LayerZero - V2 Protocol Contracts

Audit Report

Prepared by Zugzwang LLC (Christoph Michel)

December 13, 2023.

Contents

1 Introduction

LayerZero is an omnichain interoperability protocol designed for lightweight message passing across chains.

1.1 Scope of Work

The auditors were provided with a GitHub repository at commit hash 2526ec0 (December 2nd 2023).

The task was to audit the LayerZero V2 EVM **protocol contracts** (found in `layerzero-v2/evm/protocol/contracts`), consisting of the following files with their `sha1` hashes:

File	SHA1
`EndpointV2.sol`	`cd2d866d1317fe8f3e4faad9db2b66881e0c38dc`
`EndpointV2Alt.sol`	`bc92e194b7740663dc8da79df914ba5b000eb48d`
`MessageLibManager.sol`	`4f1c2f37249649b15d454717345ed23b9a80bcbd`
`MessagingComposer.sol`	`5443641f668732652ff6ae5484865839e6e78430`
`MessagingContext.sol`	`612fbd38e43a1521e5d34ff9e18b2ccee17b7572`
`MessagingChannel.sol`	`015e02ffe15156ce87cfedb5c70a343c08c2b260`
`libs/Errors.sol`	`b66f28a0c371399a7fbedd491c3048264c3ba855`
`libs/GUID.sol`	`76dd4e90783c3238e13b2b715dcb1aa91c8a299d`
`libs/Transfer.sol`	`ab3139485f679c91fe2563a42b77558574eedc5a`
`libs/AddressCast.sol`	`12001ce981da98ac94de8d82f012f416d69090db`
`libs/CalldataBytesLib.sol`	`7d5bb1f1697dca32b6f08de47e45f32413d57f53`
`messagelib/BlockedMessageLib.sol`	`168e72c1fbf162326d72b694c987311a75b64fed`
`messagelib/SimpleMessageLib.sol`	`ec90f6eeb77c79593ff42b3b6f4c39f01ceda340`
`messagelib/libs/ExecutorOptions.sol`	`4aa1eea7169ec99d1eead0057d1dc4b71196d2f2`

File	SHA1
messagelib/libs/BitMaps.sol	650bd4426cee9eae4ac6d1663384fbf6c353008b
messagelib/libs/PacketV1Codec.sol	25a5281e3cff1aff24bf203793f757e98979c934
interfaces/IMessageLib.sol	8976de90bccf8acc6d161c0303a78cef05e43589
interfaces/ILayerZeroEndpointV2.sol	87805d87301cc3e099ac6661604c6b755fdcc380
interfaces/ILayerZeroComposer.sol	85521db7c1a8d6eef26b63fde7ab2ab252d52294
interfaces/IMessagingChannel.sol	891afe60f996850e4bd4f88b0ec05abe6088f41d
interfaces/ISendLib.sol	b4aa04c69ac9debbf4b420e33916c607f9646d04
interfaces/IMessagingContext.sol	1436f076c284a80ccc970ea315c1626f2c5a3a85
interfaces/IMessageLibManager.sol	9ea57bc7b19401b3c31996bbdc64607aea4f60cd
interfaces/IMessagingComposer.sol	754891dabd983ac5976e6292ba77b9868258bd6d
interfaces/ILayerZeroReceiver.sol	6c66013130045c3756c118cfbf10f7e84829665d

The rest of the repository was out of the scope of this audit. Explicitly, the code for the V2 EVM `messagelib` and `oapp` was not part of this audit. The documentation for version 2 of the protocol was not fully ready at the time of the audit and the specifications might have changed.

> The team confirmed that the `SimpleMessageLib` contract provided in the repo will *not* be used as a send or receive library contract as it is insecure; it is only used for testing. The issues found therein are therefore only of informational severity.

Fixes & Changes

The auditors were provided with commit hash 2de05cd (December 12th 2023) containing fixes to part of the findings and the following additional features:

- OApp authorization delegation logic for OApp-facing `Endpoint` functions
- `nilify` and `burn` functions are added
- re-verification of old, not yet executed messages is enabled
- initializing the `endpoint` with a different `owner` in the constructor

1.2 Security Assessment Methodology

The smart contract's code is scanned both manually and automatically for known vulnerabilities and logic errors that can lead to potential security threats. The conformity of requirements (e.g., specifications, documentation, White Paper) is reviewed as well on a consistent basis.

1.3 Auditors

Zugzwang LLC (Christoph Michel)

2 Severity Levels

We assign a risk score to the severity of a vulnerability or security issue. For this purpose, we use 4 *severity levels* namely:

INFORMATIONAL

Informational issues are generally subjective in nature or potentially associated with topics like “best practices” or “readability”. As a rule, informational issues do not indicate an actual problem or bug in the code. The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

LOW

Low-severity issues are generally objective in nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is a clear reason not to.

MEDIUM

Medium-severity issues are bugs or vulnerabilities. These issues may not be directly exploitable or may require certain conditions in order to be exploited. If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations that make the system exploitable.

HIGH

High-severity issues are directly exploitable bugs or security vulnerabilities. If unaddressed, these issues are likely or guaranteed to cause major problems or, ultimately, a full failure in the operations of the contract.

3 Discovered issues

3.1 Message execution options not part of message hash (high)

Context: EndpointV2.sol#L173

After a message has been verified anyone can execute the message. While initiating a cross-chain message with `send(..., options)` allows one to define `MessageOptions` that include the `gasLimit` and the `value` the message should be executed with, these options are not part of the verified message hash. An executor can just ignore the options and use their own `gasLimit`, `msg.value`, and `extraData` when calling `endpointV2.lzReceive`.

This can lead to issues as there is usually no easy way for the receiving OApp to verify the integrity of the provided `msg.value`, gas limit, and `extraData`. An OApp function might be executed receiving fewer `msg.value` than anticipated, a wrong gas limit, or malicious extra data. All of these can change the result and side effects of the execution.

Recommendation

Ideally, the message execution options should be part of the verified message hash such that these values can be verified in `lzReceive`. For example, by including the `value` and a `minGasLimit`. This would reduce the trust assumptions of the executor to the trust assumptions of the verifiers. These parties currently have very different trust assumptions as anyone can be the executor.

> Note: The same issue exists for `lzCompose`.

Response

> Acknowledged. The options are worker functions that are not a part of the protocol, if an app would like the options validated, then they should pack it into their message on send and validate them on receive.

3.2 `endpoint.executable` returns `Executed` for skipped nonces (medium)

Context: EndpointV2.sol#330

The `endpoint.executable(origin, receiver)` function returns the execution state for a receiver's nonce. It wrongly returns `ExecutionState.Executed` for:

- The `0` nonce.
- Nonces that are skipped by the receiver with an `endpoint.skip(origin.srcEid, origin.sender, origin.nonce)` call.

This could be problematic if the sender wants to verify that their message was successfully executed, calls `endpoint.executable()` for their message and it returns `Executed` while in reality the receiver skipped this message and the message side effects are never applied.

Recommendation

Consider returning `NotExecutable` for nonce `0` as the first legitimate message will start at nonce 1. There is currently no state for `Skipped`. Consider renaming the `Executed` state to `ExecutedOrSkipped`. If a further distinction between skipped and executed states is desired, the code needs further changes, like a special payload hash value of `bytes32(1)` for skipped or executed messages.

Response

> Acknowledged. The `executable` function is only used off-chain.

Note that after the introduction of a `burn` function, an `Executed` state may mean any of the following:

- executed
- skipped
- burned

Furthermore, after the introduction of the `nilify` function, nilified messages can be shown as `Executable` even though they are semantically not executable (and neither are they practically executable as executing them would require finding the preimage of the 256 1-bits hash).

3.3 `endpoint.verify` allows verifying `bytes32(0)` payloads (`medium`)

Context: EndpointV2.sol#146

The empty payload `bytes32(0)` is used as the special value to indicate:

1. The nonce has not been used for verification yet
2. The nonce was already executed (execution clears the payload)
3. The nonce was skipped.

Verifying a nonce with this special value of `bytes32(0)` therefore leads to issues in the contract and breaks some implicit invariants that are true when `verify` is called with a non-zero payload hash:

1. The `_inboundNonce()` does not increase when verifying `bytes32(0)` for the current `nonce=_inboundNonce()`.
2. Non-cleared, executable messages can become unexecutable again, see `endpoint.executable()`. This should never be the case. `verify` with a `bytes32(0)` payload leads to a buggy clearance of a nonce as it does not advance the `lazyInboundNonce`. Assume nonces `[1,2,3]` are all verified and executable with `lazyInboundNonce = 0`: Calling `verify(nonce=2, hash=bytes32(0))` will reset the `_inboundNonce()` to `1`. Nonce 3's message will become non-executable and cannot be executed through `lzReceive()`. Note that a normal `clear(nonce=2)` does not invalidate any other nonces as it advances the `lazyInboundNonce` to the cleared nonce and therefore does not reduce the `_inboundNonce()`.

Recommendation

A `bytes32(0)` payload hash cannot be executed as there is no known pre-image to the `0` hash and using `verify` to overwrite an existing hash as a `clear` function is flawed. Therefore, we see no use case of verifying the special `bytes32(0)` value, consider reverting in `endpoint.verify` if `_payloadHash == bytes32(0)`.

Response

> Fixed.

3.4 PUSH0 might not be supported on all chains (`low`)

Context: EndpointV2.sol#3

The contracts use solidity version `0.8.22` which supports using the new `PUSH0` opcode (supported since `0.8.20`, introduced in the `shanghai` upgrade). Not all chains support the new instruction already, therefore deployment or execution of the Endpoint and related contracts might break.

Recommendation

Consider using Solidity 0.8.19 (or below) or set the EVM Version to `paris`. This enforces the same bytecode on all chains.

Response

> Acknowledged. Deployment will fail if the opcode is unknown, so we can use a lower version when needed.

3.5 Anyone can emit failure events (low)

Context: EndpointV2.sol#L194, MessagingComposer.sol#L78

The `lzReceiveAlert` and `lzComposeAlert` functions can be called by anyone and emit `LzReceiveAlert` and `LzComposeAlert` events. These events are used by the LayerZero Scan indexer to display failed messages in its frontend. Depending on how the backend indexes these events, there's the potential for a variety of web2 / phishing scams. For example, the receiver (attacker) requires a payment from the sender (victim). The attacker triggers the failure event and when the indexer picks it up, the attacker shows it as proof of a failed payment to the sender. The sender sends the payment a second time. As the initial payment did not actually fail but was just delayed, the attacker gets paid twice.

Recommendation

Make sure the off-chain indexing code is robust:

- Match the event parameters against existing source messages.
- Ideally, it would take the event's `executor` into account and their trustworthiness.

Response

> Acknowledged. The `msg.sender` is included, and the frontend can filter based on the app-specified executor.

3.6 Failed executions can be retried forever (low)

Context: EndpointV2.sol#174

The `lzReceive` function only succeeds if the underlying `ILayerZeroReceiver(_receiver).lzReceive(...)` call to the receiver succeeds. It's important to note that the `Endpoint` contract has no notion of a *failed execution*. Executions can be retried indefinitely. Messages can be `clear`ed but only by the receiver of the message, the sender has no way to invalidate an execution that currently reverts.

It could be that an execution that currently reverts might succeed under different conditions. This poses an additional risk as there's no way for the sender to guarantee that their failed message does not suddenly become successfully executable in the future. For example, if a cross-chain token transfer

from the sender's smart contract wallet on the destination chain (receiver) was scheduled and it fails for some reason (for example, the transfer receiver reverts), they need to be careful about sending another payment.

Recommendation

The receiving smart contracts can implement their own invalidation protocol as part of the message but it might be beneficial to implement this functionality directly into the base protocol. For example, by defining a deadline for each packet, or marking failed executions as actually "failed" in storage if the execution reverted. This requires further protocol changes to ensure the execution was called with the executor options specified by the sender, like the minimum gas to use and the native value for the call. These options are currently not verified, see the "Message execution options not part of message hash" issue.

Response

> Acknowledged. If the app doesn't want it retried, they can clear it.

Note that only the receiving OApp (or its delegate) can clear it.

3.7 LZ token payments when sending messages must be atomic (`low`)

Context: EndpointV2.sol#L280

When paying in LZ tokens via the `payInLzToken` flag for `endpoint.send`, the contract uses its LZ token balance for the payment and sends the fees to the sender's send library contract. The transfer of funds to the endpoint and the call to `send` must happen atomically, otherwise, there is the chance of an attacker frontrunning the `send` call and using the victim's funds, which are already in the contract, as the payment.

Recommendation

Make sure the transfer and the call to `send` happens atomically. This is easy to achieve for smart contracts OApps. For EOAs, consider developing periphery contracts that allow performing both actions in a single transaction.

Response

> Acknowledged.

3.8 `set(Default)ReceiveLibraryTimeout` still requires `_newLib` when clearing (`low`)

Context: MessageLibManager.sol#L208, MessageLibManager.sol#L286

The `setReceiveLibraryTimeout` and `setDefaultReceiveLibraryTimeout` functions allow clearing the current grace period but still require and check a `_newLib` address parameter. This parameter is checked to satisfy `onlyRegistered(_lib)`, `isReceiveLib(_lib)`, `onlySupportedEid(_lib, _eid)` but the deletion code path completely ignores it. This parameter is also emitted in the `ReceiveLibraryTimeoutSet` and `DefaultReceiveLibraryTimeoutSet` events at the end. It's possible to delete the current timeout-receive-library by passing in an arbitrary `_lib` parameter that satisfies the conditions. It does not need to be the old timeout lib parameter. The emitted event will have a `_lib` parameter that is essentially meaningless when clearing the timeout.

Recommendation

Consider using different functions to set and clear the receive library timeouts as their parameters, checks and events are too different.

Response

> Acknowledged.

3.9 `PacketV1Codec.receiverB20` ignores dirty upper bits (`low`)

Context: PacketV1Codec.sol#L90

The `PacketV1Codec.receiverB20()` function reads 32 bytes and uses the lower 20 bytes as the address, ignoring the other 12 upper bytes. This leads to situations where packet encodings are not unique. For example, in `SimpleMessageLib.validatePacket(bytes calldata packetBytes)`, several `packetBytes` encodings will decode to the same packet verification. This is usually an undesirable property, as `encode` and `decode` should be inverses of each other but here `encode(decode(packetBytes))!= packetBytes`.

Recommendation

Consider requiring all 12 upper bytes to be zero in `receiverB20()`.

Response

> Acknowledged.

3.10 Messages can already be verified through the default library before deployment (low)

Context: MessageLibManager.sol#L98

Even before the receiving OApp is deployed, messages to its future contract address can already be verified through the default receive library. Even if the OApp sets a different default library and config in the constructor at deployment, there might still already be messages that had been verified under the potentially weaker and undesired default receive library assumptions. This is because all addresses automatically opt-in to the default receive library. There's currently no easy way to get the library that verified a message after the message has already been verified.

Recommendation

OApps need to be aware that messages might have already been verified under a different receive library even if they opted out of the default library at deployment.

Response

> Fixed. Added a `ILayerZeroReceiver(_receiver).allowInitializePath(_origin)` call to the receiver in `Endpoint.verify` when receiving messages while `lazyInboundNonce == 0`.

3.11 Using the default library comes with auto-updates (low)

Context: MessageLibManager.sol#L98

OApps using the default library (represented through the `DEFAULT_LIB = address(0)` special value) automatically opt in to any upgrades to the default library. If the endpoint owner is compromised and sets a malicious default library, the OApp will also adopt it and can receive forged messages.

Recommendation

OApps that want to use the default library but opt out of auto-updates, should explicitly set the current default library as the receive library through the `setReceiveLibrary` function.

Response

> Acknowledged. Apps can lock in the library if they don't want to subscribe to updates.

3.12 LZ fee token can be changed (low)

Context: EndpointV2.sol#L215

The `_lzToken` that is used for fee payments can be changed by the Endpoint `owner` at any time using the `setLzToken` function. A user might mine a transaction and pay with the old token and expect a refund. All these tokens can be lost if the `setLzToken` is mined before and a griefer send enough new LZ tokens to the contract to cover the victim's message fees. In this case, `_suppliedLzToken` does not revert and the users' old LZ token are lost.

Recommendation

It's an unlikely edge-case and we don't recommend adding more code to address it as stuck funds can be recovered via `recoverToken`. However, we recommend users to not rely on the refund feature for huge payments and only send small amounts to the contract, if not the exact `quote`d amount.

Response

> Acknowledged.

3.13 `sendContext` collision with `NOT_ENTERED` (informational)

Context: MessagingContext.sol#L18

The `sendContext(uint32 _dstEid, address _sender)` modifier sets the current `_sendContext` to:

```
uint256 private constant NOT_ENTERED = 1;

_sendContext = (uint256(_dstEid) << 160) | uint160(_sender);
```

This is supposed to prevent re-entrancy by ensuring the current `_sendContext` storage variable does not equal the `NOT_ENTERED = 1` constant. However, technically it's possible to create a collision with the `NOT_ENTERED` value when the modifier is called with `sendContext(_dstEid=0, _sender=address(1))`. Such a call could re-enter the `send` function.

As the modifier is currently only called with `msg.sender` as the second argument this does currently not have any impact. We still recommend properly fixing it by making it impossible to end up with a collision for all input values.

Recommendation

The `_sendContext` is a 256-bit value but the `sendContext` modifier only uses the 192 (160 + 32) least-significant bits. Consider choosing a `NOT_ENTERED` value of `1 << 192` to avoid collisions for all possible inputs of `sendContext`.

Response

> Acknowledged.

3.14 `AddressCast.toBytes` might truncate addresses (`informational`)

Context: MessagingContext.sol#L18

The `AddressCast.toBytes(bytes32 _addressBytes32, uint256 _size)` function will only take the `_size` lower-bytes of the `_addressBytes32`. Some non-zero address bytes might be cut off if a `_size` parameter is used that is too small.

Recommendation

Consider adding additional validation. If there are any non-zero dirty bits in the upper 32 `_size` bytes, something went wrong with the decoding and one could consider reverting instead of continuing with a truncated address.

Response

> Acknowledged.

3.15 Trust assumptions on verifiers (`informational`)

Context: EndpointV2.sol#L147

Messages on the receiving chain are all verified by the *receive library* contract set by the OApp. This can be the default *receive library* provided by LayerZero or another registered one. The privileged roles of the receive library contract (verifiers) must be fully trusted as they can verify arbitrary messages.

Recommendation

Users must be aware of the operators of their receive library contract and the risks involved with these parties. Note that the default library contracts at any time and users using the default libraries will automatically be upgraded to the new default libraries.

Response

> Acknowledged.

3.16 Anyone can verify messages in `SimpleMessageLib` if no whitelist is set (`informational`)

Context: SimpleMessageLib.sol#L67

The `SimpleMessageLib.validatePacket` calls `endpoint.verify` to verify packets. If no `whitelist` is set anyone can validate any packets. If this is the default receive library, anyone can inject fake messages for all users of the default receive library. Note that the `whitelist` is not set in the constructor and if the library is already registered as the default, anyone could frontrun `whitelist` calls.

Recommendation

We don't see a use case for allowing everyone to validate arbitrary packets. Consider reverting in the function if `whitelist` is not set.

Response

> Acknowledged.

3.17 Paying fees in `LZ` token for `SimpleMessageLib` still requires native payment (`informational`)

Context: SimpleMessageLib.sol#81

When paying for fees for `send`ing a message, the `SimpleMessageLib.send` function always requires the native fee payment, even if `_payInLzToken` is set.

```
1 fee = MessagingFee(nativeFee, _payInLzToken ? lzTokenFee : 0);
```

There's no reason to ever pay in LZ tokens as it just requires an LZ token fee payment on top of the normal native fee payment.

Recommendation

Rethink the fees for the `SimpleMessageLib`. Consider taking no native fee if `_payInLzToken` is true:

```
fee = MessagingFee(_payInLzToken ? 0 : nativeFee, _payInLzToken ?
    lzTokenFee : 0);
```

Response

> Acknowledged.

3.18 send can be frontrun with a default options change (`informational`)

Context: SimpleMessageLib.sol#85

The `SimpleMessageLib` send library can define default options (like gas limit and `msg.value`) that will be used in case a user does not specify any options. These default options can be changed at any time. A user might be frontrun and have to pay for more gas and value usage than expected.

Recommendation

If users don't want to blindly subscribe to default option changes of their send lib, they should explicitly set the options to the current default options.

Response

> Acknowledged.

3.19 Miscellaneous (`informational`)

- MessageLibManager.sol#L176, MessageLibManager.sol#L205: The `onlyOwner` modifier sometimes appears to the left, sometimes to the right. This influences which checks the functions performs first and what the revert error will be. Consider using the same order of modifiers on all functions. If the owner check should be applied first, it should appear as the left-most modifier in this case.

- MessagingComposer.sol#L33: The "@dev can not re-entrant" comment is not entirely accurate. Nothing prevents one from re-entering this function with another compose message. What was meant here was probably to say "@dev can not replay same message".
- MessageLibManager.sol#L256: Typo: "which would should" -> "which should"
- IMessageLibManager.sol#L23: Typo: "`ReceiveLibraryTimoutSet`" -> "`ReceiveLibraryTimeoutSet`"
- ISendLib.sol#L31: `SimpleMessageLib` does not explicitly implement all functions of `ISendLib` (`getConfig`, `setConfig`, `setTreasury`). They will only be implicitly caught in the `fallback`. It might be desirable to inherit the `ISendLib` interface and explicitly implement them with the revert for readability and type-checks.
- EndpointV2.sol#L95: The `payNative` function gives control to the `_refundAddress`. While the `send` function has a reentrancy-protection through the `sendContext` modifier, we still recommend following the Checks-Effects-Interactions pattern and performing the `payNative` call at the end, **after** the `_payToken` call. Currently, the contract's LZ token balance has not been updated yet when control is handed to the refund address. Consider doing the same in `messagelib` contracts.
- MessageLibManager.sol#L231: Checking if a library supports an eid is only done when setting it. Depending on what library is used this value might change afterwards.
- SimpleMessageLib.sol#L35: The `InvalidEndpoint` event is never used. Consider removing it.
- SimpleMessageLib.sol#L38: The `TransferFailed` event is never used. Consider removing it.
- SimpleMessageLib.sol#L54: The `defaultOptions` are not defined initially. It's unclear what verifiers and executors will do when a message is sent with no options specifying the gas limit or the value.

Response

> Acknowledged.

Disclaimer

This audit is a time-restricted security review based on the scope and snapshot of the code mentioned in the introduction. The contracts used in a production environment may differ drastically. Neither did this audit verify any deployment steps or multi-signature wallet setups. Audits cannot provide a guarantee that all vulnerabilities have been found, nor might all found vulnerabilities be completely mitigated by the project team. An audit is not an endorsement of the project or the team, nor guarantees its security. No third party should rely on the audit in any way, including for the purpose of making any decisions about investing in the project.