

LayerZero EndpointV2 Security Analysis Report and Formal Verification Properties



certora

December 2023

Table of Contents

Table of Contents.....	2
Summary.....	3
Summary of findings.....	4
Disclaimer.....	4
Main Issues Discovered.....	5
Issue-01: getUlnConfig() can revert and prevent functionality.....	5
Issue-02: A single reverting verifier interface can kill a whole transaction.....	5
Issue-03: Owner of contract can DoS send().....	6
Issue-04: A verifier can gain double fees.....	6
Issue-05: Contract is left with no admins.....	7
Issue-06: Zero address is admin.....	7
Issue-07: Contradiction of ALLOWLIST and DENYLIST ROLES.....	7
Issue-08: Grieving by admin.....	8
Issue-09: Race Condition.....	8
Issue-10: Zero hash is deliverable.....	9
Issue-11: Unexpected behavior for optional oracle configuration.....	9
Issue-12: Permanent DoS for MultiSig Oracle.....	9
Issue-13: Bad Admin can un-verify message.....	10
Note-14: Admins can revoke each other.....	10
Note-15: Function name is misleading.....	10
Formal Verification.....	11
Notations.....	11
VerifierNetwork.sol properties.....	11
EndpointV2.sol properties.....	14
UltraLightNode301.sol properties.....	16
UltraLightNode302.sol properties.....	17
Formal Verification - revision.....	19
Issue-A1: An oApp can DoS a call to send() by another oApp.....	19
Notations.....	20
EndpointV2.sol properties.....	20
UltraLightNode301.sol properties.....	22
UltraLightNode302.sol properties.....	23

Summary

This document describes the specification and verification of the new **LayerZero protocol** using the Certora Prover and manual code review findings. The work was undertaken from **2nd August 2023** to **6th September 2023**. The latest commit that was reviewed is [7a3ce889](#).

The following contracts list is included in the **scope**:

```
packages/layerzero-v2/evm/protocol/contracts/libs
packages/layerzero-v2/evm/protocol/contracts/messagelib/libs/BitMaps.sol
packages/layerzero-v2/evm/protocol/contracts/messagelib/libs/PacketV1Codec.
sol
packages/layerzero-v2/evm/protocol/contracts/messagelib/BlockedMessageLib.s
ol
packages/layerzero-v2/evm/protocol/contracts/EndpointV2.sol
packages/layerzero-v2/evm/protocol/contracts/MessageLibManager.sol
packages/layerzero-v2/evm/protocol/contracts/MessagingChannel.sol
packages/layerzero-v2/evm/protocol/contracts/MessagingComposer.sol
packages/layerzero-v2/evm/protocol/contracts/MessagingContext.sol

packages/layerzero-v2/evm/messagelib/contracts/uln/libs
packages/layerzero-v2/evm/messagelib/contracts/uln/uln301/MessageLibBaseE1.
sol
packages/layerzero-v2/evm/messagelib/contracts/uln/uln301/TreasuryFeeHanlde
r.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/uln301/UltraLightNode301
.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/uln302/UltraLightNode302
.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/MultiSig.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/UlnBase.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/UlnConfig.sol
packages/layerzero-v2/evm/messagelib/contracts/uln/VerfierNetwork.sol
packages/layerzero-v2/evm/messagelib/contracts/MessageLibBase.sol
packages/layerzero-v2/evm/messagelib/contracts/MessageLibBaseE2.sol
packages/layerzero-v2/evm/messagelib/contracts/OutboundConfig.sol
packages/layerzero-v2/evm/messagelib/contracts/Worker.sol
packages/layerzero-v2/evm/oapp/contracts/OApp.sol
```

The contracts were verified for Solidity version 0.8.19.

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all Solidity contracts. During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below.

Summary of findings

The table below summarizes the issues discovered during the audit, categorized by severity.

Severity	Total discovered	Total fixed	Total acknowledged
High	1	0	1
Medium	4	0	1
Low	5	0	3
Informational	2		2
Total (High, Medium, Low)	10	0	5

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

Main Issues Discovered

Issue-01: `getUlnConfig()` can revert and prevent functionality

Severity: High

Probability: Medium

Category: DoS / Unexpected revert

File(s): `UlnConfig.sol`

Bug description: The check for the total number of verifiers in `getUlnConfig()` isn't accurate and can lead to a state that will make `getUlnConfig()` revert in the future unexpectedly.

Exploit scenario: A call to `getUlnConfig()` can revert if the default configuration is incompatible with `getUlnConfig()` requirements. For example, assume that the initial count of configured mandatory oracles is 254, and the count of optional ones is taken from the default configuration, which is initially one. Then, the default configuration updates the optional oracles count from one to three. Now, the total number of oracles is $257 > \text{max}(\text{uint8})$ which will cause a revert on `getUlnConfig()`.

Implications: Unexpected revert for every function which calls `getUlnConfig()`, including `send()`.

LayerZero's response: "This is fine. the oapp has the ability to recover from any configuration failure and that's what matters."

Issue-02: A single reverting verifier interface can kill a whole transaction

Severity: Medium

Probability: Medium

Category: DoS / Unexpected revert

File(s): `UlnBase.sol`

Bug description: When calling `send()` from the endpoint, the ULN library calculates the total amount of fees accrued to the verifiers by summing over the results received from the ``assignJob`` function in the ``_assignJobToVerifiers`` function. Since the function loops over all verifiers, including the optional ones, it takes one bad verifier whose function will revert to kill the entire transaction.

Exploit scenario: If the default configuration of the library includes one bad verifier as such, apps who use this configuration will not be able to send messages and without any ability to exclude that verifier from the configuration, which is reserved to the contract owner exclusively.

LayerZero's response: "The app can override the configuration."



Issue-03: Owner of contract can DoS send()

Severity: Medium

Probability: Medium

Category: DoS / Unexpected revert

File(s): `UltraLightNode301.sol`, `MessageLibBaseE1.sol`, `EndpointV2.sol`

Bug description: The call to `send()` by an OApp in `UltraLightNode301` or in the `EndpointV2` may be unexpectedly rejected/reverted if the owner of the sending library decides to change/ set anew some basic configurations of the contract by calling either `setTreasury()` or `setLayerZeroToken()`.

Exploit scenario: The owner of the `send` library `UltraLightNode301` or of the `Endpoint` contract has the power to change the global settings of the library which can affect the success of a sending procedure used by an OApp. Changing the fee structure before a `send()` function is invoked by an OApp can lead to the former call to revert, since conditions expected to be met originally in `send()` will no longer hold. In the case of changing the fee structure, or the alternative payment token, the oApp isn't obliged to quote the fee price before sending a message, hence it is expected to hold the sufficient balance of the fee payment token before sending a message, being unaware of sudden changes to the fee structure.

Implications: However unintentionally, or non-maliciously, the owner of the sending library contract could stop OApps from sending messages.

LayerZero's response:

Issue-04: A verifier can gain double fees

Severity: Medium

Probability: probable

Category: Unfair distribution of protocol fees.

File(s): `UlnConfig.sol`

Bug description: A check for addresses duplicates between the mandatory and optional verifiers is missing.

Exploit scenario: Any configuration, whether the default or an application-custom one, allows a verifier address to appear both on the optional and mandatory lists, since duplicates are checked only within each list separately. The contract owner, or the app that configures its own list, can decide to include the same verifier in both lists, while giving the permission that each verifier gets a single "slice" of the message-sending fees.

Implications: A single verifier contract can get a double amount of fees.

LayerZero's response: "The app can override the configuration."

Issue-05: Contract is left with no admins

Severity: Medium

Probability: Low

Category: Loss of control

File(s): VerifierNetwork.sol

Bug description: There is no admin.

Exploit scenario: The `revokeRole` function is called by the last admin with parameters `ADMIN_ROLE` and their own address, thus renouncing their own role, hence the contract is left with zero admins.

Property violated: `AtLeastOneAdmin`

Implications: The quorum will have to choose a new admin.

LayerZero's response: "quorum can grant new admin with `quorumChangeAdmin`."

Issue-06: Zero address is admin

Severity: Low

Probability: Low

Category: Invalid state

File(s): VerifierNetwork.sol

Bug description: Account with zero address could be an admin.

Exploit scenario: Can only happen during construction, in the `grantRole` or the `quorumChangeAdmin` function.

Property violated: `adminsNonZero`

Implications: Giving admin rights to zero address has no benefits other than saving gas for the missing check. In this case, we can always set a new admin with `quorumChangeAdmin` and revoke the admin role from the zero address if needed.

LayerZero's response: "we check if address is zero in constructor before setting. won't check for `grantRole`, if such action is performed, suppose it is intentional."

Issue-07: Contradiction of ALLOWLIST and DENYLIST ROLES

Severity: Low

Probability: Low

Category: State contradiction

File(s): VerifierNetwork.sol

Bug description: Account can have both `DENYLIST` and `ALLOWLIST` roles.

Exploit scenario: By calling `grantRole` with one of the said roles when the other is already acquired by the user.

Property violated: `cannotBeBothAllowedAndDenied`

Implications: Currently this is not a major problem, but if we checked for instance for `ALLOWLIST` first in `Worker:onlyAcl`, some entity could get approval even when on `DENYLIST`. The issue might become severe if a future check for the account being on those lists is reversed in order, where the `ALLOWLIST` role is checked first.

LayerZero's response: "`ALLOWLIST` and `DENYLIST` are intended to work together with `onlyAcl` and `allowlistSize`, will add a comment that the roles should only be used with `onlyAcl`."

Issue-08: Grieving by admin

Severity: Low

Probability: Low

Category: Grieving

File(s): `VerifierNetwork.sol`

Bug description: Contract's admin can reduce confirmations for a specific message by replaying an old confirmation message for that message.

Implications: Minor annoyance to the signers.

LayerZero's response: "Grieving interfaces are fine. The quorum can swap the bad admin out."

Issue-09: Race Condition

Severity: Low

Probability: Low

Category: Grieving

File(s): `EndpointV2.sol`

Bug description: Delivered message can be received by anyone since `lzReceive()` is permissionless. Therefore, when an OApp wants to skip a delivered message via the `clear` method, it is effectively in a race condition with any user who wants to deliver the message (assuming the call to the receiver does not fail).

Implications: Unwanted messages can't be safely dropped.

LayerZero's response: "Yes the expected behaviour is atomic transaction."

Issue-10: Zero hash is deliverable

Severity: Low

Probability: Low

Category: Wrong input allowed

File(s): `EndpointV2.sol`



Bug description: The zero hash has a special significance in the endpoint, an indication that the message payload is empty. However, there is no restriction in delivering a zero hash as the payload, which results in a bad transition of the system (for example, the inbound nonce might decrease). While the outcome of an actual message payload hash being zero is completely unlikely, it can still be delivered if a group of verifiers decides to verify it intentionally.

Implications: Nonce mechanism and payload delivery is broken.

LayerZero's response:

Issue-11: Unexpected behavior for optional oracle configuration

Severity: Low

Probability: Low

Category: Unexpected behavior

File(s): `ULnBase.sol`

Bug description: When setting *optionalOraclesThreshold* to zero for a default or custom configuration, the threshold is effectively one (As in, we need at least a single optional oracle that would verify the message to set the message to a "Deliverable" state).

Implications: Unexpected behavior

LayerZero's response: "noted. just need careful configuration. not changing the code for this."

Issue-12: Permanent DoS for MultiSig Oracle

Severity: High

Probability: Medium

Category: Denial of Service

File(s): `MultiSigOracle.sol`

Bug description: An admin can by mistake (or on purpose) remove all other admins, including himself, thus locking the fees and any further executions in the contract.

Implications: Permanent DoS

LayerZero's response: Fixed.

Issue-13: Bad Admin can un-verify message

Severity: Medium

Probability: Low

Category: Griefing



File(s): VerifierNetwork.sol

Bug description: By design, an Admin can replay verify() messages. Therefore, if the quorum had sent a verification message with M confirmations, and then with N confirmations ($N > M$), the Admin could replay the first message and reduce the number of confirmations. Remark: This only works for EndpointV2, as in the original Endpoint, verifications (via updateHash) are strictly monotone.

Implications: Unintentional behavior of the Verifier Network

LayerZero's response: Fixed.

Note-14: Admins can revoke each other

Severity: Informational

Probability: High

Category: Fragile governance

File(s): VerifierNetwork.sol

Bug description: Admins can revoke other admins which could be a design flaw depending on the way you want to handle admin roles.

LayerZero's response: Acknowledged.

Note-15: Function name is misleading

Severity: Informational

Probability: High

Category: Misleading documentation

File(s): VerifierNetwork.sol

Bug description: Documentation for `quorumChangeAdmin` as well as its name says its purpose is to change admin, but it actually adds admin and doesn't remove any existing admin.

LayerZero's response: Acknowledged.

Formal Verification

The structure of properties:

1. <notation> <property description> (<property name in spec code>)

Function names (and signatures) shall be written in Source code Pro font size 11 with the gray highlight, e.g., `foo(uint256)`

Notations

✓ Indicates the rule is formally verified.

✗ Indicates the rule is violated.

⌚ Indicates the rule is timing out.

VerifierNetwork.sol properties

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- View functions filtering: Rules checking state changes of all available functions do not check view functions.
- Optimistic fallback: all fallback functions were assumed to be empty.
- The following functions have been ignored during this verification, we assume that those never revert and can return arbitrary values:
 - ILayerZeroPriceFeed
 - estimateFeeByEid(uint32,uint,uint)
 - estimateFeeOnSend(uint32,uint,uint)
 - IMessageLib
 - withdrawFee(address, uint)
 - ILayerZeroUltraLightNodeV2
 - withdrawNative(address, uint)
 - IUltraLightNode
 - deliver(bytes, bytes32)
 - deliverable(bytes, bytes32)
 - verify(bytes, bytes32, uint64)

Properties

1. ✓ Signers can only be changed with the `setSigner` function.
(onlySetSignerCanChangeSigners)
2. ✗ No account can be both on `DENYLIST` and `ALLOWLIST`.
(cannotBeBothAllowedAndDenied) - [Issue 07](#)

3. ❌ The zero address cannot be admin. (`adminsNonZero`) - [Issue 06](#)
4. ✅ Quorum is never zero. (`quorumIsNonzero`)
5. ✅ There cannot be zero signers. (`signerSizesNonZero`)
6. ❌ There is always at least one admin. (`AtLeastOneAdmin`, violated by `revokeRole`) - [Issue 05](#)
7. ✅ `allowListSize` is equal to the number of users that have role `ALLOWLIST`. (`allowListSizeEqualsListCount`)
8. ✅ The `execute` function fails after passed `expiration`. (`executeExpiredFails`)
9. ✅ After successful execution of `grantRole(role, account)` the account has the role. (`grantRoleWorks`)
10. ✅ After successful execution of `revokeRole(role, account)` the account does not have the role. (`revokeRoleWorks`)
11. ✅ Only an admin can grant the admin role and only current contract can grant a basic¹ role. (`grantRoleProtected`)
12. ✅ Only admin can revoke admin role and only current contract can revoke a basic role (`revokeRoleProtected`)
13. ✅ The `grantRole(role, account)` function does not revert when one of the following is met: (`grantRoleDoesntRevert`)
 - `msg.sender` is admin and the `role` is `ADMIN_ROLE`.
 - `msg.sender` is the contract and the `role` is one of the basic roles.
14. ✅ The `revokeRole(role, account)` function does not revert when one of the following is met: (`revokeRoleDoesntRevert`)
 - `msg.sender` is admin and the `role` is `ADMIN_ROLE`.
 - `msg.sender` is the contract and the `role` is one of the basic roles.
15. ✅ Quorum changes only with `setQuorum`. (`quorumChangesOnlyWithSetQuorum`)
16. ✅ Admin changes only with `quorumChangeAdmin`, `grantRole` or `revokeRole`. (`changeAdminOnlyWithRoleChangingFunctions`)
17. ✅ The `verifySignatures` function doesn't revert. (`verifySignaturesDoesNotRevert`)

¹ By basic roles we consider: `ALLOWLIST`, `DENYLIST` and `MESSAGE_LIB_ROLE`.















- 18. ✓ The `execute` function does not execute when there isn't a quorum of distinct signatures. (`executeOnlyWithEnoughSignatures`)
- 19. ✓ The `execute` function does not execute any job that was already executed. (`executeDoesntExecuteDuplicateJobsSamePatch`, `executeDoesntExecuteDuplicateJobsDifferentPatch`)
- 20. ✓ Only the zero address has the `DEFAULT_ADMIN_ROLE`. (`NobodyHasZeroRole`)
- 21. ✓ Admin role of any role is `DEFAULT_ADMIN_ROLE`. (`EveryRoleAdminIsZeroRole`)
- 22. ✓ Any function can change only one role of only one account. (`oneRoleChangeAtATime`)
- 23. ✓ Calling `quorumChangeAdmin(B)` with parameters `B` cannot block `quorumChangeAdmin(A)` from being executed if `A != B`. (`impossibleToFrontrunQuorumChangeAdminWithOtherParams`)
- 24. ✗ Admins cannot revoke each other (`adminsCannotKickEachotherOut`) - [Note 14](#)
- 25. ✓ Setting signer true cannot block `quorumChangeAdmin`. (`setSignerCannotBlockChangeAdmin`)
- 26. ✓ `execute` does not revert. (`executeDoesntRevert`)
- 27. ✓ It is always possible to change admin (`alwaysPossibleToChangeAdmin`, `alwaysPossibleToChangeAdminAfterChangeOfQuorum`)

EndpointV2.sol properties

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- No overflow of nonces: we assume transactions cannot revert because of overflows of the nonce increments (All nonces $< 2^{63}$).
- Optimistic fallback: all fallback functions were assumed to be empty.
- We ignored any behavior regarding the apps that are interacting with the endpoint and assumed them to be neutral.

Properties

1.  The inbound nonce never decreases². - [Issue-10](#)
2.  The outbound nonce can only increase by 1 at a time (or stay the same).
3.  It's only possible to deliver a message with a nonce larger than the lazy nonce.
4.  The inbound nonce can only be changed by its receiver, or by calling `deliver()` for a message that is deliverable.
5.  No action performed by any actor can prevent calling `send()` by another actor. - [Issue-03](#)
6.  `lzReceive()` can only succeed once for the same set of arguments.
7.  `lzCompose()` can only succeed once for the same set of arguments.
8.  Only the owner of the contract can call the 'set default' functions.
9.  `transferOwnership()` reverts if and only if the `msg.sender` is not the owner or the new owner address is the zero address.
10.  Only the receiver of a message can successfully clear its payload (calling `clear()`).
11.  No single action can change both the inbound and outbound nonce at once.
12.  Any action can only change the inbound nonce for at most a single triplet of receiver, src eid and sender at a time.
13.  Any action can only change the outbound nonce for at most a single triplet of sender, dst eid and receiver at a time.
14.  Any action can change the inbound payload hash for at most a single group of receiver, src eid, sender and nonce at a time.

² If one restricts the delivered hash from being the zero hash, the rule is verified.



15. ✓ Any action can change the deliverable status of a message for a single pair of origins `src eid`, senders and recipients at a time.
16. ✓ If the deliverable status of two different nonces change together, then those nonces correspond to messages already delivered (with payload hash).
17. ✓ The function `deliver()` is idempotent: the second call to the function with the same arguments doesn't change any state variables and never reverts.
18. Integrity of the function `deliver()` :
- ✓ Doesn't change the output value of `deliverable()` of another origin.
 - ✓ Sets the correct payload hash into the right message origin and receiver.
 - ✗ If the value of `hasPayloadHash()` has changed, then it must have been false before (there was no payload hash)³. - [Issue-10](#)
19. ✓ An action which changes the value of the `isValidReceiveLibrary(address receiver, ...)` could only be done if the `msg.sender` is the receiver.
20. ✓ Mutual-independence of `clear()` : clearing a message cannot prevent another call to clearing a message if:
- The `msg.senders` are different.
 - The `msg.senders` are identical and the origins are different.
21. ✓ It's impossible to call `clear()` twice for the same `msg.sender` and origins.
22. ✓ Integrity of the function `lzReceive()` :
- ✓ If the success variable returned is false, then the `msg.sender` ETH balance doesn't change.
 - ✓ If the success variable returned is true, then the `msg.sender` ETH balance is decreased by `msg.value`.
 - ✓ The inbound nonce should not change at all.
23. Uniqueness of GUID:
- ✓ A call to `clear()` with a given set of arguments can only succeed for a unique GUID.
 - ✓ A call to `lzReceive()` with a given set of arguments can only succeed for a unique GUID.

³ If one restricts the delivered hash from being the zero hash, the rule is verified.

24. ✓ If a message is received, it cannot be cleared afterwards (a call to `clear()` by the receiver with the same origins and guid, reverts).
25. ✓ Only previous calls to `lzReceive()`, `clear()` and `deliver()` can prevent `clear()` from succeeding.
26. ✓ The only way to extract tokens from Endpoint (i.e. decrease its balance) is by calling `recoverToken()`.
27. ✓ When calling `send()` or `sendWithAlt()`, the `msg.sender` pays the correct amount of tokens to the library, as indicated by the messaging receipt for both tokens (native and LZ).

UltraLightNode301.sol properties

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- We assume empty options and Worker options lists for `send()`.
- We assume that the fees mapping never overflows, i.e. no single one could be distributed more than `max_uint256` fees.

Properties

1. ✓ The following functions could only be called by the admin:
 - `transferOwnership()`
 - `setAddressSize()`
 - `setTreasury()`
 - `setLayerZeroToken()`
 - `setDefaultConfig()`
 - `renounceOwnership()`
2. ✗ The sum of mandatory and optional verifiers for the configuration of every app and eid is bounded by 255 (max uint8). - [Issue-01](#)
3. ✓ The optional verifiers threshold is never greater than the optional verifiers count for every app and eid.
4. ✓ Any action can only change the `verified()` and `verifyConditionMet()` status for a single pair of header and payload hashes.
5. ✓ Integrity of the `verifyConditionMet()`:
 - If the mandatory verifiers list is non-empty, then if at least one of the verifiers didn't verify, the condition is not met.
 - If there are only optional verifiers, and the threshold is equal to the count, then if at least one of the verifiers didn't verify, the condition is not met.
6. ✗ Configurations are mutually independent: changing the configuration of one type cannot affect the possibility of changing the configuration of a different type. - [Issue-01](#)

7. ❌ The function `getUlnConfig()` should never revert. - [Issue 01](#)
8. ✅ The change in sum of worker fees is equal to difference in native balance of the ULN contract between after and before calling `send()`.
9. ✅ No action can front-run (DoS) `withdrawFee()`.
10. ❌ `send()` pathway is immutable: no call by a different party (including the owner) can make the sending procedure of an oApp fail. - [Issue-03](#)
11. ✅ `send()` could never be front-run and make it revert by another call to `send()` from another `msg.sender`.
12. ❌ If an app had at least one DVN, no action can turn its number of DVNs to zero.

UltraLightNode302.sol properties

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- We assume empty options and Worker options lists for `send()`.
- We assume that the fees mapping never overflows, i.e. no single one could be distributed more than `max_uint256` fees.

Properties

1. ✅ The following functions could only be called by the admin:
 - `transferOwnership()`
 - `setAddressSize()`
 - `setTreasury()`
 - `setLayerZeroToken()`
 - `setDefaultConfig()`
 - `renounceOwnership()`
2. ❌ The sum of mandatory and optional verifiers for the configuration of every app and eid is bounded by 255 (`max uint8`). - [Issue-01](#)
3. ✅ The optional verifiers threshold is never greater than the optional verifiers count for every app and eid.
4. ✅ The `verifyConditionMet()` status of a configuration can only be changed by one of the configuration verifiers.
5. ✅ `verified()` status can only be changed by the verifier.
6. ✅ Any action can only change the `verified()` and `verifyConditionMet()` status for a single pair of header and payload hashes.
7. ✅ Integrity of the `verifyConditionMet()`:
 - a. If the mandatory verifiers list is non-empty, then if at least one of the verifiers didn't verify, the condition is not met.
 - b. If there are only optional verifiers, and the threshold is equal to the count, then if at least one of the verifiers didn't verify, the condition is not met.

8. ❌ The function `getUlnConfig()` should never revert. – [Issue 01](#)
9. ✅ No action can front-run (DoS) `withdrawFee()`.
10. ✅ Once a config is set, it can always be reset afterwards.
11. ✅ The change in sum of worker fees is equal to the fee value in the message receipt given by `send()`.
12. ✅ `send()` could never be front-run and make it revert by another call to `send()` from another `msg.sender`.
13. ❌ If an app had at least one DVN, no action can turn its number of DVNs to zero.

Formal Verification – revision

This section shows the revised formal verification of the EndpointV2 and the ULN verification libraries of the new code version given in the following commit of the LayerZero monorepo repository: [c71f996b](#).

Below we only list the status of the rules that were tested in the previous section, with slight modifications which account for interface and code structure changes.

If a new bug was introduced or a rule became violated with respect to the previous verification round, we mark it explicitly.

Issue-A1: An oApp can DoS a call to send() by another oApp

Severity: Medium

Probability: Low

Category: Front-running / race condition

File(s): EndpointV2.sol

Bug description: If an oApp chooses to pay in LZ token when sending a message through the EndpointV2 after having sent the LZ token directly to the contract, another oApp could front-run the call and steal the fee for its own use.

Exploit scenario: When sending a message through EndpointV2, the contract checks whether the sender has supplied enough LZ tokens to cover for the calculated fees. The supplied amount of tokens is the contract ERC20 balance for that token. In the case where an oApp decides to send the LZ tokens to the Endpoint separately (not in the same transaction) from actually sending the message, another oApp could take advantage of the free tokens in the Endpoint contract by sending a message for itself, causing also the intended original message to be declined, since the Endpoint would not have enough balance to cover the original message fees.

The attacker which front-runs the original send transaction could simply take the fees for himself by choosing any desired refund address to send the excess balance to.

Looking at the implementation of the oApp, as an example of using the EndpointV2, it seems that there is no guarantee that the application makes sure to send the tokens in the same transaction as calling send(). We note that the previous version of the code (the one verified originally in this report) did make use of a transfer call from the app to the library.

Implications: LZ tokens could be taken away from an oApp by any other oApp if the first doesn't provide the tokens safely.

LayerZero's response: "yes they need to do it atomically.", "We will add it to the doc if it is not there already"

Notations

✓ Indicates the rule is formally verified.

✗ Indicates the rule is violated.

⌚ Indicates the rule is timing out.

EndpointV2.sol properties

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
 - No overflow of nonces: we assume transactions cannot revert because of overflows of the nonce increments (All nonces $< 2^{63}$).
 - Optimistic fallback: all fallback functions were assumed to be empty.
 - We ignored any behavior regarding the apps that are interacting with the endpoint and assumed them to be neutral.
 - `ILayerZeroReceiver.allowInitializedPath()` returns true always.
1. ✓ The inbound nonce never decreases.
 2. ✓ The outbound nonce can only increase by 1 at a time (or stay the same).
 3. ✓ It's only possible verifying a message with origin.nonce larger than the lazy nonce **or if it already has payload but not executed.**
 4. ✓ The inbound nonce can only be changed by its receiver or its delegate, or by calling `verify()` for a message that is verifiable.
 5. ✗ No action performed by any actor can prevent calling `send()` by another actor.
 6. ✗ **`send()` performed by any actor cannot prevent calling `send()` by another actor. [Issue-A1](#) - new bug introduced.**
 7. ✓ `lzReceive()` can only succeed once for the same set of arguments.
 8. ✓ `lzCompose()` can only succeed once for the same set of arguments.
 9. ✓ Only the owner of the contract can call the 'set default' functions.
 10. ✓ `transferOwnership()` reverts if and only if the `msg.sender` is not the owner or the new owner address is the zero address.

11. ✓ Only the app or its delegate can successfully clear a message payload (calling `clear()`).
12. ✓ No single action can change both the inbound and outbound nonce at once.
13. ✓ Any action can only change the inbound nonce for at most a single triplet of receiver, src eid and sender at a time.
14. ✓ Any action can only change the outbound nonce for at most a single triplet of sender, dst eid and receiver at a time.
15. ✓ Any action can change the inbound payload hash for at most a single group of receiver, src eid, sender and nonce at a time.
16. ✓ Any action can change the verifiable status of a message for a single pair of origins src eid, senders and recipients at a time.
17. ✓ If the verifiable status of two different nonces change together, then those nonces correspond to messages already verified (with payload hash).
18. ✓ The function `verify()` is idempotent: the second call to the function with the same arguments doesn't change any state variables and never reverts.
19. Integrity of the function `verify()`:
 - a. ✓ Doesn't change the output value of `verifiable()` for another origin.
 - b. ✓ Sets the correct payload hash into the right message origin and receiver.
 - c. ✓ If the value of `hasPayloadHash()` has changed, then it must have been false before (there was no payload hash).
20. ✓ An action which changes the value of the `isValidReceiveLibrary(address receiver, ...)` could only be done if the by the app (receiver) or its delegate.
21. ✓ Mutual-independence of `clear()`: clearing a message cannot prevent another call to clearing a message if:
 - a. The other msg.sender isn't authorized.
 - b. Both are authorized for the same app and the origins are different.
22. ✓ It's impossible to call `clear()` twice for the same msg.sender and origins.
23. ✓ Integrity of the function `lzReceive()`:



- a. ☒ The msg.sender ETH balance is decreased by msg.value.
 - b. ☒ The inbound nonce should not change at all.
24. Uniqueness of GUID:
- a. ☒ A call to `clear()` with a given set of arguments can only succeed for a unique GUID.
 - b. ☒ A call to `lzReceive()` with a given set of arguments can only succeed for a unique GUID.
25. ☒ If a message is received, it cannot be cleared afterwards (a call to `clear()` by the receiver with the same origins and guid, reverts).
26. ☒ Only previous calls to `lzReceive()`, `clear()`, `nilify()`, `burn()` and `verify()` can prevent `clear()` from succeeding.
27. ☒ The only way to extract tokens from Endpoint (i.e. decrease its balance) is by calling `recoverToken()`.
28. ☒ When calling `send()` the msg.sender pays the correct amount of tokens to the library, as indicated by the messaging receipt for both tokens (native and LZ).

UltraLightNode301.sol properties

The properties enlisted here were tested for the contracts `SendUln301.sol` and `ReceiveUln301.sol`

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- We assume empty options and Worker options lists for `send()`.

Properties

1. ☒ The following functions could only be called by the admin:
 - `transferOwnership()`
 - `setAddressSize()`
 - `setTreasury()`
 - `setLayerZeroToken()`
 - `setDefaultConfig()`
 - `renounceOwnership()`
2. ☒ The sum of mandatory and optional verifiers for the configuration of every app and eid is bounded by 255 (max uint8).
3. ☒ The optional verifiers threshold is never greater than the optional verifiers count for every app and eid.

4. ☒ Any action can only change the `verified()` and `checkVerifiable()` status for a single pair of header and payload hashes.
5. ☒ Integrity of the `checkVerifiable()` status:
 - a. If the mandatory verifiers list is non-empty, then if at least one of the verifiers didn't verify, the condition is not met.
 - b. If there are only optional verifiers, and the threshold is equal to the count, then if at least one of the verifiers didn't verify, the condition is not met.
6. ☒ Configurations are mutually independent: changing the configuration of one type cannot affect the possibility of changing the configuration of a different type.
7. ☒ The function `getUlnConfig()` should never revert.
8. ☒ The change in sum of worker fees is equal to difference in native balance of the ULN contract between after and before calling `send()`.
9. ☒ No action can front-run (DoS) `withdrawFee()`.
10. ☒ `send()` pathway is immutable: no call by a different party (including the owner) can make the sending procedure of an oApp fail.
11. ☒ Setting one config type cannot prevent from setting a different type (for the same App).

UltraLightNode302.sol properties

The properties enlisted here were tested for the contracts `SendUln302.sol` and `ReceiveUln302.sol`

Assumptions

- Loop unrolling: We assume any loop can have at most 3 iterations.
- We assume empty options and Worker options lists for `send()`.

Properties

1. ☒ The following functions could only be called by the admin:
 - `transferOwnership()`
 - `setAddressSize()`
 - `setTreasury()`
 - `setLayerZeroToken()`
 - `setDefaultConfig()`
 - `renounceOwnership()`
2. ☒ The sum of mandatory and optional verifiers for the configuration of every app and eid is bounded by 255 (max uint8).
3. ☒ The optional verifiers threshold is never greater than the optional verifiers count for every app and eid.
4. ☒ The `checkVerifiable()` status of a configuration can only be changed by one of the configuration verifiers, excluding `commitVerification()`

- a. If one calls `commitVerification()`, then the status could only turn to non-verifiable.
- 5. ☒ `verified()` status can only be set to true by the verifier.
- 6. ☒ Any action can only change the `verified()` and `checkVerifiable()` status for a single pair of header and payload hashes.
- 7. ☒ Integrity of the `checkVerifiable()`:
 - a. If the mandatory verifiers list is non-empty, then if at least one of the verifiers didn't verify, the condition is not met.
 - b. If there are only optional verifiers, and the threshold is equal to the count, then if at least one of the verifiers didn't verify, the condition is not met.
- 8. ☒ The function `getUlnConfig()` should never revert.
- 9. ☒ No action can front-run (DoS) `withdrawFee()`.
- 10. ☒ `send()` pathway is immutable: no call by a different party (including the owner) can make the sending procedure of an oApp fail.
- 11. ☒ The change in sum of worker fees is equal to the fee value in the message receipt given by `send()`.