LayerZero Verifier

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` `VerifierNetwork` programs. This assessment was conducted between July 14th and August 25th, 2023. For more information on our auditing methodology, see Appendix B.

This report represents an interim report prepared specifically for the `VerifierNetwork` and adjacent components.

Key Findings

Over the course of this audit engagement, we produced 5 findings total.

In particular, we noted incorrect multisig signature replay protection (OS-LZ-ADV-01) and provided a discussion of potential centralization risks (OS-LZ-ADV-02) to ensure proper liveness and safety guarantees even with the compromise of various privileged parties.

We also provided suggestions around hardening UltraLightNode (ULN) verification behavior (OS-LZ-SUG-00) and minor improvements for gas efficiency (OS-LZ-SUG-01).

02 | Scope

The source code was delivered to us in a git repository at github.com/LayerZero-Labs/monorepo. This audit was performed against commit 7a3ce88.

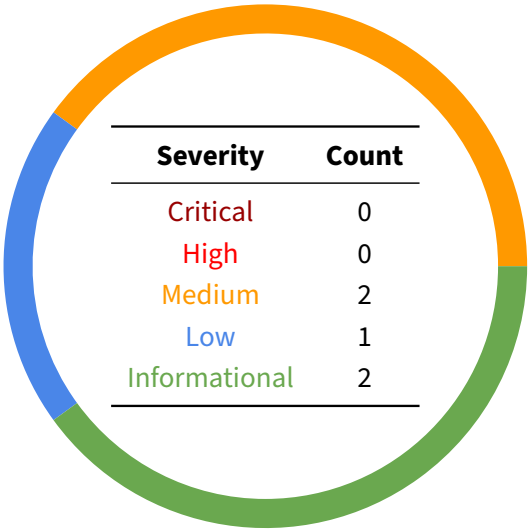
A brief description of the programs is as follows.

VerifierNet	Description
`VerifierNetwork`	`VerifierNetwork` is the contract that emits messages for relays to collect and send to remote chains and attest messages from remote chains and deliver to `EndpointV2`.

03 | Findings

Overall, we reported 5 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 don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	2
Low	1
Informational	2

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	`VerifierNetwork` `multisig` signatures used in `execute` and `quorumChangeAdmin` may be replayed.
OS-LZ-ADV-01	Medium	Resolved	The `execute` function of `VerifierNetwork` implements replay protection in an incorrect manner.
OS-LZ-ADV-02	Low	Resolved	We have analyzed the implications of compromised roles and provided suggestions to mitigate their risks.

OS-LZ-ADV-00 [med] | VerifierNetwork Signature Replay

Description

The original implementation of `VerifierNetwork` `multisig` signed message does not include unique `VerifierNetwork` ids. Thus it is possible to replay signatures across `VerifierNetwork`.

Remediation

Add a unique id to the message hash for verification.

packages/layerzero-v2/evm/messagelib/contracts/uln/VerifierNetwork.sol 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-01 [med] | Incorrect Multisig Replay Protection

Description

The `execute` function in `VerifierNetwork` implements replay protection in an incorrect manner as it permanently blocks the execution of transactions, even if they resulted in failure. To prevent replay attacks, the `usedHashes` mapping is utilized. 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

```
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 is not done by providing an invalid signature. This may result in a single compromised `ADMIN_ROLE` to cause a denial of service, compromising the robustness of the system.

Remediation

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

Patch

Fixed in 3bb3e16 and 93211fc.

OS-LZ-ADV-02 [low] | Potential Centralization Risks

Description

We enumerated the potential consequences for centralized role in the following components: `VerifierNetwork`.

#	Component	Role	Consequences
1	`VerifierNetwork`	`DEFAULT_ADMIN_ROLE`	An attacker may add any `MessageLib` to deny list, which may be exploited to incur denial-of-service.
2	`VerifierNetwork`	`ADMIN_ROLE`	An attacker may call the `execute` function in a malicious manner by reordering the transactions or partially dropping them.

We provide further analysis of each centralized roles here.

#	Analysis
1	This `DEFAULT_ADMIN_ROLE` is properly set to `address(0x0)` in current contract. We only include it here for completeness.
2	The ability for `ADMIN_ROLE` to drop messages introduces a DoS risk which should be mitigated. Contrarily, message reordering attacks is less of an immediate threat, since none of the currently implemented functionalities are affected by it. However, to prevent future upgrades / expansions from breaking this invariant and leading to exploitable issues, we still advise enforcing message ordering for additional resilience.

Remediation

The centralization risk represents a trade-off between security and design complexity, rather than being a risk that can be entirely prevented.

Possible ways to reduce risks includes not setting `DEFAULT_ADMIN_ROLE` for `VerifierNetwork` and allowing `multisig` to assign additional `admin` when the old one is compromised and not properly relaying messages.

Patch

Resolved in cb3a118 and 361d741.

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 antipatterns and could lead to security issues in the future.

ID	Description
OS-LZ-SUG-00	`UlnBase::_verify` fails to behave idempotently, contradicting assumptions made in the verifier network.
OS-LZ-SUG-01	Checking hashes prior to signature verification allows for decreased gas consumption on reused hashes.

OS-LZ-SUG-00 | ULN Idempotent Behavior

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 important. 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

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

OS-LZ-SUG-01 | Hash Check Ordering

The verifier network undergoes a multistep 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

```
    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, it may make sense to perform the cheaper check first.

Remediation

Consider reordering the two assertions to save gas on error scenarios.

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