to ensure that experimental compilers or compilers that lack recently-added optimizations or security updates cannot be used to compile the contracts.

Relayer/oracle trust dependence

As stated in the LayerZero protocol whitepaper and on the documentation website, the chosen relayer and oracle must be controlled by independent entities; otherwise, they could conspire to falsify messages.

Endpoint may block messages to all UAs if one transaction reverts

If any destination user application (UA)'s `lzReceive` function reverts, the destination endpoint will prevent all future messages from the source chain and address from being delivered—regardless of the destination UA—until the message is retried successfully (`retryPayload`) or `forceResumeReceive` is called.

The LayerZero Labs team noted that this is the intended design and that this is the only way to ensure messages delivered in the correct order on the entire chain. UAs may be built on top of the endpoint if this behavior is not desired.

4.3 Notes on out-of-scope contracts

`packet.ulnAddress` must never be `0x0`

The proof validation library (prooflib) was not in scope for this assessment. However, proper implementations of the prooflib must not be able to be manipulated into returning a packet whose ulnAddress is 0x0; otherwise, malicious relayers may be able to bypass the following assertion in UltraLightNode:

```
// (e) assert that the packet was emitted by the source ultra light node
require(ulnLookup[_srcChainId] == _packet.ulnAddress, “LayerZero:
    _packet.ulnAddress is invalid”);
```

If _srcChainId does not exist in ulnLookup, the operation ulnLookup[_srcChainId] will output 0x0.

The LayerZero Labs team stated that at this time, the intended design of prooflib does not allow the validateProof function to return a packet with a ulnAddress of 0x0.

notifyRelayer is not onlyULN

The notifyRelayer function in Relayer.sol does not restrict the caller to be the ULN only. So, it is critical that any code written in that function in the future either restrict the caller or treat all function inputs as untrusted.

The LayerZero Labs team acknowledged that this function does not restrict the caller, stating that the function may have later use or be implemented in third-party libraries for metadata tracking or accounting purposes.

Demo contracts contain setCaller but do not check caller

The demo user application contracts—which may be used as templates—at mocks/OmniCounter.sol and mocks/PingPong.sol contain a function called setConfig that allows the caller to modify the ULN's configuration without checking who the caller is:

```
function setConfig(
    uint16, /*_version*/
    uint16 _chainId,
    uint _configType,
    bytes calldata _config
) external override {
    endpoint.setConfig(endpoint.getSendVersion(address(this)), _chainId,
    _configType, _config);
}
```

Although the lack of restriction on the caller presents no immediate security concern to LayerZero, these files may be used as templates and should be secure by default. Any contract containing this setConfig function allows attackers to control the config and potentially change configuration settings such as the relay or oracle address.

Additionally, the following functions do not restrict the caller, but modify endpoint or UA configuration:

- `OmniCounter.setSendVersion`
- `OmniCounter.setReceiveVersion`
- `OmniCounter.setOracle`
- `OmniCounter.setInboundConfirmations`
- `OmniCounter.setOutboundConfirmations`
- `PingPong.setSendVersion`
- `PingPong.setReceiveVersion`

Demo contracts contain `notifyOracle` functions that are not `onlyULN`

The demo user application contracts—which may be used as templates—at `mocks/LayerZeroOracleMock.sol` and `mocks/LayerZeroOracleBadMock.sol` do not restrict the caller to the ULN as they should in their `notifyOracle` functions:

```
function notifyOracle(uint16 _dstChainId, uint16 _outboundProofType,
    uint64 _outboundBlockConfirmations) external override {
    emit OracleNotified(_dstChainId, _outboundProofType,
    _outboundBlockConfirmations);
}
```

Although the lack of restriction on the caller presents no immediate security concern to LayerZero, these files may be used as templates and should be secure by default. Any oracle contract using the code provided in these two files will allow allow attackers to control the data sent to the off-chain oracle.

Zellic recommends restricting the caller as much as possible. For example, `chainlink/ChainlinkOracleClient.sol` properly restricts the caller to the ULN using the function modifier `onlyULN`:

```
modifier onlyULN() {
    require(msg.sender == address(uln), “OracleClient: caller must be
    LayerZero.”);
    _;
}

// ...
```

```
function notifyOracle(uint16 _dstChainId, uint16 _outboundProofType,
    uint64 _outboundBlockConfirmations) external override onlyULN {
//  ...
```

Multiple functions declared as public that could be external

Many of these functions are from demo files, but it is worth noting that the following functions are declared as public but could be declared as external to save gas per operation:

- `Relayer.initialize(address)`
- `ChainlinkOracleClient.withdrawTokens(address,address,uint256)`
- `ChainlinkOracleClient.setJob(uint16,address,bytes32,uint256)`
- `ChainlinkOracleClient.setDeliveryAddress(uint16,address)`
- `ChainlinkOracleClient.fulfillNotificationOfBlock(bytes32,bytes32)`
- `ChainlinkOracleClient.isApproved(address)`
- `LayerZeroOracleBadMock.isApproved(address)`
- `LayerZeroOracleMock.isApproved(address)`
- `MockLinkToken.transferAndCall(address,uint256,bytes)`
- `MockLinkToken.mint(address,uint256)`
- `LayerZeroOracleBadMock.withdraw(address,uint256)`
- `LayerZeroOracleBadMock.setJob(uint16,address,bytes32,uint256)`
- `LayerZeroOracleBadMock.setDeliveryAddress(uint16,address)`
- `LayerZeroOracleMock.withdraw(address,uint256)`
- `LayerZeroOracleMock.setJob(uint16,address,bytes32,uint256)`
- `LayerZeroOracleMock.setDeliveryAddress(uint16,address)`
- `MockToken.mint(address,uint256)`
- `OmniCounter.getCounter()`
- `OmniCounter.incrementCounter(uint16,bytes)`
- `OmniCounter.incrementCounterWithPayload(uint16,bytes,bytes)`
- `OmniCounter.incrementCounterWithAdapterParamsV1(uint16,bytes,uint256)`
- `OmniCounter.incrementCounterWithAdapterParamsV2(uint16,bytes,uint256,uint256,address)`
- `OmniCounter.incrementCounterMulti(uint16[],bytes[],address)`