Zellic

Endpoint V2

Smart Contract Security Assessment

December 13, 2023

Prepared for:

Ryan Zarick and Isaac Zhang

LayerZero Labs

Prepared by:

Jasraj Bedi 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 contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for LayerZero Labs from July 26th to August 22nd, 2023. Further patches were reviewed till commit c71f996b, dating up to December 13, 2023. During this engagement, Zellic reviewed Endpoint V2'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:

- Can a malicious OApp cause other OApps' paths to be blocked?
- Can the protocol censor messages?
- Can malicious DVNs permanently block an Oapp?
- Can messages be forged for an arbitrary source address?
- What centralization risks does Endpoint V2 have?

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

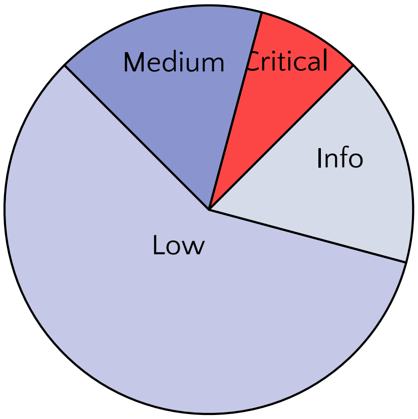
1.3 Results

During our assessment on the scoped Endpoint V2 contracts, we discovered 12 findings. One critical issue was found. Two were of medium impact, seven were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for LayerZero Labs's benefit in the Discussion section (5) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	2
Low	7
Informational	2



2 Introduction

2.1 About Endpoint V2

LayerZero is a generic messaging protocol. Endpoint V2 is the second version of the endpoints — the proxy layer to the message libraries — in the core of the LayerZero protocol.

For more information, see section 3 for a more in-depth description of LayerZero Endpoint V2's properties.

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 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.

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 (5) 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:

Endpoint V2 Contracts

Repository	https://github.com/LayerZero-Labs/monorepo
Versions	monorepo: `b68547b05ac540944697b220163291aeb69f6e92` monorepo: `b68547b05ac540944697b220163291aeb69f6e92 (patch review)`

Programs	protocol/contracts/libs/Errors.sol protocol/contracts/libs/CalldataBytesLib.sol protocol/contracts/libs/AddressCast.sol protocol/contracts/libs/SafeCall.sol protocol/contracts/messagelib/libs/BitMaps.sol protocol/contracts/messagelib/libs/PacketV1Codec.sol protocol/contracts/messagelib/BlockedMessageLib.sol protocol/contracts/EndpointV2.sol protocol/contracts/MessageLibManager.sol protocol/contracts/MessagingChannel.sol protocol/contracts/MessagingComposer.sol protocol/contracts/MessagingContext.sol messagelib/contracts/uln/libs/UlnOptions.sol messagelib/contracts/uln/libs/VerifierOptions.sol messagelib/contracts/uln/uln301/MessageLibBaseE1.sol messagelib/contracts/uln/uln301/TreasuryFeeHandler.sol messagelib/contracts/uln/uln301/UltraLightNode301.sol messagelib/contracts/uln/uln302/UltraLightNode302.sol messagelib/contracts/uln/MultiSig.sol messagelib/contracts/uln/UlnBase.sol messagelib/contracts/uln/UlnConfig.sol messagelib/contracts/uln/VerifierNetwork.sol messagelib/contracts/MessageLibBase.sol messagelib/contracts/MessageLibBaseE2.sol messagelib/contracts/OutboundConfig.sol messagelib/contracts/Worker.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 four person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jasraj Bedi, CTO
jazzy@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, CTO
jazzy@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 26, 2023	Start of primary review period
August 22, 2023	End of primary review period
December 13, 2023	Patch review of new changes

3 Details About Endpoint V2

LayerZero is a generic messaging protocol. Endpoint V2 is the second version of the endpoints — the proxy layer to the message libraries — in the core of the LayerZero protocol.

3.1 Architecture changes from Endpoint V1

The following table documents the architectural changes between Endpoint V1 and Endpoint V2.

	Endpoint V1	**Endpoint V2**
Validation	Uses a relayer and oracle to submit the necessary information for delivery.	Abstracts the entities required to validate a message for delivery to verifiers whose addresses can be an EOA or a contract. This gives the OApp the flexibility to decide how it wants to perform validation.
Verification	Performed by the proof library in the Ultra Light Node (ULN), using the Merkle-Patricia trie for proof-of-message inclusion (delivered by the relayer).	The verification is performed independently by the verifiers. There are just a set of oracles that can perform their independent verification and reach consensus on-chain An example verifier is the VerifierNetwork contract, which implements a multi-sig verifier. The admin calls `VerifierNetwork.execute` with the quorum of signatures, signaling the ULN to verify and/or deliver the message

Delivery & execution	Done atomically by default. See the ULN's call to `receivePayload`, which executes the message based on delivery to the ULN. Note that technically a future ULN could separate delivery from execution by not immediately calling `Endpoint.receivePayload`.	Separated - a payload hash can be delivered (i.e., fully verified and stored in the endpoint) without being executed (sent to the OApp). From the endpoint's perspective, the message is delivered upon call to deliver and executed upon call to `lzReceive`.
Failed execution handling	If the message execution fails, the endpoint stores the payload hash so that it can be executed again in the future while still incrementing the nonce. See `Endpoint.receivePayload`'s use of `storedPayload`.	The payload hash is stored when delivering the message. Then, the payload hash is cleared before execution. If execution fails, the payload hash is restored. Because the payload hash is stored upon delivery, there is no need to store it separately if execution fails. Also, messages can execute out of order by default. See `EndpointV2.lzReceive`.

Message ordering	Delivery and execution order is enforced by endpoint (note that this is an atomic operation in V1). Messages cannot be delivered and executed out of order. See `Endpoint.receivePayload`'s tracking of `inboundNonce`. LayerZero V1 provides a `NonblockingLzApp`, which implements additional logic to allow out-of-order execution, but messages are always ordered at the endpoint level. The additional logic essentially separates delivery from execution.	Message execution ordering is not enforced by the endpoint; rather, it is left up to the OApp to enforce. See `OApp.acceptNonce`, which provides flexibility for enforcing nonce ordering. OApps that choose to enforce message execution ordering should implement a method to skip a nonce so that inexecutable messages not cause the protocol to come to a halt. Note that this does introduce centralization risk because it would then be possible to censor specific messages. Messages can be delivered to the endpoint out of order, but a message can only be executed if all messages with an equal or lower nonce value have already been delivered. This ensures that if a message is not delivered, the protocol's message execution comes to a halt. That is, it does not prevent censorship, but it adds a higher cost to a protocol's censorship.
Message origin auth	Enforced by the user application (UA) for sending and receiving. See `LzApp.lzReceive`.	Enforced by the OApp for receiving only. See `OApp.lzReceive`.

3.2 Risk of collusion

In LayerZero V1, the soundness of the protocol depended on two independent actors (the relayer and oracle) providing correct information to the receiver chain. If only one entity provided false, malicious information, the UA deployer could change the entity before the protocol would be harmed. But if both entities provided false information, false messages could be delivered before the UA deployer could change the configuration.

> Any protocol that relies on any number of entities to provide true information — regardless of how and where it sources the information — carries this risk. It is inherent to the design, not the protocol.

LayerZero V2 reduces this risk by abstracting the verifier entities required to deliver a message. There can be more than two entities as well. The new verification library provides configuration to allow any EOA or contract to be a verifier, which gives OApps the flexibility to decide how they want to perform validation.

Additionally, the verification library provides a verifier network component (the VerifierNetwork contract), which can be configured as a verifier. It acts as a multi-sig for a quorum of any number of entities (configured by the deployer) so that message approvals (i.e., the act of a verifier approving of a message) can be done off chain by signing a specific payload, and the signatures can be submitted in a single transaction.

> In general, the more entities required to assert the validity of a message, the lower the risk of a collusion attack.

3.3 Risks of default configurations

Zellic would like to clarify the default configurations aspect of LayerZero V2's threat model for those who are not intimately familiar with the protocol.

We want to dispel any unreasonable concerns about centralization risks but plainly document any true risks for the benefit of protocol developers and users.

Types of configuration

LayerZero V2 provides the following general types of configuration to OApps:

- **Messaging libraries:** OApps' messaging library choices are restricted to those approved by LayerZero Labs. Note that messaging libraries may have specific configuration options such as the number of inbound confirmations required to deliver a message.
  - **Send library:** Responsible for fee calculation and interacting with off-chain entities.
  - **Receive library:** Responsible for message validation (i.e., ensuring all proper verifiers approve of the message).
- **Verifiers:** In the receive library, the verifiers can be configured to some extent depending on the library implementation. Verifiers are responsible for ensuring packets are valid, but because they are abstracted, the logic or process for determining what constitutes a valid packet is up to the verifier (i.e., it is configurable). In all receive libraries as of the time of this writing, the OApp may manually configure the verifiers to whatever they desire.
  - **Required verifiers:** All of these types of configured verifiers must approve of the packet, or the packet will be rejected.
  - **Optional verifiers:** A quorum of these types of configured verifiers must approve of the packet, or the packet will be rejected. Note that the quorum number is also configurable by the OApp.

Abilities of LayerZero Labs

LayerZero Labs has the following abilities and limitations relating to configuration:

- All configuration options have defaults that can be set by LayerZero Labs at any time.

  > For all configuration options, LayerZero Labs does not have the ability to override user-set configuration. However, if the values are unset by the user, the LayerZero Labs–specified defaults — which can be changed at any time — will be used.

- LayerZero Labs has the ability to add permitted messaging libraries at any time and may permit malicious libraries. However, as previously mentioned, user configuration overrides default configuration. If the user has configured a specific library version, LayerZero Labs does not have the ability to change the OApp's receive library to a malicious one.

- LayerZero Labs does not have the ability to remove messaging libraries. An implication of this fact is that it is not possible for LayerZero Labs to deny service by disabling or preventing execution of a user-configured library.

- Contracts are immutable; that is, core LayerZero V2 contracts cannot be upgraded by LayerZero Labs or any entity.

Recommendations to OApp deployers

Based on these abilities of LayerZero Labs, if an OApp does not desire a specific custom configuration, we recommend fixing the configuration to the current default configuration at the time of deployment.

Note that the purpose of LayerZero Labs's ability to change the default configuration is to make the protocol future-proof without upgradability. As time passes, vulnerabilities or functionality bugs could potentially be discovered in the messaging libraries. Similarly, new technology may be developed to increase the efficiency of verification (i.e., gas savings).

To that end, we recommend that OApp deployers evaluate their present configuration and LayerZero Labs's new default configuration whenever their default configuration changes.

4 Detailed Findings

4.1 Zero confirmations lead to arbitrary payload execution

- **Target**: ReceiveUlnBase
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The Oapp can configure requiredConfirmations to 0 on the receiving side to allow quicker message delivery. This should be okay as long as the Oapp understands the risk of 0 confirmation transactions.

```
function _verified(
    address _dvn,
    bytes32 _headerHash,
    bytes32 _payloadHash,
    uint64 _requiredConfirmation
) internal returns (bool verified) {
    uint64 confirmations = hashLookup[_headerHash][_payloadHash][_dvn];
    // return true if the dvn has signed enough confirmations
    verified = confirmations ≥ _requiredConfirmation;
    delete hashLookup[_headerHash][_payloadHash][_dvn];
}
```

There exists an edge case in `_verified` where the `confirmations` default to 0 on an empty slot, so the `verified` is always true

Impact

This would allow complete forgery of messages as every message would be considered valid

Recommendations

We would recommend storing a flag along with confirmations in `hashLookup`. This would prevent the default value of 0 to be considered valid number of confirmations

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 32981204.

4.2 Overlapping DVNs may lead to unverifable messages

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

Description

To optimize for gas refunds, confirmations are deleted from `hashLookup` after being read.

```
function _verified(
    address _dvn,
    bytes32 _headerHash,
    bytes32 _payloadHash,
    uint64 _requiredConfirmation
) internal returns (bool verified) {
    uint64 confirmations = hashLookup[_headerHash][_payloadHash][_dvn];
    // return true if the dvn has signed enough confirmations
    verified = confirmations >= _requiredConfirmation;
    delete hashLookup[_headerHash][_payloadHash][_dvn];
}
```

In the scenario where a DVN is part both the required DVNs and optional DVNs, the confirmations from the DVN will be falsely deleted before they're read. This will cause the DVNs confirmation to not count as part of the optional DVN threshold.

Impact

Messages that should've been verifiable are falsely not verified.

Recommendations

We would recommend deleting the confirmations after the optional DVN lookup.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 03244a85.

4.3 Potential reentrancy in `lzReceive` function

- **Target**: EndpointV2
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `lzReceive` function deletes the payload hash prior to executing a delivered message to the receiver, thereby mitigating the risk of a reentrancy attack. In instances where message delivery fails, the hash is reinstated for resending. Subsequently, an external call is made to refund the native fee to the caller.

```
function lzReceive(
        Origin calldata _origin,
        address _receiver,
        bytes32 _guid,
        bytes calldata _message,
        bytes calldata _extraData
    ) external payable returns (bool success, bytes memory reason) {
        // clear the payload first to prevent reentrancy, and then execute
    the message
        bytes32 payloadHash = _clearPayload(_origin, _receiver,
    abi.encodePacked(_guid, _message));
        (success, reason) = _safeCallLzReceive(_origin, _receiver, _guid,
    _message, _extraData);

        if (success) {
            emit PacketReceived(_origin, _receiver);
        } else {
            // if the message fails, revert the clearing of the payload
            _inbound(_origin, _receiver, payloadHash);

            // refund the native fee if the message fails to prevent the
    loss of fund
            if (msg.value > 0) {
                (bool sent, ) = msg.sender.call{value: msg.value}(””);
                require(sent, Errors.INVALID_STATE);
            }

            emit LzReceiveFailed(_origin, _receiver, reason);
        }
```

```
  }
```

During this second external call, the caller may reenter and execute the message correctly, as the `payloadHash` has been restored prior to this call.

Impact

Initially, the `PacketReceived` event will be emitted following successful execution. However, the `LzReceiveFailed` event will also be emitted for the same packet within the same transaction, but in an incorrect order.

Recommendations

Restore the hash after the external call:

```
function lzReceive(
    Origin calldata _origin,
    address _receiver,
    bytes32 _guid,
    bytes calldata _message,
    bytes calldata _extraData
  ) external payable returns (bool success, bytes memory reason) {
    // clear the payload first to prevent reentrancy, and then execute
  the message
    bytes32 payloadHash = _clearPayload(_origin, _receiver,
  abi.encodePacked(_guid, _message));
    (success, reason) = _safeCallLzReceive(_origin, _receiver, _guid,
  _message, _extraData);

    if (success) {
      emit PacketReceived(_origin, _receiver);
    } else {
      // if the message fails, revert the clearing of the payload
      _inbound(_origin, _receiver, payloadHash);

      // refund the native fee if the message fails to prevent the
  loss of fund
      if (msg.value > 0) {
        (bool sent, ) = msg.sender.call{value: msg.value}(””);
        require(sent, Errors.INVALID_STATE);
      }
```

```
            // if the message fails, revert the clearing of the payload
            _inbound(_origin, _receiver, payloadHash);

            emit LzReceiveFailed(_origin, _receiver, reason);
        }
    }
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 6ce8d31c.

4.4 Potential reentrancy in lzCompose function

- **Target**: MessagingComposer
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The lzCompose function deletes the composed message hash prior to executing a message to the composer, thereby mitigating the risk of a reentrancy attack. In instances where message delivery fails, the hash is reinstated for resending. Subsequently, an external call is made to refund the native fee to the caller.

```
function lzCompose(
    address _sender,
    address _composer,
    bytes32 _guid,
    bytes calldata _message,
    bytes calldata _extraData
) external payable returns (bool success, bytes memory reason) {
     ...
    composedMessages[_sender][_composer][_guid]
= _RECEIVED_MESSAGE_HASH;
    {
        bytes memory callData = abi.encodeWithSelector(
            ILayerZeroComposer.lzCompose.selector,
            _sender,
            _guid,
            _message,
            msg.sender,
            _extraData
        );
        (success, reason) = _composer.safeCall(gasleft(), msg.value,
callData);
    }
    if (success) {
        emit ComposedMessageReceived(_sender, _composer, _guid,
expectedHash, msg.sender);
    } else {
        // if the message fails, revert the state
        composedMessages[_sender][_composer][_guid] = expectedHash;
```

```
            // refund the native fee if the message fails to prevent the
    loss of fund
            if (msg.value > 0) {
                (bool sent, ) = msg.sender.call{value: msg.value}("");
                require(sent, Errors.INVALID_STATE);
            }
            emit LzComposeFailed(_sender, _composer, _guid, expectedHash,
    msg.sender, reason);
        }
    }
}
```

During this second external call, the caller may reenter and execute the message correctly, as the `composedMessages` has been restored prior to this call.

Impact

Initially, the `ComposedMessageReceived` event will be emitted following successful execution. However, the `LzComposeFailed` event will also be emitted for the same packet within the same transaction, but in an incorrect order.

Recommendations

Restore the hash after the external call:

```
    function lzCompose(
        address _sender,
        address _composer,
        bytes32 _guid,
        bytes calldata _message,
        bytes calldata _extraData
    ) external payable returns (bool success, bytes memory reason) {
        ...
        composedMessages[_sender][_composer][_guid]
    = _RECEIVED_MESSAGE_HASH;
        ...
        if (success) {
            emit ComposedMessageReceived(_sender, _composer, _guid,
    expectedHash, msg.sender);
        } else {
            // if the message fails, revert the state
            composedMessages[_sender][_composer][_guid] = expectedHash;

            // refund the native fee if the message fails to prevent the
    loss of fund
            if (msg.value > 0) {
                (bool sent, ) = msg.sender.call{value: msg.value}("");
                require(sent, Errors.INVALID_STATE);
            }

            // if the message fails, revert the state
            composedMessages[_sender][_composer][_guid] = expectedHash;

            emit LzComposeFailed(_sender, _composer, _guid, expectedHash,
    msg.sender, reason);
        }
    }
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 6ce8d31c.

4.5 Potential replay across chains

- **Target**: VerifierNetwork
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

As LayerZero is a cross-chain application, VerifierNetwork might be deployed across multiple chains. There exists a possibility of message replay if signers are shared between multiple instances of VerifierNetwork. This is because there is no unique identifier pinning the VerifierNetwork the message can be executed at.

Impact

A message can be replayed between instances of VerifierNetwork if the signers/quorum is shared.

As the signed message includes the target address, calls to `onlySelf(orAdmin)` functions cannot be replayed. Furthermore, calls to ULN functions such as verify would not be useful to an attacker as well.

Recommendations

Add an identifier to VerifierNetwork that is checked as part of the signature.

Remediation

LayerZero labs acknowled the issue and has fixed it in commit 175c08bd

4.6 Re-execution of instructions is blocked if a signature verification failed

- **Target**: VerifierNetwork
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The execute function is designed to process a sequence of instructions, executing them in the specified order. If any instruction fail during this process, the function will emit the ExecuteFailed event and proceed with the execution of the next instructions. The usedHashes array is used to prevent reentrancy and replay attacks. If the hash of an instruction is identified as already used during the execution process, the HashAlreadyUsed event is emitted, and the function moves on to the next instruction. In cases where the hash is not previously marked as used, it will be marked and signature verification will be conducted. Upon successful verification, an external call is initiated to execute it. But the execution of an instruction fails, usedHashes is reset, allowing for the possibility of re-execution. However, if signature validation fails, the instruction is still marked as used.

Impact

Consequently, instructions that are marked as used but fail signature validation are blocked from being re-attempted for execution.

Recommendations

```
function execute(ExecuteParam[] calldata _params)
    external onlyRole(ADMIN_ROLE) {
        for (uint i = 0; i < _params.length; ++i) {
            ExecuteParam calldata param = _params[i];
            ...

            // 2. skip if hash used
            bool shouldCheckHash
    = _shouldCheckHash(bytes4(param.callData));
            if (shouldCheckHash) {
                if (usedHashes[hash]) {
                    emit HashAlreadyUsed(param, hash);
                    continue;
                } else {
                    usedHashes[hash] = true; // prevent reentry and replay
    attack
                }
            }

            // 3.  check signatures
            if (verifySignatures(hash, param.signatures)) {
                // execute call data
                (bool success, bytes memory rtnData)
    = param.target.call(param.callData);
                if (!success) {
                    if (shouldCheckHash) {
                        usedHashes[hash] = false;
                        emit ExecuteFailed(i, rtnData);
                    }
                }
            } else {
                if (shouldCheckHash) {
                    usedHashes[hash] = false;
                }
                emit VerifySignaturesFailed(i);
            }
        }
    }
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 3bb3e16d.

4.7 SafeCall does not check that target is contract

- **Target**: SafeCall
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The function `safeCall` is used to call the `_target` contract with a specified gas limit and value and captures the return data. But at the same time, there is no verification that the address is really a contract.

Impact

If the `_target` address is not a contract, the call will be successful although the function call has not actually been made.

Recommendations

We recommend adding a check that ensures the `_target` has a code.

```
function safeCall(
        address _target,
        uint256 _gas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool, bytes memory) {
        uint size;
        assembly {
            size := extcodesize(_target)
        }
        if (size == 0) {
            return (false, bytes(string(”no code!”)));
        }

        // set up for assembly call
        uint256 _toCopy;
        bool _success;
        ...
    }
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 0d04db22.

4.8 Potential reentrancy through `execute` function

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

Description

The `execute` function takes an array of `ExecuteParam`. For each of the parameters, it verifies the `signatures`, executes the `callData`, and stores its hash (if the call is successful) to prevent replay:

```
// 2. skip if hash used
bool shouldCheckHash = _shouldCheckHash(bytes4(param.callData));
if (shouldCheckHash && usedHashes[hash]) {
    emit HashAlreadyUsed(param, hash);
    continue;
}

// 3. check signatures
if (verifySignatures(hash, param.signatures)) {
    // execute call data
    (bool success, bytes memory rtnData)
    = param.target.call(param.callData);
    if (success) {
        if (shouldCheckHash) {
            // store usedHash only on success
            usedHashes[hash] = true; // prevent reentry and replay attack
        }
    } else {
        emit ExecuteFailed(i, rtnData);
    }
}
```

The call can be made more than once for one signature if the `execute` function reenters during the external call since the `hash` is not stored before the external call.

Impact

Though unlikely to be exploited, there is the potential for unexpected behavior because the function does not sufficiently prevent reentrancy attacks.

Recommendations

Store the hash before the external call:

```
// 2. skip if hash used
bool shouldCheckHash = _shouldCheckHash(bytes4(param.callData));
if (shouldCheckHash && usedHashes[hash]) {
    emit HashAlreadyUsed(param, hash);
    continue;
}

// 3. check signatures
if (verifySignatures(hash, param.signatures)) {
    usedHashes[hash] = shouldCheckHash; // prevent reentry and replay

    // execute call data
    (bool success, bytes memory rtnData)
    = param.target.call(param.callData);
    if (success) {
        if (shouldCheckHash) {
            // store usedHash only on success
            usedHashes[hash] = true; // prevent reentry and replay
                attack
        }
    } else {
    if (!success) {
        delete usedHashes[hash];
        emit ExecuteFailed(i, rtnData);
    }
}
```

Remediation

This issue has been acknowledged by LayerZero Labs.

4.9 UlnConfig inconsistencies

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

Description

There are a few inconsistencies in the UlnConfig code:

- For CONFIG_TYPE_VERIFIERS and CONFIG_TYPE_OPTIONAL_VERIFIERS, the config-setting code should only do assertions and assignments if !useCustomVerifiers or !useCustomOptionalVerifiers, respectively; otherwise, there are odd situations where, for example, a specific custom verifiers config cannot be set because UlnConfig thinks the custom optional verifiers will be used in the config, when in reality useCustomOptionalVerifiers is false.

- The _assertNoDuplicates function would be more useful if it would also assert no collisions between verifiers and optional verifiers (i.e., that there is no intersection).

- More importantly, there is also a situation where there could be zero — or greater than the max uint8 — (required and optional) verifiers configured:

  - config.useCustomVerifiers = true
  - config.verifierCount = 0
  - config.optionalVerifierThreshold = 1
  - config.useCustomOptionalVerifiers = false
  - defaultConfig.optionalVerifierCount = 0
  - other specific values required to set the above config

  This is due to the following code:

```
function _assertVerifierList(uint32 _remoteEid, address _oapp)
    internal view {
    UlnConfigStruct memory config = getUlnConfig(_oapp, _remoteEid);
    // it is possible for sender to configure nil verifiers
    require(config.verifierCount
    > 0 || config.optionalVerifierThreshold > 0,
        Errors.VERIFIERS_UNAVAILABLE);
        // verifier options restricts total verifiers to 255
        require(config.verifierCount + config.optionalVerifierCount
        ≤ type(uint8).max, Errors.INVALID_SIZE);
    }
```

Impact

It is possible to set invalid configuration in certain edge cases that may allow a message to pass with no confirmations. This situation is only achievable if both the admin and OApp independently configure specific values as the OApp and default configurations.

Recommendations

Enforce function requirements during the `getUlnConfig` call. Then, call `getUlnConfig()` after changing any configurations.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 3dfec105.

4.10 Signature verification `ecrecover` is missing error condition check

- **Target**: MultiSig, MultiSigUpgradeable
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `ecrecover` call in the following function recovers the signer adress from the signature components:

```
function verifySignatures(bytes32 _hash, bytes calldata _signatures)
    public view returns (bool) {
    if (_signatures.length != uint(quorum) * 65) {
        return false;
    }

    bytes32 messageDigest = _getEthSignedMessageHash(_hash);

    address lastSigner = address(0); // There cannot be a signer with
    address 0.
    for (uint i = 0; i < quorum; i++) {
        (uint8 v, bytes32 r, bytes32 s) = _splitSignature(_signatures, i);
        address currentSigner = ecrecover(messageDigest, v, r, s);

        if (currentSigner <= lastSigner) return false; // prevent
    duplicate signatures
        if (!signers[currentSigner]) return false; // signature is not
    from a signer
        lastSigner = currentSigner;
    }
    return true;
}
```

Per the Solidity documentation, the `ecrecover` built-in function returns zero on error:

> ... recover the address associated with the public key from elliptic curve signature or return zero on error.

Impact

The duplicate signer check ensures zero is not a valid signer address. However, the error condition is not explicitly checked.

Recommendations

We recommend checking the return value to ensure it is nonzero.

```
// [ ... ]

address lastSigner = address(0); // There cannot be a signer with address 0.
for (uint i = 0; i < quorum; i++) {
    (uint8 v, bytes32 r, bytes32 s) = _splitSignature(_signatures, i);
    address currentSigner = ecrecover(messageDigest, v, r, s);
    require(currentSigner != 0);

    if (currentSigner <= lastSigner) return false; // prevent duplicate signatures
    if (!signers[currentSigner]) return false; // signature is not from a signer
    lastSigner = currentSigner;
}

// [ ... ]
```

Remediation

This issue has been acknowledged by LayerZero Labs.

4.11 Unnecessary caller restriction on execute function

- **Target**: VerifierNetwork
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Low

Description

The execute function restricts the caller to those with the admin role only:

```
function execute(ExecuteParam[] calldata _params)
    external onlyRole(ADMIN_ROLE) {
    for (uint i = 0; i < _params.length; ++i) {
        ExecuteParam calldata param = _params[i];
        // 1. skip if expired
        if (param.expiration <= block.timestamp) {
            continue;
        }
        // generate and validate hash
        bytes32 hash = hashCallData(param.target, param.callData,
    param.expiration);
        // 2. skip if hash used
        bool shouldCheckHash = _shouldCheckHash(bytes4(param.callData));
        if (shouldCheckHash && usedHashes[hash]) {
            emit HashAlreadyUsed(param, hash);
            continue;
        }
        // 3. check signatures
        if (verifySignatures(hash, param.signatures)) {
            // execute call data
            (bool success, bytes memory rtnData)
    = param.target.call(param.callData);
            if (success) {
                if (shouldCheckHash) {
                    // store usedHash only on success
                    usedHashes[hash] = true; // prevent reentry and replay
    attack
                }
            } else {
                emit ExecuteFailed(i, rtnData);
            }
        }
    }
}
```

However, this restriction is unnecessary because the function requires a quorum of valid signatures.

Impact

If an admin were to fail to call the `execute` function for any reason, the ULN would not deliver any messages to the endpoint, even if all of the signers were online.

Recommendations

The function should be able to be called permissionlessly to ensure the signatures may always be submitted.

Remediation

This issue has been acknowledged by LayerZero Labs.

4.12 Admin can block relayer in VerifierNetwork

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

Description

The VerifierNetwork contract takes a quorum of signers (i.e., multi-sig) to verify a message with the ULN. The submission process uses a single call to the execute function, which can only be done by the admin:

```
/// @dev takes a list of instructions and executes them in order
/// @dev if any of the instructions fail, it will emit an error event and
    continue to execute the rest of the instructions
/// @param _params array of ExecuteParam, includes target, callData,
    expiration, signatures
function execute(ExecuteParam[] calldata _params)
    external onlyRole(ADMIN_ROLE) {
```

However, even if the signers are online, if the admin does not submit signatures for any reason, the messages will no longer be verified.

Impact

The admin can block message verification and require manual intervention by the OApp.

Recommendations

Make the execute call permissionless.

Remediation

LayerZero Labs acknowledged this finding, noting that they intend to add a method for the quorum to change the admin should the admin ever go offline.

5 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.

5.1 Cautions for OApp developers

It is important that OApp developers consider the following:

- **Only override `_lzReceive` — not `lzReceive`.** Note that overriding `_lzReceive` (i.e. not `lzReceive`) would not impact functionality (i.e., it would still pass tests) and would produce no warning but would skip the following critical security checks:
  - `require(address(endpoint) == msg.sender, “OApp: endpoint only”);`
  - `assertRemoteAddress(_srcEid, _srcAddress);`
  - `_acceptNonce(_srcEid, _srcAddress, _nonce);`
- **Messages are not ordered by default.** Delivery of messages is ordered, but execution is not.

5.2 The transfer function does not work on zkSync

Gas calculations on zkSync use a dynamic and divergent gas measurement method. Using transfer() on zkSync will exceed the 2300 gas limit, causing the transaction to revert automatically. The `recoverToken` function of EndpointV2 contract uses `transfer` to recover native tokens balance from the contract. But in zkSync this call will be reverted. We recommend using `call` function instead.

```
function recoverToken(address _token, address _to, uint256 _amount)
    external onlyOwner {
    if (_token == address(0)) {
        payable(_to).transfer(_amount);
    } else {
        IERC20(_token).safeTransfer(_to, _amount);
    }
}
```

6 Threat Model

Bug classes

These are the impacts of potential bug classes:

- A vulnerability in in the LayerZero protocol that allows an entity to deliver arbitrary messages from an arbitrary origin may cause total loss of funds on the receiving OApp, despite the OApp logic being correct and safe, in the worst case.
- If one message can be executed more than once, the scope of the impact is changed, but the receiving OApp may suffer a major financial loss in the worst case.
- A bug allowing messages to be sent without paying fees (i.e. bypassing fees) may cause a major financial loss to the executor in the worst case scenario if the executor pays `msg.value` without properly being compensated.
- A denial-of-service bug may cause funds to be frozen in-flight in the worst case.

6.1 Component: Sending from Source Chain

A reversion in this component results in the the message not being sent.

OApp sends the message

To send the message, the OApp calls `_lzSend` which makes an external call to `EndpointV2.send`.

- `EndpointV2.send` will send the message with the source set to msg.sender so that the destination OApp can authenticate the sender.

This makes an internal call to `_send`.

Getting the outbound nonce

First, the `_send` function fetches the outbound nonce via `MessagingChannel._outbound` which first increments the nonce, then returns the value.

Getting the send library

Then, it fetches the send library given the OApp and destination endpoint ID using the MessageLibManager.getSendLibrary function.

If the value configured by the OApp is _DEFAULT_LIB, it uses the default-configured send library.

- If the default send library is _UNSET_LIB then the transaction reverts. That means that LayerZero can block messages temporarily if the OApp leaves the send library unconfigured, but the OApp can change the send library at any time.

Calling to messaging library

The messaging library returns a MessagingReceipt including the fee amount that should be charged later.

- The messaging library could return an arbitrary value for the MessagingReceipt, but the library must be allowed by the LayerZero, so the user cannot bypass fees unless LayerZero approves a library which allows this.

Fees are paid

Finally, back in the EndpointV2.send call, the native or LayerZero token fees are paid.

- If fees are insufficiently paid, the whole transaction reverts, and the message is not sent (at the cost of the caller).

6.2 Component: Receiving on Destination Chain

Messaging library marks the message as delivered

The messaging library calls `EndpointV2.deliver` which calls the internal `inbound`.

- The msg.sender is authenticated to make sure it is the receiver's configured receive library. The library logic for processing the packet
- If left unconfigured (i.e. set as the default library), LayerZero could add a new malicious messaging library and change the default receive library. However, protocols can mitigate this centralization risk simply by configuring a specific receive library.
  - If the messaging library does not have a default configuration for the remote endpoint ID, the library cannot be used (i.e. it will revert). However, once a default configuration has been set,
- The message's nonce must be greater than the lazy inbound nonce; the nonce is set to the message with the highest nonce that has been executed which ensures messages cannot be delivered (i.e. the payload hash cannot be stored) for messages that have already been delivered and possibly executed.

  Note that delivery can happen out of order, but packet-level censorship would only be possible by not delivering a message (to ensure it cannot be executed)—which would block future operation on the path.
- A receiving message library could censor a packet by delivering the incorrect payload hash. However, a message library can only be chosen if LayerZero trusts it—and, if the OApp configured a custom receive library, if the OApp trusts it too.
- Replay has no effect since delivery simply stores the payload hash.

Endpoint executes the message

The executor calls `EndpointV2.lzReceive` which clears the stored payload and calls the OApp's `lzReceive` function.

- Execution can happen permissionlessly to ensure delivered messages can be fairly executed.
- If it reverts, it re-stores the payload hash and refunds the msg.value to the caller.
- Messages may be executed out of order, but they can only be executed if all

messages have been delivered up until the nonce of the desired message. This ensures packet-level censorship cannot happen without blocking all future delivery of messages.

- Replay cannot happen at the execution level because the payload hash will be cleared from storage if execution does not revert. The `_clearPayload` function reverts if there is no stored payload hash

OApp receives the message

The endpoint calls the `lzReceive` function which then passes execution to the internal `_lzReceive` function on the OApp which receives the message.

- The msg.sender is authenticated to ensure it is sent from the endpoint.
- The source sender of the message is checked to ensure it is from a trusted peer
- The nonce is checked to ensure it is receiving the message when expected. The OApp may override `_acceptNonce` to enforce a specific nonce order, etc.

7 Assessment Results

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

During our assessment on the scoped Endpoint V2 contracts, we discovered 12 findings. One critical issue was found. Two were of medium impact, seven were of low impact, and the remaining findings were informational in nature. LayerZero Labs acknowledged all findings and implemented fixes.

7.1 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.