# LayerZero v2

# Audit



Presented by:

**OtterSec** contact@osec.io

**Woosun Song** procfs@osec.io

**James Wang** james.wang@osec.io

**Robert Chen** r@osec.io

# Contents



## Appendices





# 01 | Executive Summary

## Overview

LayerZero Labs engaged OtterSec to perform an assessment of the `layerzero-v2` program. This assessment was conducted between July 14th and August 3rd, 2023. For more information on our auditing methodology, see Appendix B.

After the initial engagement, we performed additional reviews of the `bridge.move` modifications in #233.

## Key Findings

Over the course of this audit engagement, we produced 10 findings in total.

In particular, we discovered a vulnerability that allows attackers to steal funds from a user who has approved the fee handler (OS-LZ-ADV-00). Apart from this, we identified issues related to signature replay, including missing cross-chain signature replay checks (OS-LZ-ADV-02) and incorrect per-chain replay protection (OS-LZ-ADV-02), which could be abused for denial of service. We also discussed the security implications of compromised privileged roles and solutions to mitigate them (OS-LZ-ADV-03).

We have also provided recommendations regarding an out-of-bounds memory read (OS-LZ-SUG-00) and incorrect data processing logic (OS-LZ-SUG-01). While these issues did not pose any security implications and were solely triggerable by faulty user input, addressing them is recommended to enhance the protocol's robustness. Additionally, we advised modifying `getFeeOnSend` to be payable to enable its callers to cover price feed fees (OS-LZ-SUG-02).

# 02 | Scope

The source code was delivered to us in a git repository at github.com/LayerZero-Labs/monorepo/tree/main/packages/layerzero-v2. This audit was performed against commit 41085b1. We performed additional reviews up to c71f996.

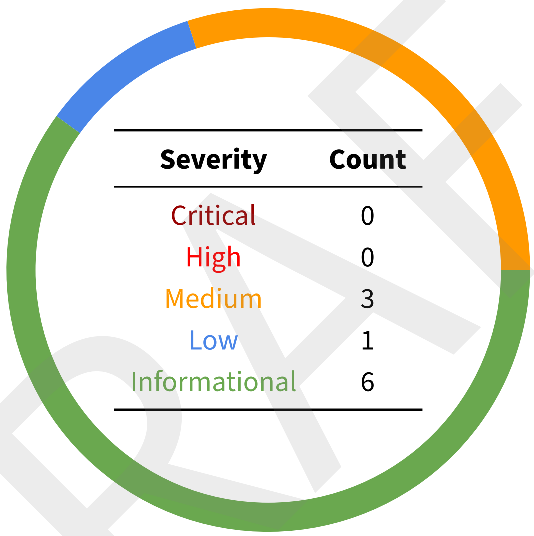
A brief description of the programs is as follows.

| Name | Description |
|---|---|
| endpoint | `Endpoint` facilitates end-to-end interactions with an `OApp`. It utilizes contracts named `MessageLibs` for sending and receiving operations. |
| ultralightnode | `UltraLightNode` is the most commonly utilized `MessageLib`, with two implementations, 301 and 302, compatible with endpoint versions 1 and 2, respectively. To validate and deliver messages, UltraLightNode interacts with two entities, `VerifierNetwork` and `Executor`. |
| treasury | It is a smart contract that collects `lzToken` fees. |
| verifier network | It is responsible for verifying messages. It verifies messages via multi-signature approval; once a quorum of approval is reached, it passes approval to `UltraLightNode` to be ready for delivery. |
| executor | It is responsible for delivering messages to an endpoint after they have been verified by the `VerifierNetwork`. |

# 03 | Findings

Overall, we reported 10 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will help mitigate future vulnerabilities.



| Severity | Count |
| --- | --- |
| Critical | 0 |
| High | 0 |
| Medium | 3 |
| Low | 1 |
| Informational | 6 |

# 04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

| ID | Severity | Status | Description |
|---|---|---|---|
| OS-LZ-ADV-00 | Medium | Resolved | `payFee` allows the invocation of `ERC20.transferFrom` with arbitrary arguments, potentially resulting in the theft of funds for those who have approved `TreasuryFeeHandler`. |
| OS-LZ-ADV-01 | Medium | Resolved | `VerifierNetwork` `multisig` signatures used in `execute` and `quorumChangeAdmin` may be replayed. |
| OS-LZ-ADV-02 | Medium | Resolved | `execute` of `VerifierNetwork` implements replay protection in an incorrect manner. |
| OS-LZ-ADV-03 | Low | Resolved | We have analyzed the implications of compromised roles and provided suggestions to mitigate their risks. |

# OS-LZ-ADV-00 [med] | Theft Of Funds

## Description

`payFee` allows the invocation of `ERC20.transferFrom` with arbitrary arguments, which may be exploited to steal tokens from those who have approved the fee handler. A check exists on the payment and sender addresses, but the check may be bypassed by specifying the same address. By specifying `_lzTokenPaymentAddress` and `_sender` to the victim address and `_treasury` as an attacker-owned address, an attacker may steal funds equal to the approved amount.

*packages/layerzero-v2/evm/messagelib/uln301/TreasuryFeeHandler.sol* SOLIDITY

```solidity
function payFee(
    address _lzToken,
    address _sender,
    address _lzTokenPaymentAddress,
    uint _required,
    uint _supplied,
    address _treasury
) external {
    // lz token payment address must equal the sender or the tx.origin otherwise
    ↪  the transaction reverts
    require(_lzTokenPaymentAddress == _sender || _lzTokenPaymentAddress ==
    ↪  tx.origin, Errors.INVALID_ARGUMENT);
    require(_required <= _supplied, Errors.INVALID_AMOUNT);

    // send lz token fee to the treasury directly
    IERC20(_lzToken).safeTransferFrom(_lzTokenPaymentAddress, _treasury,
    ↪  _required);
}
```

## Remediation

A validation check must be implemented to ensure that `msg.sender` is of type `MessageLibV1`.

## Patch

Resolved in 5de9b05 and 079633a.

# OS-LZ-ADV-01 [med] | VerifierNetwork Signature Replay

## Description

The original implementation of the `VerifierNetwork` `multisig` signed message does not include unique `VerifierNetwork` IDs. This omission allows for the potential replay of signatures across different `VerifierNetworks`. Furthermore, there is no assurance that the assigned admin account is necessarily an Externally Owned Account (EOA) due to the introduction of `quorumChangeAdmin`. Consequently, there exists a possibility of designating a contract that only exists on a specific chain as the `admin`, and a replay of signatures in this context may result in an unexpected admin assignment on a different chain.

## Remediation

Add a unique ID to the message hash for verification.

*packages/layerzero-v2/evm/messagelib/contracts/uln/VerifierNetwork.sol* DIFF

```diff
  constructor(
+     uint32 _vid,
      address[] memory _messageLibs,
      address _priceFeed,
      address[] memory _signers,
      uint64 _quorum,
      address[] memory _admins
  ) Worker(_messageLibs, _priceFeed, 12000, address(0x0), _admins)
    ↪  MultiSig(_signers, _quorum) {
+     vid = _vid;
  }

  function quorumChangeAdmin(ExecuteParam calldata _param) external {
      require(_param.expiration > block.timestamp, "Verifier: expired");
      require(_param.target == address(this), "Verifier: invalid target");
+     require(_param.vid == vid, "Verifier: invalid vid");
      ...
  }

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

### Patch

Resolved in 175c08b and d17f2a0.

# OS-LZ-ADV-02 [med] | Incorrect Multisig Replay Protection

## Description

`execute` in `VerifierNetwork` implements replay protection incorrectly, permanently blocking the execution of transactions, even if they result in failure. The `usedHashes` mapping is utilized to prevent replay attacks. However, the mapping is populated even if the signature validation (call to `verifySignature`) or low-level call fails.

*packages/layerzero-v2/evm/messagelib/contracts/uln/VerifierNetwork.sol* SOLIDITY

```solidity
function execute(ExecuteParam[] calldata _params) external onlyRole(ADMIN_ROLE) {
   for (uint i = 0; i < _params.length; ++i) {
       /* ... */
       // 2. skip if hash used
       if (_shouldCheckHash(bytes4(param.callData))) {
           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) {
               emit ExecuteFailed(i, rtnData);
           }
       }
   }
   }
```

This allows a compromised `ADMIN_ROLE` to populate hashes even if the execution does not occur by providing an invalid signature. This may result in a single compromised `ADMIN_ROLE` to cause a denial of service, compromising the system's robustness.

## Remediation

Perform the population of the `usedHashes` map after execution completion.

## Patch

Resolved in 3bb3e16 and 93211fc.

## OS-LZ-ADV-03 [low] | Potential Centralization Risks

### Description

We enumerated the potential consequences for each role in the following components: `VerifierNetwork`, `EndpointV2`, and `PriceFeed`:

| # | Component | Role | Consequences |
|---|---|---|---|
| 1 | `VerifierNetwork` | `DEFAULT_ADMIN_ROLE` | An attacker may add any `MessageLib` to the deny list, they may exploit to incur denial-of-service. |
| 2 | `VerifierNetwork` | `ADMIN_ROLE` | An attacker may call `execute` maliciously by reordering the transactions or partially dropping them. |
| 3 | `EndpointV2` | `owner` | An attacker may register a malicious `messagelib` and manipulate the behavior of associated `OApps`. |
| 4 | `PriceFeed` | `PriceUpdater` | An attacker may censor messages by setting abnormally high cross-chain transfer fees. |

We provide further analysis of each centralized role here:

| # | Analysis |
|---|---|
| 1 | This `DEFAULT_ADMIN_ROLE` is properly set to `address(0x0)` in the current contract. We only included it here for completeness. |
| 2 | The ability for `ADMIN_ROLE` to drop messages introduces a denial of service risk, which should be mitigated. Contrarily, message reordering attacks are less of an immediate threat since none of the currently implemented functionalities are affected. However, we still advise enforcing message ordering for additional resilience to prevent future upgrades/expansions from breaking this invariant and resulting in exploitable issues. |
| 3 | Only `OApps` utilizing default configurations will be affected by malicious `EndpointV2` `owners`. Nonetheless, we suggest developers provide additional protection for these `OApps` due to their potential severity. |
| 4 | Since `PriceFeed` assignment for `VerifierNetwork` cannot be updated, a malicious `PriceUpdater` may bring down an entire `VerifierNetwork`. While `OApps` may employ custom `VerifierNetwork` to avoid such risks, we suspect that the complexity of running a custom `VerifierNetwork` + off-chain message relayer will deter most users from doing so. Thus, it is preferable to introduce mechanisms that raise the bars of `priceFeed` based denial of service attacks to improve the UX and robustness of the protocol. |

## Remediation

In cases 1, 2, and 4, the centralization risk represents a trade-off between security and design complexity rather than being a risk that may be entirely prevented. In contrast, case 3 may be prevented by implementing a timelock mechanism for `messageLib` management, similar to how it is managed in `EndpointV1`.

## Patch

The Layerzero team minimized the chances of `VerifierNetwork` `ADMIN_ROLE` maliciously dropping messages in 361d741 by allowing `multisig` to bypass `execute` and directly grant `ADMIN_ROLE` and removing code related to setting `DEFAULT_ADMIN` in cb3a118.



# 05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and could lead to security issues in the future.

| ID | Description |
|---|---|
| OS-LZ-SUG-00 | `_splitSignature` lacks bounds checks when performing memory reads via inline assembly, potentially exposing subsequent logic to spurious data. |
| OS-LZ-SUG-01 | `_assignJobToVerifiers` falsely assumes sorted input. |
| OS-LZ-SUG-02 | `getFeeOnSend` for `*FeeLibs` should be marked `payable` in order to pay `pricefeed` fees. |
| OS-LZ-SUG-03 | `UlnBase::_verify` fails to behave idempotently, contradicting assumptions made in the verifier network. |
| OS-LZ-SUG-04 | Checking hashes prior to signature verification decreases gas consumption on reused hashes. |
| OS-LZ-SUG-05 | Use `Call` instead of `Transfer` while sending native tokens to avoid potential future gas cost issues. |



# OS-LZ-SUG-00 | Out Of Bounds Memory Read

## Description

`_splitSignature` lacks bounds checks when performing memory reads via inline assembly. As a result, subsequent logic may be exposed to unintended data if the function is provided with truncated signature data.

*packages/layerzero-v2/evm/messagelib/contracts/uln/MultiSig.sol* SOLIDITY

```solidity
function _splitSignature(
    bytes memory _signatures,
    uint256 _pos
) internal pure returns (uint8 v, bytes32 r, bytes32 s) {
    assembly {
        let signaturePos := mul(0x41, _pos)
        r := mload(add(_signatures, add(signaturePos, 0x20)))
        s := mload(add(_signatures, add(signaturePos, 0x40)))
        v := and(mload(add(_signatures, add(signaturePos, 0x41))), 0xff)
    }
}
```

When extracting `r`, `s`, and `v` from the raw signature bytes, no checks are in place to ensure that `signaturePos` is within a valid range. Consequently, if a signature has a length not a multiple of 65, it may result in reading out-of-bounds data and passing it to subsequent cryptographic operations. The security implications of this issue are minimal, as the values `r`, `s`, and `v` are already entirely configurable. Nevertheless, we recommend addressing this concern to enhance robustness.

## Remediation

Implement bounds checks for signature splitting.

## Patch

Fixed in 20d29fd.

# OS-LZ-SUG-01 | Data Inconsistency

## Description

`_assignJobToVerifiers` assumes that verifier options are sorted based on verifier indices. However, `groupdVerifierOptionsByIdx` only aggregates options and does not necessarily sort them. As a result, unsorted options may be overlooked, resulting in unintended behavior.

*packages/layerzero-v2/evm/messagelib/contracts/uln/UlnBase.sol* SOLIDITY

```solidity
function _ulnSend(
    /* ... */
) internal returns (uint totalFee, address executor, uint maxMsgSize) {
    /* ... */
    (bytes[] memory optionsArray, uint8[] memory verifierIndices) =
    ↪  VerifierOptions.groupVerifierOptionsByIdx(
        verifierOptions
    );
    (totalFee, verifierFees) = _assignJobToVerifiers(
        /* ... */
        optionsArray,
        verifierIndices
    );
}
function _assignJobToVerifiers(
    /* ... */
) internal returns (uint totalFee, uint[] memory verifierFees) {
    /* ... */
    for (uint i = 0; i < verifiersLength; ) {
        /* ... */
        unchecked {
            /* ... */
            if (_verifierIds.length > 0 && i == _verifierIds[j]) {
                options = _optionsArray[j++];
            }
        }
        /* ... */
        unchecked {
            ++i;
        }
    }
}
```

`groupVerifierOptionsByIdx` parses a serialized option string (`verifierOptions`) and groups them by verifier indices. For each element in `optionsArray`, the corresponding verifier index is the element within `verifierIndices` at the same position. However, the implementation of `_assignJobVerifiers`, which uses the condition `i == _verifierIds[j]`, assumes that `_verifierIds` is monotonically increasing. Consequently, if the input options are unordered, part of the options may be dropped by `_assignJobToVerifiers`.

## Remediation

Re-implement `_assignJobToVerifiers` by searching for the corresponding options for each whitelisted verifier through a linear scan within each iteration. It is crucial to note that this change will increase the gas consumption complexity from $O(n)$ to $O(n^2)$, but it will relax the requirement for sorted input.

## Patch

Fixed in b4db8c6.

# OS-LZ-SUG-02 | Make Function Payable

## Description

`getFeeOnSend` in `ExecutorFeeLib` and `VerifierFeeLib` should be `payable` to enable the payment of `pricefeed` fees.

*packages/layerzero-v2/evm/messagelib/contracts/ExecutorFeeLib.sol* SOLIDITY

```
function getFeeOnSend(
    FeeParams memory _params,
    IExecutor.DstConfig memory _dstConfig,
    bytes calldata _options
) external returns (uint fee) {
    /* ... */
    uint priceFeedFee = ILayerZeroPriceFeed(_params.priceFeed).getFee(
        _params.dstEid,
        _params.calldataSize,
        totalGas
    );
    (
        uint totalGasFee,
        uint128 priceRatio,
        uint128 priceRatioDenominator,
        uint128 nativePriceUSD
    ) = ILayerZeroPriceFeed(_params.priceFeed).estimateFeeOnSend{value:
    ↪   priceFeedFee}(
            _params.dstEid,
            _params.calldataSize,
            totalGas
        );
}
```

The invocation of `LayerZeroPriceFeed.estimateFeeOnSend` plays a crucial role in calculating the execution and validation fees, as both parameters heavily rely on the price of the native asset on the destination chain. To accommodate for the funds required when fetching data from the price feed, the `getFeeOnSend` function should be payable.

## Remediation

Modify `getFeeOnSend` to be payable.

## Patch

Fixed in 81c4470.

# OS-LZ-SUG-03 | ULN Idempotent Behavior

## Description

The verifier network will skip checking hashes for certain idempotent functions to save gas.

*packages/layerzero-v2/evm/messagelib/contracts/uln/VerifierNetwork.sol* SOLIDITY

```
/// @dev to save gas, we don't check hash for some functions (where replaying
↪ won't change the state)
/// @dev for example, some administrative functions like changing signers, the
↪ contract should check hash to double spending
/// @param _functionSig function signature
/// @return true if should check hash
function _shouldCheckHash(bytes4 _functionSig) internal pure returns (bool) {
    // never check for these selectors to save gas
    return
        _functionSig != IUltraLightNode.verify.selector && // replaying won't
↪ change the state
        _functionSig != this.verifyAndDeliver.selector && // replaying calls
↪ deliver on top of verify, which will be rejected at uln if not deliverable
        _functionSig != ILayerZeroUltraLightNodeV2.updateHash.selector; //
↪ replaying will be revert at uln
}
```

However, ensuring that the called functions are idempotent is essential. Problematically, current implementations of `IUltraLightNode.verify` are not actually idempotent.

*packages/layerzero-v2/evm/messagelib/contracts/uln/UlnBase.sol* SOLIDITY

```
function _verify(bytes calldata _packetHeader, bytes32 _payloadHash, uint64
↪ _confirmations) internal {
    hashLookup[keccak256(_packetHeader)][_payloadHash][msg.sender] =
↪ _confirmations;
    emit PayloadSigned(msg.sender, _packetHeader, _confirmations,
↪ _payloadHash);
}
```

A malicious admin could replay previous messages to decrease the amount of confirmations associated with a particular packet and payload.

## Remediation

Check that the new confirmation amount exceeds the original, similar to the original `UltraLightNodeV2`. As an additional precaution, consider a strict whitelist of targets to avoid any risk of hash collision.

# OS-LZ-SUG-04 | Hash Check Ordering

## Description

The verifier network undergoes a multi-step verification process when receiving payloads in quorumChangeAdmin. In particular, it performs both a signature verification and a hash duplication check.

*packages/layerzero-v2/evm/messagelib/contracts/uln/VerifierNetwork.sol* SOLIDITY

```solidity
function quorumChangeAdmin(ExecuteParam calldata _param) external {
    require(_param.expiration > block.timestamp, "Verifier: expired");
    require(_param.target == address(this), "Verifier: invalid target");

    // generate and validate hash
    bytes32 hash = hashCallData(_param.target, _param.callData,
↪   _param.expiration);
    require(verifySignatures(hash, _param.signatures), "Verifier: invalid
↪   signatures");
    require(!usedHashes[hash], "Verifier: hash already used");

    usedHashes[hash] = true;
    _grantRole(ADMIN_ROLE, abi.decode(_param.callData, (address)));
}
```

However, signature verification is significantly more expensive. As a general best practice, performing the cheaper check first may make sense.

## Remediation

Reorder the two assertions to save gas on error scenarios.

## OS-LZ-SUG-05 | Use Of Transfer

### Description

Layerzero utilizes `transfer` instead of `call` to send native tokens. Generally, `call` is the recommended function due to `transfer` having a hard cap of 2300 gas, which may fail if operation code gas changes in the future.

`transfer` is currently utilized four times within the code, in `SimpleMessageLib.recoverToken`, `EndpointV2.withdrawFee`, `EndpointV2.lzReceive` and `MessagingComposer.lzCompose`.

An example of `transfer` in `MessagingComposer.lzCompose`:

*packages/layerzero-v2/evm/protocol/contracts/MessagingComposer.sol* SOLIDITY

```solidity
function lzCompose(
    address _sender,
    address _composer,
    bytes32 _guid,
    bytes calldata _message,
    bytes calldata _extraData
) external payable returns (bool success, bytes memory reason) {
    [...]
    if (success) {
        emit ComposedMessageReceived(_sender, _composer, _guid,expectedHash,
    ↪   msg.sender);
    } else {
        if (msg.value > 0) {
            payable(msg.sender).transfer(msg.value);
        }
    }
    [...]
}
```

### Remediation

Utilize `call` instead of `transfer`.

# A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

| | |
|---|---|
| **Critical** | Vulnerabilities that immediately lead to loss of user funds with minimal preconditions<br>Examples:<br>• Misconfigured authority or access control validation<br>• Improperly designed economic incentives leading to loss of funds |
| **High** | Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.<br>Examples:<br>• Loss of funds requiring specific victim interactions<br>• Exploitation involving high capital requirement with respect to payout |
| **Medium** | Vulnerabilities that could lead to denial of service scenarios or degraded usability.<br>Examples:<br>• Malicious input that causes computational limit exhaustion<br>• Forced exceptions in normal user flow |
| **Low** | Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.<br>Examples:<br>• Oracle manipulation with large capital requirements and multiple transactions |
| **Informational** | Best practices to mitigate future security risks. These are classified as general findings.<br>Examples:<br>• Explicit assertion of critical internal invariants<br>• Improved input validation |



# B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

