LayerZero V2 Security Audit

Prepared for LayerZero

Created by Blockian

Contents

1. Introduction
2. Disclaimer
3. About Blockian
4. About LayerZero
5. Severity Classification
6. Security Assessment Summary
7. Findings Summary
8. Detailed Findings
    1. M-01 - Double Fee Payments for Overlapping DVNs
    2. M-02 - Discrepancy Between PacketSent Events in ULN301 and EndpointV2
    3. M-03 - Risks of Breaking the Messaging Pipe During Migration to ULN301
    4. L-01 - `_checkAndEmptyVerified` Can Be Called Twice for the Same Input
    5. L-02 - `addressSizes[_dstEid]` is Allowed to be Zero When Sending
    6. I-01 - Variable Shadowing Issue with `sendLibrary` in `EndpointV2.sol`
    7. I-02 - Safer Payment Order in EndpointV2's `Send` Function
    8. I-03 - OApp Can Potentially Avoid Treasury Fees by Reverting getFees
    9. I-04 - `validatePacket` Function Accessibility Issue In `SimpleMessageLib`
    10. I-05 - `ExecutionState` Missing Possible states

Introduction

A security review of the upcoming Endpoint V2 and ULN V3 of the **LayerZero** protocol was done by **Blockian** (**pwnmansh1p** and **ControlZ**).

Disclaimer

It's important to note that a review of smart contract security can't assure that all potential issues have been identified. Given the constraints of time and resources, the goal is to uncover as many vulnerabilities as feasible. However, there's no guarantee that the security review will either find all vulnerabilities or ensure total security afterward. Follow-up security evaluations, as well as establishing bug bounty initiatives and continuous on-chain surveillance, are highly advised.

Blockian

Blockian is a team of independent smart contract security researchers. The team includes @pwnmansh1p, who brings over a decade of security research experience to the table, and @ControlZ, a former software architect now specializing in security research. They've uncovered numerous vulnerabilities across various protocols and are committed to improving the blockchain landscape. Reach out on Twitter @blockian

Read more Here

Protocol

LayerZero is an open-source, immutable messaging protocol designed to facilitate the creation of omnichain, interoperable applications.

Read more at the Docs

Centralization

Overview

This section evaluates the level of centralization present in the **LayerZero V2** smart contract system. A higher score indicates a more decentralized architecture, which is generally preferred for reducing risks associated with single points of failure and promoting trustless interaction.

Evaluation Criteria

1. Smart Contract Ownership: Assess whether the smart contracts are owned by a single address or a decentralized governance system.
2. Upgradeability: Determine if the contracts can be upgraded and, if so, who controls these upgrades.
3. Oracles and External Dependencies: Identify reliance on external data sources and their control mechanisms.
4. Funds Custody: Evaluate how user funds are managed and the extent of control exerted by the contract over these funds.

Smart Contract Ownership

While ownership of the contracts is handled by a single entity (With the VerifierNetwork being the exception) the scope of control the owner possesses is fairly limited. For example, setting trusted libs, setting the lz token, etc. **(Score: 7/10).**

Upgradeability

The contracts are not upgradeable (with the Executor being an exception. However, users can opt to use any Executor they prefer). Thus, there can't be unexpected upgrades, and what you see is what you get. **(Score: 10/10).**

Oracles and External Dependencies

The system relies on a single centralized oracle for price feeds for the default Verifier and Executor fees, introducing a single potential point of failure. However, as mentioned before, any user can opt to use an infrastructure trusted by them **(Score: 7/10)**.

Funds Custody

The system doesn't hold user funds, it only collects fees paid for workers **(Score: 10/10)**.

Score

The overall Centralization Score is calculated as the average of the individual scores:
(7 + 10 + 7 + 10) / 4 = 8.5
Final Score: **8.5/10**

Severity

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Summary

Security Assessment Summary

This security audit is focused on **LayerZero V2**.

***review commit hash* - 06241846f471704c644b5ec1a233e9aef2a64665**

***fixes review commit hash* - TBD**

Scope

The following smart contracts were in scope of the audit:
`layerzero-v2/evm/protocol/contracts/**`
`layerzero-v2/evm/messagelib/contracts/**`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 3 issues
- Low: 2 issues
- Information: 4 notes

Findings

ID	Title	Severity
M-01	Double Fee Payments for Overlapping DVNs	Medium
M-02	Discrepancy Between PacketSent Events in ULN301 and EndpointV2	Medium
M-03	Risks of Breaking the Messaging Pipe During Migration to ULN301	Medium
L-01	`_checkAndEmptyVerified` Can Be Called Twice for the Same Input	Low
L-02	`addressSizes[_dstEid]` is Allowed to be Zero When Sending	Low
I-01	Variable Shadowing Issue with `sendLibrary` in `EndpointV2.sol`	Info
I-02	Safer Payment Order in EndpointV2's `send` Function	Info
I-03	OApp Can Potentially Avoid Treasury Fees by Reverting `getFees`	Info
I-04	`validatePacket` Function Accessibility Issue In `SimpleMessageLib`	Info
I-05	`ExecutionState` Missing Possible states	Info

Detailed

[M-01]

Double Fee Payments for Overlapping DVNs

During the execution of the `_assignJobs` function in the process of DVN fee payments:

```
function _assignJobs(
    mapping(address => uint256) storage _fees,
    UlnConfig memory _ulnConfig,
    ILayerZeroDVN.AssignJobParam memory _param,
    bytes memory dvnOptions
) internal returns (uint256 totalFee, uint256[] memory dvnFees) {
    (bytes[] memory optionsArray, uint8[] memory dvnIds) =
     DVNOptions.groupDVNOptionsByIdx(dvnOptions);

    uint8 dvnsLength = _ulnConfig.requiredDVNCount + _ulnConfig.optionalDVNCount;
    dvnFees = new uint256[](dvnsLength);
    for (uint8 i = 0; i < dvnsLength; ++i) {
        address dvn = i < _ulnConfig.requiredDVNCount
            ? _ulnConfig.requiredDVNs[i]
            : _ulnConfig.optionalDVNs[i - _ulnConfig.requiredDVNCount];

        bytes memory options = "";
        for (uint256 j = 0; j < dvnIds.length; ++j) {
            if (dvnIds[j] == i) {
                options = optionsArray[j];
                break;
            }
        }

        dvnFees[i] = ILayerZeroDVN(dvn).assignJob(_param, options);
        if (dvnFees[i] > 0) {
            _fees[dvn] += dvnFees[i];
            totalFee += dvnFees[i];
        }
    }
}
```

Due to the allowance of DVN overlaps in both required and optional lists, the current implementation results in certain DVNs being paid a fee twice and assigned a job twice. This occurs when the same DVN appears in both lists, leading to unnecessary additional fees for a service required only once.

Suggestion

Considering a DVN listed as required should not simultaneously be optional, we propose prohibiting DVN overlaps. This change seems logical as there is no practical need for a DVN to be in both categories.

Should overlaps be essential for protocol operations, ensure `ILayerZeroDVN(dvn).assignJob(_param, options)` is invoked only once for each unique DVN.

[M-02]

Discrepancy Between PacketSent Events in ULN301 and EndpointV2

There is an inconsistency in the definition of the `PacketSent` event between `SendLibBaseE1.sol` and `ILayerZeroEndpointV2.sol` (the interface implemented by `EndpointV2`).

In SendLibBaseE1.sol, the PacketSent event is defined as follows:

```
abstract contract SendLibBaseE1 is SendLibBase, AddressSizeConfig, IMessageLibE1 {
 // ...
 // this event should be identical to the one on Endpoint V2
 event PacketSent(bytes encodedPayload, bytes options, uint256 nativeFee, uint256
 lzTokenFee);
```

Whereas in `ILayerZeroEndpointV2.sol`, the event is defined differently:

```
interface ILayerZeroEndpointV2 is IMessageLibManager, IMessagingComposer,
IMessagingChannel, IMessagingContext {
   event PacketSent(bytes encodedPayload, bytes options, address sendLibrary);
```

It's evident that these event signatures are not identical, even though they are intended to be the same for interoperability purposes.

Impact

This inconsistency poses a risk as off-chain components depend on these events. If some off-chain components are designed to expect only a single format of the `PacketSent` event, it could lead to operational issues.

Discussion

LayerZero: `EndpointV2` handles the fee payment to the `sendLibrary`. so the event signatures are different.

[M-03]

Risks of Breaking the Messaging Pipe During Migration to ULN301

For an OApp to transition to ULN301, it must undergo a minimum of two key steps:

1. On the source chain, change the sending library to `uln301`.
2. On the destination chain, switch the receiving library to `uln301`.

These actions are unlikely to occur simultaneously (differences in block mining speeds, etc). Consequently, any messages sent during this transitional phase could potentially disrupt the messaging pipe between the source and destination chains.

Detailed explanation

The process of sending a message involves incrementing the `outboundNonce` in the `NonceContract`. And, when receiving a message, the `inboundNonce` in the `Endpoint` is incremented and is verified with the nonce received in the packet.

If there is a nonce mismatch, this particular messaging pipe is considered broken.

Consider a scenario during the transitional phase where one chain employs uln2 and the other uln301. The message sent will increase the `outboundNonce` as expected. However, due to the incompatibility of uln2 and uln301, the message fails to be delivered, preventing the increment of `inboundNonce` and thus breaking the pipe.

This issue can arise either naturally in a highly active pipe or be induced maliciously by an attacker executing a strategically timed transaction.

Impact

The impact of breaking a pipe is Critical when the pipe breaks permanently. However, in most instances, Verifiers and Executors can intervene and fix this pipe manually. This reduces the severity of the problem from 'Critical' to 'Medium'.

Suggestion

While unavoidable, It is advisable for OApps to initiate their migration by first updating the send library to 301, followed by the receive library to 301. This sequence minimizes risks: if a message is sent during the transition, the Verifier will already update the correct `hashLookup` in the corresponding uln 301 receive library. Subsequently, only the execution of the old package needs to be manually handled, not its verification.

Discussion

`LayerZero:` Yes we will have a migration guide.

[L-01]

`_checkAndEmptyVerified` Can Be Called Twice for the Same Input

The function `_checkAndEmptyVerified` is called within both receive libraries. It operates under the assumption that all entries in `hashLookup` for a given header and payload are deleted, preventing the function from being executed twice for the same payload. However, the code reveals a different scenario:

```
function _checkAndEmptyVerified(UlnConfig memory _config, bytes32 _headerHash,
    bytes32 _payloadHash) internal {
    // ...

    // then it must require optional validations
    uint8 threshold = _config.optionalVerifiersThreshold;
    for (uint8 i = 0; i < _config.optionalVerifiersCount; ++i) {
        if (_verified(_config.optionalVerifiers[i], _headerHash, _payloadHash,
            _config.confirmations)) {
            // increment the optional count if the optional verifier has signed
            threshold--;
            if (threshold == 0) {
                // early return if the optional threshold has hit
                return;
            }
        }
    }

    // revert by default as a catch-all
    revert StillSigning();
}

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

Once a sufficient number of Optional Verifiers have `_verified` and approved the payload, meeting the threshold, the function exits without deleting the `hashLookup` entries for the remaining verifiers.

Consider this scenario:
There are two Optional Verifiers, and the Threshold is one.

On the first invocation of `_checkAndEmptyVerified`, after the first Optional Verifier's entry in `hashLookup` is validated and deleted, the function exits as the threshold is met.

Upon a second invocation, the first Optional Verifier fails the `_verified` check, but the loop continues to the second Optional Verifier.
Since their entry was not removed in the previous call, it passes the `_verified` check, allowing the function to exit again upon meeting the threshold.

Suggestion

There are two options:

1. Continue to delete the `hashLookup` entries for all remaining Optional Verifiers.
2. As each payload is used only once due to the nonce, consider not deleting `hashLookup[_headerHash][_payloadHash][_verifier]` at all, as this action consumes gas for storage operations and is not necessary.

Discussion

`Blockian:` Fixed.

[L-02]

`addressSizes[_dstEid]` is Allowed to be Zero When Sending

In the `SendLibBaseE1.sol` contract, specifically within the `_assertPath` function, there's no check to ensure that `addressSizes[_dstEid]` is not zero. This allows the sending of a `Packet` when the `_path` variable contains only the source address.

Example attack flow

Let's examine the process of creating packet data:

```
function _outbound(
    address _sender,
    uint16 _dstEid,
    bytes calldata _path,
    bytes calldata _message
) internal returns (Packet memory packet) {
    // assert toAddress size
    uint256 remoteAddressSize = addressSizes[_dstEid];
    _assertPath(_sender, _path, remoteAddressSize);

    // increment nonce
    uint64 nonce = nonceContract.increment(_dstEid, _sender, _path);

    bytes32 receiver = AddressCast.toBytes32(_path[0:remoteAddressSize]);

    bytes32 guid = GUID.generate(nonce, localEid, _sender, _dstEid, receiver);

    // assemble packet
    packet = Packet(nonce, localEid, _sender, _dstEid, receiver, guid, _message);
}
```

An attacker sandwiches the transaction setting `addressSizes[_dstEid]` and dispatches two messages:

First Transaction Before Setting `addressSizes[_dstEid]`

In the initial transaction, `_path` is set to only `srcAddress`.
Here, the `nonce` for this path will be 0, and `receiver` will be `bytes32(0)` since `AddressCast.toBytes32` returns `bytes32(0)` for an empty input. GUID is computed with `nonce = 0, localEid, _sender, _dstEid, receiver = bytes32(0)`.

Second Transaction After Setting `addressSizes[_dstEid]`

For the subsequent transaction, `_path` is altered to `address(0) + srcAddress`, making the receiver extracted identical to that in the first transaction.
GUID is computed again with
`nonce = 0, localEid, _sender, _dstEid, receiver = bytes32(0)`,
identical to the previous, allowing the attacker to generate two distinct packets with the same GUID.

Potential Impact

Components dependent on the uniqueness of every GUID might face operational issues.

Suggestion

It is recommended to introduce a check in the `_assertPath` function to confirm that `remoteAddressSize` is not zero.

Discussion

`LayerZero:` Accepted.

`Blockian:` Fixed.

[I-01]

Variable Shadowing Issue with `sendLibrary` in `EndpointV2.sol`

In the `MessageLibManager.sol` contract, a storage variable named `sendLibrary` is declared. Since `EndpointV2` extends `MessageLibManager`, any local variable within `EndpointV2` that is also named `sendLibrary` inadvertently shadows the storage variable declared in `MessageLibManager`.

This shadowing effect can lead to confusion and potential errors in the code, as references to `sendLibrary` in `EndpointV2` may ambiguously point to either the local variable or the inherited storage variable, depending on the context. This can lead to bugs if not handled properly.

Suggestion

The easiest way to avoid such issues is to ensure unique naming for all variables in derived contracts.

Discussion

`LayerZero:` Accepted.

`Blockian:` Fixed partially, the `send` and `_send` functions in EndpointV2 still shadow the `sendLibrary` variable.

[I-02]

Safer Payment Order in EndpointV2's `send` Function

The send function in the `EndpointV2` contract currently handles native payments first, followed by token transactions. The code snippet is as follows:

```
function send(
    MessagingParams calldata _params,
    address _refundAddress
) public payable sendContext(_params.dstEid, msg.sender) returns (MessagingReceipt
  memory) {
    // ...
    // handle native fees
    _payNative(receipt.fee.nativeFee, suppliedNative, sendLibrary, _refundAddress);

    // handle lz token fees
    _payToken(lzToken, receipt.fee.lzTokenFee, suppliedLzToken, sendLibrary,
    _refundAddress);
    // ...
}
```

In this setup, when `_payNative` is called, control is transferred to `_refundAddress` through a `call` operation. While the function is non-reentrant, the current state at this point does not fully reflect the completed transaction and may allow for a type of read-only reentrancy attack.

Suggestion

To be on the safe side, it is recommended to reverse the order of the payment handling by calling `_payToken` before `_payNative`:

```
function send(
    MessagingParams calldata _params,
    address _refundAddress
) public payable sendContext(_params.dstEid, msg.sender) returns (MessagingReceipt
  memory) {
    // ...
    _payToken(lzToken, receipt.fee.lzTokenFee, suppliedLzToken, sendLibrary,
    _refundAddress);

    _payNative(receipt.fee.nativeFee, suppliedNative, sendLibrary, _refundAddress);
    // …
```

Discussion

LayerZero: Accepted.

Blockian: Fixed.

[I-03]

OApp Can Potentially Avoid Treasury Fees by Reverting `getFees`

In `SendLibBase.sol` contract, `getFees` is called within a try statement in the `_quoteTreasury` function:

```
function _quoteTreasury(
    address _sender,
    uint32 _eid,
    uint256 _workerFee,
    bool _payInLzToken
) internal view returns (uint256 nativeFee, uint256 lzTokenFee) {
    // ...
    try
        ILayerZeroTreasury(treasury).getFee{ gas: treasuryGasLimit }(_sender,_eid,
        _workerFee, _payInLzToken)
    // ...
    catch {
        // failure, charges nothing
    }
}
```

Which means that if `getFee` reverts, no fee is charged (as the comment states). A problem can arise if getFee uses some OApp-controlled parameter (for example `_workerFee`, as the OApp can use custom workers) in a way that can revert (for example performing arithmetic operations).

In the current treasury implementation in `Treasury.sol`:

```
function getFee(
    address /*_sender*/,
    uint32 /*_eid*/,
    uint256 _totalFee,
    bool _payInLzToken
) external view override returns (uint256) {
    // ...
    if (_payInLzToken) {
        if (!lzTokenEnabled) revert LzTokenNotEnabled();
        return lzTokenFee;
    } else {
        return (_totalFee * nativeBP) / 10000;
    }
}
```

As can be seen, this function reverts if `_payInLzToken` is passed and `lzTokenEnabled` is false. This is only an issue if it should be allowed for the treasury to not enable paying fees with LzToken when such a token exists. In addition, it can revert if `_totalFee * nativeBP` is overflown.

Discussion

LayerZero: Noted.

[I-04]

`validatePacket` Function Accessibility Issue In `SimpleMessageLib`

ASSUMPTION: It is presumed that SimpleMessageLib is not intended for deployment and active use, hence this is categorized as an informational note rather than a significant issue.

However, if SimpleMessageLib is intended for use as a functional library, this issue would immediately escalate to a critical level.

In the `validatePacket` function:

```
function validatePacket(bytes calldata packetBytes) external {
    require(
        whitelistCaller == address(0x0) || msg.sender == whitelistCaller,
        "SimpleMessageLib: only whitelist caller"
    );
    Origin memory origin = Origin(packetBytes.srcEid(), packetBytes.sender(),
       packetBytes.nonce());
    ILayerZeroEndpointV2(endpoint).deliver(origin, packetBytes.receiverB20(),
       keccak256(packetBytes.payload()));
}
```

The issue arises when `whitelistCaller` is unset, allowing anyone to call `validatePacket` and transmit any data they choose.

Example attack flow

An attacker could exploit the `validatePacket` function in any OApp utilizing the Simple Message Lib as their receive library. If the OApp fails to confirm that the Executor is trustworthy in their `_lzReceive` implementation, a malicious user could potentially execute any desired logic.

Suggestion

To mitigate this risk, either remove the condition `whitelistCaller == address(0x0)` from the require statement or ensure that SimpleMessageLib is not deployed as a legitimate messaging library.

Discussion

LayerZero: Noted. Yes not for production.

[I-05]

`ExecutionState` Missing Possible states

The current definition of `ExecutionState` in the system is limited to three states:

```
enum ExecutionState {
    NotExecutable,
    Executable,
    Executed
}
```

However, the introduction of new functionalities like `burn` in the system architecture may require additional states. Presently, a message that undergoes the `burn` process is categorized under `Executed`, which is a misleading classification.

Potential Impact

Any components that depend on accurate execution state information can be affected. For instance, `LzExecutor`, utilizes the `endpoint.executable` call.

Since it seems the system functions correctly with these states, this issue is classified as informational.