Zellic

December 13, 2024

Prepared for
Ryan Zarick
Isaac Zhang
LayerZero Labs

Prepared by
GunHee Ahn
Aaron Esau
Daniel Lu
Nan Wang
Zellic

LayerZero TON
TON Application Security Assessment

Contents

About Zellic 5

1. Overview 5

1.1. Executive Summary 6

1.2. Goals of the Assessment 6

1.3. Non-goals and Limitations 6

1.4. Results 7

2. Introduction 7

2.1. About LayerZero TON 8

2.2. Methodology 8

2.3. Scope 9

2.4. Project Overview 13

2.5. Project Timeline 14

3. Detailed Findings 14

3.1. Bypassable packet hash check 15

3.2. Wrong permissions on `ulnConnectionCommitPacket` 17

3.3. Broken authentication in `endpointCommitPacket` 19

3.4. Possible runtime crash when setting hash lookups 21

3.5. Risks of malformed object construction 23

3.6. Improper eviction from send queue clogs channel 29

3.7. Unsafe options version check in OApp 30

3.8. Unhandled nonce state after `nilify` 32

3.9. Ineffective short circuit in `_optimizedNonceCommittable` 34

3.10. Lack of quorum and verifier count validation 36

3.11. Confusing else branch in `compareObjectFields` 37

3.12. Missing reinitialization safeguard in baseHandler/baseStorage framework 40

3.13. Unused opcodes without permission checks 42

3.14. Incorrect field parsing in `parseGasLimitsPrices` 45

4. Discussion 48

4.1. Possibility of messages being sent out of order 49

4.2. Warning regarding message sizes 50

4.3. Lack of support for composed message options 52

4.4. Risks of flexible object structure 52

4.5. Handler reuse in channel 53

4.6. Type alias error in `Channel::sendRequestQueue` 54

4.7. Differences from EVM specification 56

4.8. Disclaimer regarding send-queue congestion 56

4.9. Gas optimizations 56

5. System Design 57

5.1. Risk of collusion 58

5.2. Risks of default configurations 59

5.3. Threat model regarding channels 61

6. Assessment Results **63**

6.1. Disclaimer 64

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for LayerZero Labs from September 16th until January 24th [1]. During this engagement, Zellic reviewed LayerZero TON's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a malicious OApp cause other OApps' paths to be blocked?
- Can the protocol censor messages?
- Can malicious DVNs permanently block an OApp?
- Can messages be forged for an arbitrary source address?
- What centralization risks does LayerZero TON have?
- What differences does TON have from the EVM specification and white paper?
- Are there race conditions when sending messages between contracts?
- Can fees be recovered at each step in the flow of messages if there is an out-of-gas (OOG) condition?
- If there are failures at any step, is it possible to recover permissionlessly?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Worker feelib
- Infrastructure relating to the project
- Front-end components
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, the fact that the code was not frozen during our audit prevented us from achieving the coverage we would have preferred.

[1] Note that there were multiple schedule changes and auditor assignment changes during the assessment period. For more details, please refer to the Project Timeline table.

Given this, and based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes.

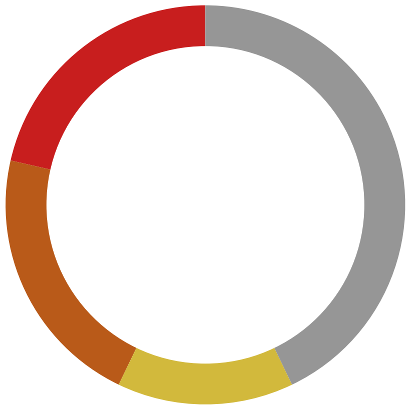
1.4. Results

During our assessment on the scoped LayerZero TON contracts, we discovered 14 findings. Three critical issues were found. Three were of high impact, two were of medium impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of LayerZero Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	3
🟧 High	3
🟨 Medium	2
🟩 Low	0
⬜ Informational	6



2. Introduction

2.1. About LayerZero TON

LayerZero Labs contributed the following description of the project:

> LayerZero is a technology that enables applications to move data across blockchains, uniquely supporting censorship-resistant messages and permissionless development through immutable smart contracts.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

LayerZero TON Contracts

Type	FunC
Platform	TON

Target	LayerZero TON funC++
Repository	https://github.com/LayerZero-Labs/monorepo ↗
Version	`a1dee3385752c129e324cd951c21f5f56ebccca0`

Programs	`abstract/contractMainAbstract.fc`
	`abstract/handlerAbstract.fc`
	`actions/event.fc`
	`actions/deploy.fc`
	`actions/call.fc`
	`actions/dispatch.fc`
	`actions/utils.fc`
	`actions/sendJettons.fc`
	`actions/payment.fc`
	`txnContext.fc`
	`stdlib.fc`
	`contractMain.fc`
	`testutils.fc`
	`dataStructures/AddressList.fc`
	`dataStructures/PipelinedOutOfOrder.fc`
	`dataStructures/DeterministicInsertionCircularQueue.fc`
	`stringlib.fc`
	`utils.fc`
	`handlerCore.fc`
	`classlib.fc`
	`constants.fc`
	`baseInterface.fc`

Target	LayerZero TON Endpoint
Repository	https://github.com/LayerZero-Labs/monorepo ↗
Version	`a1dee3385752c129e324cd951c21f5f56ebccca0`

Programs	`channel/callbackOpcodes.fc`
	`channel/storage.fc`
	`channel/main.fc`
	`channel/handler.fc`
	`controller/storage.fc`
	`controller/main.fc`
	`controller/handler.fc`
	`core/abstract/protocolHandler.fc`
	`core/abstract/protocolMain.fc`
	`core/baseStorage.fc`
	`endpoint/storage.fc`
	`endpoint/main.fc`
	`endpoint/handler.fc`

Target	LayerZero TON Workers
Repository	https://github.com/LayerZero-Labs/monorepo ↗
Version	`a1dee3385752c129e324cd951c21f5f56ebccca0`

Programs	`dvn/storage.fc`
	`dvn/main.fc`
	`dvn/handler.fc`
	`executor/storage.fc`
	`executor/main.fc`
	`executor/handler.fc`
	`core/abstract/workerHandler.fc`
	`core/workerCoreStorage.fc`

Target	LayerZero TON ULN
Repository	https://github.com/LayerZero-Labs/monorepo ↗
Version	`13d74b901c71f809bc3358505bbf15146e7407fe`

Programs	`uln/interface.fc`
	`uln/storage.fc`
	`uln/main.fc`
	`uln/handler.fc`
	`msgdata/DvnFeesPaidEvent.fc`
	`msgdata/InitUlnManager.fc`
	`msgdata/UlnSendConfig.fc`
	`msgdata/InitUln.fc`
	`msgdata/UlnWorkerFeelibInfo.fc`
	`msgdata/TreasuryFeeBps.fc`
	`msgdata/UlnWorkerFeelibBytecode.fc`
	`msgdata/UlnReceiveConfig.fc`
	`msgdata/UlnEvents.fc`
	`msgdata/InitUlnConnection.fc`
	`msgdata/RentRefill.fc`
	`msgdata/SetAdminWorkerAddresses.fc`
	`msgdata/Attestation.fc`
	`msgdata/ExecutorFeePaidEvent.fc`
	`msgdata/UlnSend.fc`
	`msgdata/UlnWorkerFeelibEvents.fc`
	`msgdata/VerificationStatus.fc`
	`msgdata/UlnVerification.fc`
	`ulnManager/interface.fc`
	`ulnManager/storage.fc`
	`ulnManager/main.fc`
	`ulnManager/handler.fc`
	`callbackOpcodes.fc`
	`ulnConnection/interface.fc`
	`ulnConnection/utils.fc`
	`ulnConnection/storage.fc`
	`ulnConnection/main.fc`
	`ulnConnection/handler.fc`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 16.1 person-weeks. The assessment was conducted by four consultants over the course of 19 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Project Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

GunHee Ahn
Engineer
gunhee@zellic.io

Aaron Esau
Engineer
aaron@zellic.io

Daniel Lu
Engineer
daniel@zellic.io

Nan Wang
Engineer
nan@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 16, 2024	Start of primary review period
September 26, 2024	Scope change from `bf1a1233` ↗ to `32366e8d` ↗
October 8, 2024	Scope change from `32366e8d` ↗ to `79119981` ↗
October 8, 2024	End of primary review period
October 10, 2024	Kick-off call
October 16, 2024	Scope change from `79119981` ↗ to `ab92460b` ↗
October 16, 2024	Scope change from `ab92460b` ↗ to `80c51fa5` ↗
November 13, 2024	Secondary kick-off call
November 22, 2024	Start of secondary review period
November 22, 2024	Scope change from `80c51fa5` ↗ to `fcdc3b19` ↗
November 27, 2024	Scope change from `fcdc3b19` ↗ to `b4a3719b` ↗
November 28, 2024	Scope change from `b4a3719b` ↗ to `44e6ef9e` ↗
December 13, 2024	End of secondary review period
January 2, 2025	Start of final review period
January 21, 2025	Scope changed from `44e6ef9e` ↗ to `c8ee36c5` ↗
January 21, 2025	Scope changed from `c8ee36c5` ↗ to `c3f8e990` ↗
January 24, 2025	End of final review period
February 11, 2025	Scope changed from `c3f8e990` ↗ to `a1dee338` ↗
July 21, 2025	ULN scope changed from `a1dee338` ↗ to `13d74b90` ↗

3. Detailed Findings

3.1. Bypassable packet hash check

Target	channel		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `msglibSendCallback` function in the channel is called by smlManager as part of the process of sending a message. It emits the `Channel::event::PACKET_SENT` event, which is what off-chain entities listen for to know that a message has been sent.

Execution must leave the channel from the first `channelSend` call because a call must be made to the messaging library to create a quote for the message, so `msglibSendCallback` must be able to verify that the packet being passed in matches what is expected (from the `channelSend` call).

To support this verification, the packet hash is assigned a `requestId` and stored in a DeterministicInsertionCircularQueue (DICQ) data structure in the contract state. The following code in `msglibSendCallback` then verifies that the `requestId` exists in the data structure and that the inputted packet matches the stored hash:

```
int requestId = $lzSend.cl::get<uint64>(md::LzSend::sendRequestId);

;; Read the requestId from the sendRequestQueue to ensure this send request is
    genuine
;; and is not being double-executed
(_, cell contents, _, int exists) =
    DeterministicInsertionCircularQueue::get(sendRequestQueue, requestId);
if (exists) {
    if (_readSendRequestQueueEntry(contents) == $lzSend.cl::hash()) {
        $storage = $storage.cl::set(
            Channel::sendRequestQueue,
            DeterministicInsertionCircularQueue::delete(sendRequestQueue,
    requestId)
        );
    }
} else {
    ;; if the send request doesn't exist, there are two cases
    ;; 1. a legitimate request was frontrun by a force-abort
    ;;  in this case, we can safely refund all the funds to the origin
    ;; 2. a malicious MITM attack by ULN
    ;;  in this case, we can't refund the funds, but we can still emit an event
    ;; This technically silently reverts, by not processing any output actions,
    ;; thus providing a refund, instead of hard reverting
    return actions;
}
```

However, the inner if statement highlighted above is missing an else branch that returns — meaning that, provided the `requestId` is valid (i.e., exists in the DICQ), code execution will continue past this check without the function returning.

Impact

Given that at least one message exists in the DICQ (i.e., is in the process of being sent), an attacker can forcefully send any number of arbitrary messages on the channel.

Note that an attacker may be able to force this condition by providing insufficient gas for the whole message-sending step to complete. The `requestId` would not be cleared while exploiting the vulnerability, so the attacker can continue to send arbitrary messages until someone intervenes.

Recommendations

Add an else condition to return early.

```
(_, cell contents, _, int exists) =
    DeterministicInsertionCircularQueue::get(sendRequestQueue, requestId);
if (exists) {
    if (_readSendRequestQueueEntry(contents) == $lzSend.cl::hash()) {
        $storage = $storage.cl::set(
            Channel::sendRequestQueue,
            DeterministicInsertionCircularQueue::delete(sendRequestQueue,
    requestId)
        );
    } else {
        return actions;
    }
} else {
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit b4a3719b ↗.

3.2. Wrong permissions on `ulnConnectionCommitPacket`

Target	ulnConnection		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

When the ULN wants to commit the packet, it sends a message to the ulnConnection to call the `ulnConnectionCommitPacket` function, which determines whether the packet should be committed or not given the OApp's configuration.

The default configuration — which can be overridden by the OApp configuration — is passed in an object (cell) input. Therefore, if the OApp does not override the default configuration, the criteria for committing the packet is determined solely using the function inputs.

Considering the endpoint and channel rely on the ulnConnection to determine the validity of the packet, it should be clear that the `ulnConnectionCommitPacket` being permissionless is problematic:

```
() _checkPermissions(int op, cell $md) impure inline {
    if (
        (op == MsglibConnection::OP::MSGLIB_CONNECTION_SEND)
        | (op ==
    MsglibConnection::OP::MSGLIB_CONNECTION_COMMIT_PACKET_CALLBACK)
    ) {
        return assertChannel($md);
    } elseif (op == UlnConnection::OP::ULN_CONNECTION_VERIFY) {
        return assertUln();
    } elseif (
        (op == UlnConnection::OP::ULN_CONNECTION_COMMIT_PACKET)
        | (op == MsglibConnection::OP::MSGLIB_CONNECTION_QUOTE)
    ) {
        return ();
    } elseif (op == UlnConnection::OP::GARBAGE_COLLECT_INVALID_ATTESTATIONS){
        ;; [...]
    }
}
```

Impact

If an OApp leaves the required/optional DVN configuration as default, an attacker can use the `UlnConnection::OP::ULN_CONNECTION_COMMIT_PACKET` operation to call `ulnConnectionCommitPacket` and thereby commit arbitrary packets.

Recommendations

Change the permissions such that only the ULN has the ability to use the `UlnConnection::OP::ULN_CONNECTION_COMMIT_PACKET` opcode.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 6c5898b2 ↗.

3.3. Broken authentication in `endpointCommitPacket`

Target	endpoint		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `endpointCommitPacket` function is permissionlessly callable with an arbitrary packet. It is intended to be called by the msglib, and it passes the packet to `channelCommitPacket`, which is permissioned.

When making that call, it passes an address taken from user input (the `MdAddress`) as the `forwardingAddress`:

```
tuple endpointCommitPacket(cell $mdAddress) impure inline method_id {
    ;; [...]
    actions~pushAction<call>(
        channelAddress,
        Channel::OP::CHANNEL_COMMIT_PACKET,
        md::ExtendedMd::New(
            $packet,
            lz::ReceiveEpConfig::New(
                receiveMsglibConnection,
                timeoutReceiveMsglibConnection,
                $storage.cl::get<uint64>(Endpoint::defaultExpiry)
            ),
            $mdAddress.cl::get<address>(md::MdAddress::address)
        )
    );
}
```

The `channelCommitPacket` function checks the `forwardingAddress` stored in `MdExtended` to ensure the original caller is the msglib:

```
tuple channelCommitPacket(cell $mdExtended) impure inline method_id {
    ;; [ ...]
    int callerMsglibConnectionAddress = $mdExtended
        .cl::get<address>(md::ExtendedMd::forwardingAddress);
    if (receiveMsglibConnection != callerMsglibConnectionAddress) {
        ;; grossly inefficient, but this will (almost) never happen
        ;; so we can optimize the happy path by isolating this logic into this
    block
        cell $defaultConfig = $mdExtended.cl::get<objRef>(md::MdObj::obj);
        int timeoutReceiveMsglibConnection = useDefaults
            ? $defaultConfig

    .cl::get<address>(lz::ReceiveEpConfig::timeoutReceiveMsglibConnectionAddress)
            : $epCon-
    figOApp.cl::get<address>(lz::EpConfig::timeoutReceiveMsglibConnection);

        int expiry = useDefaults
            ? $defaultConfig.cl::get<uint64>(lz::ReceiveEpConfig::expiry)
            :
    $epConfigOApp.cl::get<uint64>(lz::EpConfig::timeoutReceiveMsglibExpiry);

        if ((timeoutReceiveMsglibConnection != callerMsglibConnectionAddress)
            | (expiry < now())) {
            throw(Channel::ERROR::onlyApprovedReceiveMsglib);
        }
    }
    ;; [...]
}
```

That is, the address used for msglib authentication is taken from user input.

Impact

An attacker can force an endpoint to send any arbitrary packet by calling `endpointCommitPacket` with an object containing an address that passes the msglib address check.

Recommendations

Change the `forwardingAddress` to the caller's address.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `5f4193ac` ↗.

3.4. Possible runtime crash when setting hash lookups

Target	ulnConnection		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When updating hash lookups in `UlnConnection::utils::setHashLookup`, if the requested entry does not exist, a `null` value is returned instead of an `empty_cell()`. This leads to a runtime crash when attempting further operations on `null`.

Specifically, in the following code, if `hashLookups` is not `empty_cell()`, it calls `dict.udict_get_ref` to find the corresponding key. If the key is not found, `udict_get_ref` returns `null` instead of `empty_cell()`. This `null` is then assigned to `addressLookup`.

```
(cell, int) cl::dict256::get<cellRef>(cell dict, int key) inline method_id {
    if (dict.cell_is_empty()) {
        return (empty_cell(), false);
    }
    return dict.udict_get_ref?(DICT256_KEYLEN, key); ;; [0] returns null if
        key not found
}
```

When `addressLookup` is subsequently used in `setRef`, the contract crashes because FunC does not support operations on `null` values.

```
cell UlnConnection::utils::setHashLookup(
    cell $self,
    int nonce,
    int dvnAddress,
    cell $attestation
) impure inline {
    cell hashLookups = $self.cl::get<dict256>(UlnConnection::hashLookups);
    (cell addressLookup, _) = hashLookups.cl::dict256::get<cellRef>(nonce);

    ;; insert the attestation
    addressLookup = addressLookup.cl::dict256::setRef(
        dvnAddress,
        $attestation
    );
    ;; [...]
}

;; [...]
cell cl::dict256::setRef(cell dict, int key, cell val) inline method_id {
    if (dict.cell_is_empty()) {
        return new_dict().udict_set_ref(
            DICT256_KEYLEN,
            key,
            val.cast_to_cell()
        );
    }
    return dict.udict_set_ref(DICT256_KEYLEN, key, val.cast_to_cell());
}
```

Impact

This issue can cause critical runtime failures during execution of `ulnConnectionVerify`, disrupting the message-verification flow and negatively affecting the reliability of cross-chain operations.

Although it does not directly enable asset theft, the disruption of core protocol functions justifies a high impact rating.

Recommendations

Before using `addressLookup` in `setRef`, explicitly check if it is `null`.

Alternatively, modify the retrieval logic so that a missing key returns `empty_cell()` rather than `null`, preventing runtime crashes.

Remediation

LayerZero Labs independently discovered and fixed this issue in commit `3d0d11f7` ↗ during the audit period.

3.5. Risks of malformed object construction

Target	funC++		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The following technical description walks through how funC++ objects may be malformed, as further described in section 4.4. ↗. Understanding specifically how objects can be malformed is important for understanding the exploit scenario we document next.

Objects in funC++

An object cell contains a header and data, which take up the entirety of the cell:

```
(
    uint name                          [NAME_WIDTH]          ^ [HEADER_WIDTH]
    * (                                                      |
        uint field_type                [_FIELD_TYPE_WIDTH]   |
        (                                                    |
            | (                                              |
                uint cell_index        [_CELL_ID_WIDTH]      |
                uint data_offset       [_DATA_OFFSET_WIDTH]  |
                uint <3>               [_REF_OFFSET_WIDTH]   |
            )                                                |
            | (                                              |
                uint cell_index        [_CELL_ID_WIDTH]      |
                uint <MAX_CELL_BITS>   [_DATA_OFFSET_WIDTH]  |
                uint ref_offset        [_REF_OFFSET_WIDTH]   |
            )                                                |
        )                                                    |
    )                                                        |
    remaining_data                     (remaining size)      v
)
```

The cell (`remaining_data`) and cell's referenced children cells (up to four) contain the data. The index and offset describe which of those cells holds the data and how far into the cell that data is.[2]

Example funC++ object: `md::LzSend`

An example of how objects are created is in classes/msgdata/LzSend.fc's function, `md::LzSend::New`:

```
cell md::LzSend::New(
    int nativeFee,
    int zroFee,
    cell $extraOptions,
    cell $enforcedOptions,
    cell $packet,
    cell callbackData
) inline method_id {
    return cl::declare(
        md::LzSend::NAME,
        unsafeTuple([
            [cl::t::uint64, 0],                  ;; md::LzSend::sendRequestId
            [cl::t::address, NULLADDRESS],       ;; md::lzSend::sendMsglib
            [cl::t::address, NULLADDRESS],       ;;
    md::lzSend::sendMsglibConnection
            [cl::t::objRef, $packet],            ;; md::LzSend::packet
            [cl::t::coins, nativeFee],           ;; md::LzSend::nativeFee
            [cl::t::coins, zroFee],              ;; md::LzSend::zroFee
            [cl::t::objRef, $extraOptions],      ;; md::LzSend::extraOptions
            [cl::t::objRef, $enforcedOptions],   ;; md::LzSend::enforcedOptions
            [cl::t::objRef, callbackData]        ;; md::LzSend::callbackData
        ])
    );
}
```

The types and values are passed into the helper function `cl::declare`, which creates the objects using a standard, deterministic structure. Given the same inputs, the returned cell will always be the same; that is, the layout of the fields will always be consistent.

Malformed `md::LzSend` object construction

Risk arises because the internal layout of the fields in objects does not have to match the standard structure.

[2] To assist in understanding funC++, we recommend reading the funC++/classlib.fc file — especially the `cl::get<uint>` function.

Continuing with the `md::LzSend` example, consider the following function:

```
const md::lzSend::requestInfoWidth = _HEADER_WIDTH + 64 + 256 + 256;
;; low-level hacking to set these 3 fields in LzSend
;; saves about 7000 gas
cell md::lzSend::fillRequestInfo(
    cell $lzSend,
    int requestId,
    int sendMsglib,
    int sendMsglibConnection
) inline method_id {
    slice lzSendSlice = $lzSend.begin_parse();
    return begin_cell()
        .store_slice(lzSendSlice.scutfirst(_HEADER_WIDTH, 0))
        .store_uint64(requestId)
        .store_uint256(sendMsglib)
        .store_uint256(sendMsglibConnection)
        .store_slice(lzSendSlice.sskipfirst(md::lzSend::requestInfoWidth, 0))
        .end_cell();
}
```

There is a problem here: the function expects that the object was constructed using the `cl::declare` function and that the field layout is consistent with the standard structure.

That function is called from `channelSend` to fill in `sendMsglib` and `sendMsglibConnection` (the details about which msglib needs to be used):

```
;; Resolve the desired send msglib and send msglib connection
cell $epConfigOApp = $storage.cl::get<objRef>(Channel::epConfigOApp);
int sendMsglib = $epConfigOApp.cl::get<address>(lz::EpConfig::sendMsglib);
int sendMsglibConnection =
    $epConfigOApp.cl::get<address>(lz::EpConfig::sendMsglibConnection);
cell $sendPath = $storage.cl::get<objRef>(Channel::path);

;; [...]

;; Each send request is assigned a unique request ID, which is also used as the
    key into
;; the sendRequestQueue
int curRequestId = $storage.cl::get<uint64>(Channel::lastSendRequestId) + 1;

$lzSend = md::lzSend::fillRequestInfo($lzSend, curRequestId, sendMsglib,
    sendMsglibConnection);
```

These values are later used in the send-messaging step for caller authentication in `msglibSendCallback`:

```
() _checkPermissions(int op, cell $md) impure inline {
    ;; [...]
    } elseif (op == Channel::OP::MSGLIB_SEND_CALLBACK) {
        return _assertSendMsglib($md);
    } elseif (
    ;; [...]
}
```

The `_assertSendMsglib` function is as follows:

```
() _assertSendMsglib(cell $mdAddress) impure inline {
    cell epConfigOApp =
    getContractStorage().cl::get<objRef>(Channel::epConfigOApp);

    ;; Resolve the actual sendMsglib address at the time of request.
    ;; This function assumes the messagelib is not malicious or
    man-in-the-middle attacking,
    ;; as those cases are asserted in the handler itself.
    int sendMsglibAddress = $mdAddress
            .cl::get<objRef>(md::MdAddress::md)
            .cl::get<objRef>(md::MsglibSendCallback::lzSend)
            .cl::get<address>(md::LzSend::sendMsglib);

    throw_unless(Channel::ERROR::onlyApprovedSendMsglib, getCaller() ==
    sendMsglibAddress);
}
```

Note that the `sendMsglibAddress` variable comes from `mdAddress`, which is user-controllable (the `msglibSendCallback` function).

However, if the attacker simply changes the `sendMsglib` address to be a valid msglib address, the `getCaller()` assertion would fail. Likewise, if the attacker changes the address to be their address, the DICQ hash check in `msglibSendCallback` would fail:

```
;; Read the requestId from the sendRequestQueue to ensure this send request is
    genuine
;; and is not being double-executed
(_, cell contents, _, int exists) = DeterministicInsertionCircularQueue::get(
    sendRequestQueue,
    requestId
);

if (exists) {
    if (_readSendRequestQueueEntry(contents) == $lzSend.cl::hash()) {
        $storage = $storage.cl::set(
            Channel::sendRequestQueue,
            DeterministicInsertionCircularQueue::delete(sendRequestQueue,
    requestId)
        );
    } else {
        ;; See below comment, this else case is logically the same as the below
    else block,
        ;; but needs to be split due to lack of short-circuiting boolean
    expressions in funC
        return actions;
    }
} else {
```

Previously, in the `channelSend` step, the `LzSend` object's hash was stored to this data structure. Before being stored, the value is overwritten using `md::lzSend::fillRequestInfo`. So, normally, the `sendMsglib` address must be the same as the real msglib address picked in the `channelSend` step.

If the user has the ability to control the `LzSend` passed into the `channelSend` function, they can carefully craft the object in a way such that the msglib fields are not overwritten when `md::lzSend::fillRequestInfo` is called, so that they can act as the msglib. This means they can then call the `msglibSendCallback` function with arbitrary values (and therefore an arbitrary packet).

Of course, this object is passed in by the OApp. But there are two scenarios in which this is possible:

1. The OApp is designed in a risky way, where it simply performs assertions on a user-inputted `LzSend` object (e.g., that the message data matches what is expected) and then sends it. It is possible for an OApp to be designed this way, and it is not immediately obvious why this design is risky, so we consider this to be a reasonable exploit scenario. However, there is a more likely exploit scenario that is more dangerous.

2. A maliciously created OApp intentionally sends a crafted `LzSend` object to the `channelSend` function. In this scenario, the OApp has the ability to act as a msglib itself.

Impact

Depending on how the off-chain infrastructure operates (i.e., what checks it has), a malicious OApp may be able to send packets without paying fees, or potentially airdrop native coin to the destination chain without paying. In this situation, the executor is the biggest losing party as they are unable to collect fees (while paying the highest amount of gas or native coin).

Another exploit scenario is a vulnerable OApp that unintentionally allows an attacker to act as a msglib and send arbitrary packets.

Recommendations

A write invalidates all writes, reads, and therefore checks. Ensure that user-inputted objects are only ever read from.

For example, it is safe to create a new object and simply copy values out of the user-inputted object ("sanitizing"). Likewise, it is safe to create a new object containing updated fields but have the user-inputted object be a child cell that is only read from when necessary ("wrapping").

Remediation

This issue has been acknowledged by LayerZero Labs.

3.6. Improper eviction from send queue clogs channel

Target	channel		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Note: This issue was discovered early in the audit timeline. The data structures discussed in this finding were subsequently removed from the codebase.

Description

The protocol has a channel contract for every path which stores packet nonces and data. A key piece of channel state is the send queue, which tracks the state of packets.

Due to storage constraints, the send queue can only hold the state of a bounded number of packets. LayerZero Labs implemented this logic with their CompareAndSwapMap. This data structure holds the state of a nonce n under the key mod$(n, 1023)$. It offers `get` and `set` functions which get and set a nonce's state, respectively. In each case, it is verified that the currently stored state corresponds to the provided nonce, rather than a different one whose key would collide.

But when the contract attempts to evict a nonce (after a packet is aborted or marked as sent), it merely sets the nonce's state to the default value. This means that the nonce is not actually removed from the send queue, so future attempts to add nonces with colliding keys will fail.

Impact

This means that after the send queue is filled, no new packets can be sent on the channel. It effectively becomes blocked.

Recommendations

We recommend implementing logic for removing nonces from the send queue.

Remediation

LayerZero Labs acknowledged this issue before removing the data structure from the codebase.

3.7. Unsafe options version check in OApp

Target	baseOApp		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `_lzSend` function in baseOApp calls `_assertOptionsValid`, passing the extra (user-specified) and enforced (OApp-required) options as arguments.

```
() _assertOptionsValid(cell $extraOptions, cell $enforcedOptions)
    impure inline {
    if (
        (~ $extraOptions.cl::isNull())
        & (~ $enforcedOptions.cl::isNull())
    ) {
        ;; extra == 'something', enforced == 'something'
        throw_unless(ERROR::InvalidExtraOptionsVersion, $extraOptions.cl::
            equalObjTypeShallow($enforcedOptions));
```

That function then passes the arguments into the following `equalObjTypeShallow` function to require that the structure of the extra options matches the enforced options.

```
int cl::equalObjTypeShallow(cell $a, cell $b) {
    slice aSlice = $a.begin_parse();
    slice bSlice = $b.begin_parse();
    int aRefs = aSlice.slice_refs();

    if (
        (aRefs != bSlice.slice_refs())
        | (aSlice.slice_bits() != bSlice.slice_bits())
    ) {
        return false;
    }

    int refIndex = 2;
    while (refIndex < aRefs) {
        if (
            (aSlice.preload_ref_at(refIndex).begin_parse().slice_refs() !=
    bSlice.preload_ref_at(refIndex).begin_parse().slice_refs())
            | (aSlice.preload_ref_at(refIndex).begin_parse().slice_bits() !=
    bSlice.preload_ref_at(refIndex).begin_parse().slice_bits())
        ) {
            return false;
        }
        refIndex += 1;
    }

    return true;
}
```

This function performs the following checks:

- The number of children cells of each object must be equal.
- The number of bits in each object must be equal.
- The second and subsequent children cells of each object must have the same number of children cells and bits.

There are two problems with this comparison. The first is that it is not a deep comparison; that is, it does not check more than one level deep. It is possible that the extra options contains a large, deeply nested, garbage cell structure.

But also, it is possible to bypass the version assertion because there is no guarantee about the layout of fields in the extra options.

Impact

If the enforced options were OptionsV1, an attacker could create a cell with the same number of children cells and bits as the enforced options, but with pointers set to have fields overlap — allowing the attacker to pass an OptionsV2 in extra options.

Recommendations

Consider directly checking the options object type, then sanitizing the options (i.e., copying to a new object).

Remediation

This issue has been acknowledged by LayerZero Labs.

3.8. Unhandled nonce state after `nilify`

Target	channel		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

After a `nilify` operation, a nonce may end up in a state where the commit pipeline considers it committed while the execution queue regards it as uncommitted, effectively `(committed, uncommitted)`.

The `channelCommitPacket` function is designed to handle only two clear transitions:

1. `(uncommitted, uncommitted) → (committed, committed)`
2. `(committed, committed) → (committed, committed)`

However, the `(committed, uncommitted)` state introduced by `nilify` does not fit either path. When `channelCommitPacket` attempts to reverify such a nonce, it sees a committed pipeline state and tries to treat the nonce as `(committed, committed)`, updating it from committed to committed. Yet, because the execution queue is still uncommitted, the operation fails. As a result, the code cannot properly recover or reverify the nonce after `nilify`, leaving the system stuck in an inconsistent or unclear state.

```
_assertNonceCommittable(incomingNonce);
int expectedCurrentState = ExecutionQueue::committed;
if (incomingNonce >= $commitPOOO.cl::get<uint64>(POOO::f::nextEmpty)) {
    expectedCurrentState = POOO::isBitSet($commitPOOO, incomingNonce)
        ? ExecutionQueue::committed
        : ExecutionQueue::uncommitted;
}

(cell executionQueue, _, _) = CompareAndSwapMap::set(
    $storage.cl::get<cellRef>(Channel::f::executionQueue),
    incomingNonce,
    $packet,
    expectedCurrentState,
    ExecutionQueue::committed
);
```

Impact

The `channelCommitPacket` function will be handle the state caused by `nilify`. This situation can lead to partial denial-of-service (DOS) conditions, where messages cannot be properly finalized, leaving the system in an inconsistent operational state.

Recommendations

Add a preliminary state check before performing the commit action, and if the check passes, update the execution queue directly without relying on conditional map operations.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `36471c45` ↗.

3.9. Ineffective short circuit in `_optimizedNonceCommittable`

Target	channel		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the function optimizedNonceCommittable, the short circuit condition never triggers before the first commit because the `executionQueue` is only updated during commit and execution phases, where entries are set to `ExecutionQueue::committed` or `ExecutionQueue::executing`.

Before that, each entry remains `(unoccupied, status=0, key=0, ref=empty)`, so any getter call on the executionQueue returns `(DeterministicInsertionCircularQueue::invalidKey, empty_cell(), DeterministicInsertionCircularQueue::invalidStatus, exists)`, i.e. `(-1, empty_cell(), -1, 0)`.

Consequently, `(actualKey == incomingNonce)` is never satisfied prior to the initial commit, making the "short-circuit for efficiency in the common case" ineffective.

```
;; returns boolean committable
int _optimizedNonceCommittable(cell $executePOOO, cell executionQueue,
    int incomingNonce) impure inline {
    throw_if(Channel::ERROR::invalidNonce, incomingNonce <= 0);

    int firstUnexecutedNonce = $executePOOO.POOO::getNextEmpty();

    (int actualKey, _, int status, int exists) =
    DeterministicInsertionCircularQueue::get(
        executionQueue,
        incomingNonce
    );

    if (
        (incomingNonce == firstUnexecutedNonce)
        & (actualKey == incomingNonce)
        & (status != ExecutionQueue::executing)
    ) {
        ;; short-circuit for efficiency in the common case
        return true;
    }
    [...]
}
```

Impact

This implementation results in missed early validations and performance inefficiencies, as the function always reverts to more exhaustive checks. By the time `(actualKey == incomingNonce)` can match, the benefit of bypassing full verification is effectively lost, negating the intended optimization.

Recommendations

It is advisable to adjust the logic so that the short-circuit condition can be valid even before the queue undergoes its first commit, or remove the short-circuit check.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.10. Lack of quorum and verifier count validation

Target	workers		
Category	Protocol Risks	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `setQuorum` and `setVerifiers` functions of the DVN worker contract (workers/dvn/handler.fc), there are no checks to ensure that the new quorum does not exceed the number of verifiers or that the number of verifiers remains sufficient after updating them. In other words, the code does not assert that there are enough verifiers to meet the newly specified quorum, nor does it prevent setting a smaller verifier dictionary than the current quorum requires.

By contrast, the EVM implementation of the DVN (MultiSig.sol) includes these checks. For example, when setting a new quorum, it reverts if the number of signers is less than the quorum (see here ↗).

Similarly, when adding or removing signers, the EVM implementation ensures that the resulting signer set is never smaller than the current quorum (see here ↗).

Impact

This issue does not present a direct security risk. However, it allows users to create inconvenient or malformed configurations that deviate from the intended design.

Recommendations

Make the following changes:

- Add a check in `setQuorum` to ensure the quorum does not exceed the number of current verifiers.
- Update `setVerifiers` to forbid reducing the verifier dictionary below the required quorum count.

This aligns the TON DVN worker's configuration logic with the EVM implementation and prevents malformed configurations that cannot meet the intended quorum requirements.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.11. Confusing else branch in `compareObjectFields`

Target	funC++		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `compareObjectFields` function, the purpose of the else branch is to detect the end of the object fields, rather than to determine whether the object is malformed. However, the current implementation can easily be misunderstood as marking an error field in the else branch, which is not correct.

In fact, when the else branch is reached, it indicates that the object fields have reached their end. Modifying it to directly return `-1` aligns better with the design intent, clearly indicating that all fields have been matched and processed. This change also makes the code logic more intuitive and concise, while avoiding any misinterpretation of the branch's purpose.

```
;; returns -1 (true) if equal, otherwise the index of the first field that
    differs
;; returns 16 if the types of the objects are not equal
int compareObjectFields(cell $lhs, cell $rhs) impure inline {
    int malformed = cl::typeof($lhs) != cl::typeof($rhs);
    if (malformed) {
        return INVALID_CLASS_MEMBER;
    }
    if (cl::typeof($lhs) == cl::NULL_CLASS_NAME) {
        return -1;
    }
    int fieldIndex = 0;
    while (fieldIndex < INVALID_CLASS_MEMBER) {
        int curFieldType = $lhs.typeofField(fieldIndex);
        if (curFieldType == cl::t::cellRef) {
            malformed = $lhs.cl::get<objRef>(fieldIndex).cl::hash() !=
    $rhs.cl::get<objRef>(fieldIndex).cl::hash();
            if (malformed) {
                ~dump($lhs.cl::get<objRef>(fieldIndex).cell_hash());
                ~dump($rhs.cl::get<objRef>(fieldIndex).cell_hash());
            }
        } elseif (curFieldType <= cl::t::uint256) {
            int cur_field_width = _getTypeWidth(curFieldType);
            malformed = $lhs.cl::get<uint>(fieldIndex, cur_field_width) !=
    $rhs.cl::get<uint>(fieldIndex, cur_field_width);
            if (malformed) {
                str::console::log<int>("lhs:  ", $lhs.cl::get<uint>(fieldIndex,
    cur_field_width));
                str::console::log<int>("rhs:  ", $rhs.cl::get<uint>(fieldIndex,
    cur_field_width));
            }
        } else {
            fieldIndex = INVALID_CLASS_MEMBER;
        }
        if (malformed) {
            ~strdump("Malformed field");
            ~dump(fieldIndex);
            return fieldIndex;
        }
        fieldIndex += 1;
    }
    return -1;
}
```

Impact

This issue does not pose a security risk as it only appears in test code, having no effect on production logic or security guarantees.

Recommendations

Change the else branch to return -1 directly, clarifying that the object fields have been fully matched:

```
;; [...]
} else {
    fieldIndex = INVALID_CLASS_MEMBER;
    return -1;
}
```

Alternatively, `curFieldType` could be relied on to differentiate these cases. If `curFieldType` exceeds `cl::t::uint256` and is not `cl::t::cellRef`, consider it an unexpected error type and return `INVALID_CLASS_MEMBER`; if `curFieldType` is `0xF`, return `-1` to indicate that all fields have been successfully processed.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.12. Missing reinitialization safeguard in baseHandler/baseStorage framework

Target	endpoint		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The contract framework provided by baseHandler and baseStorage includes an initialized flag to prevent interactions with completely uninitialized contracts. However, it lacks an explicit mechanism to prevent reinitialization of contracts that have already undergone a successful initialization.

Without a proper check, any contract leveraging this framework risks being bricked by a second, unintended invocation of its initialization logic. This can lead to a self-inflicted denial of service, where the contract's state is reset or placed into an invalid configuration.

```
;; Step 2: initialize
tuple initialize(cell $md) impure inline {
    (cell $storage, tuple actions) = _initialize($md);

    int baseStorageIndex = _getBaseStorageIndex();
    setContractStorage(
        $storage
            .cl::set(
                baseStorageIndex,
                $storage
                    .cl::get<objRef>(baseStorageIndex)
                    .cl::set(BaseStorage::f::initialized, true)
            )
    );

    return actions;
}
```

Impact

If a contract is reinitialized after its first successful initialization, it may enter an undefined or nonfunctional state, effectively halting normal operations and requiring manual intervention.

Recommendations

Before executing any initialization actions, verify that the contract is not already marked as initialized.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `b09501a6` ↗.

3.13. Unused opcodes without permission checks

Target	uln, OApp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Within both the ULN and OApp opcode handling logic, there are opcodes that appear to be unused or unimplemented and are not included in their corresponding permission checks.

`Uln::OP::ULN_QUOTE`

This opcode seems referenced in `_executeOpcode` but not actually used, with no `_checkPermissions` entry. As dead code, it introduces confusion about intended functionality, and if implemented later without proper permissions, it could lead to access-control oversights.

```
tuple _executeOpcode(int op, cell $md) impure {
;; [...]
    } elseif (op == Uln::OP::ULN_QUOTE) {
        return ulnQuote($md);
    }
;; [...]
;;; ================PERMISSION FUNCTIONS=====================

() _checkPermissions(int op, cell $md) impure inline {
    if (op == Uln::OP::ULN_SEND) {
        return assertConnection($md);
    } elseif (
        (op == Uln::OP::ULN_VERIFY)
        | (op == Uln::OP::ULN_COMMIT_PACKET)
    ) {
        return ();
    } elseif (
        (op == Uln::OP::DEREGISTER_WORKER_FEELIB)
        | (op == Uln::OP::COLLECT_WORKER_RENT)
        | (op == Uln::OP::SET_WORKER_FEELIB_STORAGE)
        | (op == Uln::OP::REFILL_WORKER_RENT)
        | (op == Uln::OP::GC_ATTESTATIONS)
    ) {
        return ();
    } elseif (
        (op == Uln::OP::SET_DEFAULT_ULN_RECEIVE_CONFIG)
        | (op == Uln::OP::SET_DEFAULT_ULN_SEND_CONFIG)
        | (op == Uln::OP::UPDATE_WORKER_FEELIB)
        | (op == Uln::OP::SET_TREASURY_FEE_BPS)
    ) {
        return assertOwner();
    } else {
        ;; we must put a check for all opcodes to make sure we don't
        ;; mistakenly miss an opp code's permissions
        throw(BaseInterface::ERROR::invalidOpcode);
    }
}
```

`Layerzero::OP::BURN_CALLBACK`

Currently implemented as a no-operation function in the base OApp, it is also missing from `_checkPermissions`. While not a security issue as of the time of writing (as it does nothing), if later expanded to perform meaningful actions, the lack of a permission check could become problematic.

`Dvn::OP::SET_PROXY_ADMINS`

The opcode is uncallable because it is missing from its `_checkPermissions` in the dvn worker.

Impact

The absence of permission checks for these unused or no-operation opcodes is not a direct security risk today. However, retaining such code can cause maintenance difficulties and confusion and may lead to permission-related vulnerabilities if these opcodes are repurposed or implemented in the future.

Recommendations

Remove or comment out unused opcodes (`ULN_QUOTE` and `BURN_CALLBACK`) if they serve no current purpose.

Alternatively, if these opcodes are planned for future use, add corresponding `_checkPermissions` entries to maintain consistent and secure access control.

Consider adding a permission to the `_checkPermissions` for `Dvn::OP::SET_PROXY_ADMINS`.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.14. Incorrect field parsing in `parseGasLimitsPrices`

Target			
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The gas prices structure is parsed as the following:

```
;; gas_prices_ext#de gas_price:uint64 gas_limit:uint64
    special_gas_limit:uint64 gas_credit:uint64
;; block_gas_limit:uint64 freeze_due_limit:uint64 delete_due_limit:uint64
;; = GasLimitsPrices;
;;
;; gas_flat_pfx#d1 flat_gas_limit:uint64 flat_gas_price:uint64
    other:GasLimitsPrices
;; = GasLimitsPrices;
```

The expected structure should be `gas_flat_pfx#d1` with the `other` field being `gas_prices_ext#de`.

The validation check is correct, in that it properly expects `#d1` at the beginning and `#de` at offset 136 bits:

```
if (
    (cfgSlice.preload_uint(8) != 0xd1)
    | (cfgSlice.preload_bits_offset(136, 8).preload_uint(8) != 0xde)
    // Correctly expects #de at offset 136
) {
    return (-1, -1, -1, -1, -1, -1, -1, -1, -1);
}
```

However, in the following code, the parser incorrectly assumes there's a `specialGasLimit` field immediately after the `#d1` tag.

```
cfgSlice~load_uint8(); // #d1 - correct
int specialGasLimit = cfgSlice~load_uint64(); ;; incorrect: this field doesn't
    exist in gas_flat_pfx#d1
int flatGasLimit = cfgSlice~load_uint64();    ;; incorrect: should be first
    field after #d1
```

```
int flatGasPrice = cfgSlice~load_uint64();    ;; incorrect: should be second
    field after #d1
cfgSlice~load_uint8(); ;; incorrect: should be #de, but we're already
    misaligned
;; ... rest of parsing
```

Structure summary

Bit range	Expected field	Actual field	Correct?
0-7	`#d1`	discarded	☑
8-71	`flat_gas_limit`	`specialGasLimit`	☐
72-135	`flat_gas_price`	`flatGasLimit`	☐
136-199	`#de` (8 bits) + high 56 bits of `gas_price`	`flatGasPrice`	☐
200-207	low 8 bits of `gas_price`	discarded as if it were the `#de` tag	☐
208-271	`gas_limit`	`gasPrice`	☐
272-335	`special_gas_limit`	`gasLimit`	☐
336-399	`gas_credit`	`gasCredit`	☑
400-463	`block_gas_limit`	`blockGasLimit`	☑
464-527	`freeze_due_limit`	`freezeDueLimit`	☑
528-591	`delete_due_limit`	`deleteDueLimit`	☑

The initial incorrect read of `specialGasLimit` causes a 64-bit shift in all subsequent fields until the parser "realigns".

According to the TL-B schema, `gas_flat_pfx#d1` should have:

1. `flat_gas_limit:uint64` (first field)
2. `flat_gas_price:uint64` (second field)

3. `other:GasLimitsPrices` (which is `gas_prices_ext#de`)

By reading a non-existent `specialGasLimit` first, the parser consumes 64 bits that should belong to `flat_gas_limit`, causing all subsequent fields to be misaligned by exactly 64 bits.

Impact

The bug remained hidden because:

1. `gas_limit` and `special_gas_limit` happen to be identical on the base-chain.
2. The function is only used in `MaxCommitPacketValueAssertion` only, and that code uses the parsed `gasLimit` value only.
3. No tests exist for the `parseGasLimitsPrices` function.

This finding is classified as "Informational" because while it represents a fundamental parsing error, it's currently unexploitable in deployed contracts due to coincidental value equality between `gas_limit` and `special_gas_limit` on the base-chain, and the fact that the function is only used in one place (and it only uses the `gasLimit` value there.

However, it could break fee logic if deployed on TON forks or when gas parameter values diverge, making it a definite finding that warrants attention and correction.

Recommendations

Make the following change:

```
slice cfgSlice = cfg.begin_parse();

/* -- outer gas_flat_pfx (#d1) -- */
cfgSlice~load_uint8();                    // #d1
int flatGasLimit    = cfgSlice~load_uint64();
int flatGasPrice    = cfgSlice~load_uint64();

/* -- inner gas_prices_ext (#de) -- */
cfgSlice~load_uint8();                    // #de
int gasPrice        = cfgSlice~load_uint64();
int gasLimit        = cfgSlice~load_uint64();
int specialGasLimit = cfgSlice~load_uint64();
int gasCredit       = cfgSlice~load_uint64();
int blockGasLimit   = cfgSlice~load_uint64();
int freezeDueLimit  = cfgSlice~load_uint64();
int deleteDueLimit  = cfgSlice~load_uint64();

return (
    specialGasLimit,   // 1
    flatGasLimit,      // 2
    flatGasPrice,      // 3
    gasPrice,          // 4
    gasLimit,          // 5
    gasCredit,         // 6
    blockGasLimit,     // 7
    freezeDueLimit,    // 8
    deleteDueLimit     // 9
);
```

Remediation

LayerZero Labs independently discovered and fixed this issue in commit 13d74b90 ↗ after the audit period.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Possibility of messages being sent out of order

Note to OApp developers: The following technical note is relevant to developers who are building OApps on LayerZero TON. If your OApp has an invariant that messages must be received in the exact order they were sent, please read this note carefully.

In LayerZero V2, typically the OApp (and executor) is responsible for ensuring packets are delivered in order, by tracking the nonce on the receiving side (i.e., requiring that the nonce of the next packet be one greater than the nonce of the last packet). This is possible because, in the other implementations (such as EVM), message sending — and nonce assignment — happen atomically.

The TON implementation seems to allow race conditions on the sending side that could ultimately cause packets to be delivered out of order; it is possible that two executions of `msglibSendCallback` may have nonces assigned to their packets in a different order from when the messages were originally sent.

This possibility would break some protocols that have an invariant that all messages are received in the exact order they were sent.

However, it is difficult to prevent this in LayerZero since a packet cannot be assigned a nonce until it is guaranteed that the packet will not be aborted (using `forceAbort`). A message-sending flow may not fully reach nonce assignment for reasons such as an out-of-gas (OOG) condition.

Regarding this, LayerZero Labs noted the following:

> It is possible to enforce that invariant as long as the msglib follows a static contract call path for every message and the OApp is careful about changing the endpoint configuration.
>
> TON guarantees that two messages sent from contract A to contract B will be delivered in increasing LT order ↗, so two sends from the same OApp to the same channel to the same msglib will always be propagated in the same order.
>
> These properties are provided by ULN302, so it is possible to build such an application. For ULN302 on TON, we are not implementing workers as separate contracts but instead implementing them as bytecodes that are run in the ULN contract context.
>
> ... Such protocols should not use default EP configuration [to reduce centralization risk from LayerZero owner].

That is, as long as the OApp owner is careful to not change the endpoint configuration (such that the call path remains the same), the invariant can be partially held.

To reduce centralization risk (of the OApp and LayerZero owners) and fully prevent out-of-order execution, we recommend that OApps with this invariant implement their own nonce tracking inside the packet data and use that to reorder messages, or otherwise enforce message ordering, as necessary.

For information on message ordering on TON, please see the TON documentation ↗.

4.2. Warning regarding message sizes

Note to OApp developers: The following technical note is relevant to developers who are building OApps on LayerZero TON. If you have OApps that communicate with LayerZero TON, please review the information below carefully.

The message in a packet is a linked list, and the following `lz::Packet::assertValidReceiveMessage` function (called when sending and receiving in the channel) asserts its size:

```
const int lz::Packet::MAX_RECEIVE_MESSAGE_CELLS = 32;
const int lz::Packet::MAX_SEND_MESSAGE_CELLS = 255;

;; [...]

() lz::Packet::assertValidReceiveMessage(cell $self) impure inline {
    lz::Packet::_assertValidLinkedList(
        $self.cl::get<cellRef>(lz::Packet::message),
        lz::Packet::MAX_RECEIVE_MESSAGE_CELLS
    );
}
```

Per the following code — with the exception of the last cell of the linked list — each cell must contain `MAX_CELL_WHOLE_BYTE_BITS` bits:

```
() lz::Packet::_assertValidLinkedList(cell head, int maxLen) impure inline {
    slice messageSlice = head.begin_parse();
    repeat (maxLen) {
        (int sliceBits, int sliceRefs) = messageSlice.slice_bits_refs();
        if (sliceRefs == 0) {
            throw_if(lz::Packet::ERROR::INVALID_MESSAGE, sliceBits % 8 != 0);
            return ();
        } else {
            throw_if(
                lz::Packet::ERROR::INVALID_MESSAGE,
                (sliceRefs != 1) | (sliceBits != MAX_CELL_WHOLE_BYTE_BITS)
            );
        }
        messageSlice = messageSlice.preload_first_ref().begin_parse();
    }
    throw(lz::Packet::ERROR::INVALID_MESSAGE);
}
```

That is, the cells must be 127 bytes:

```
const MAX_CELL_BYTES = 127;
const MAX_CELL_WHOLE_BYTE_BITS = MAX_CELL_BYTES * 8;
```

Because the linked list must be 32 cells or shorter, the maximum receivable message size is

$$32 \times 127 = 4064$$

However, the maximum sendable message size in other LayerZero implementations, per the LayerZero documentation ↗, is this:

> The `maxMessageSize` depends on the Send Library. In SendUln302, the default max is 10000 bytes, but this value can be configured per OApp.

It is important to note that if another chain sends a message that is too large to TON, it will not be deliverable and therefore not executable. If it is not deliverable, then communications to the chain will halt.

To prevent potential loss of user funds, DOS, and other potential issues, enforce a maximum message size from the OApp on other chains if the message is being sent to the TON chain.

To reduce centralization risk if the default library is used, and reduce the risk of user error in case a nondefault library is configured without setting message size limit for TON, we do not recommend relying on `maxMessageSize` in the send library.

4.3. Lack of support for composed message options

Note to OApp developers: The following technical note is relevant to developers who are building OApps on LayerZero TON. Please read this note carefully.

Please note that LayerZero TON does not support composed messages. However, there is no mechanism by default in LayerZero to prevent composed options from being sent to the TON chain. This means that if a user pays for and sends a message to TON that has extra options for a composed message, the value paid for in fees will be lost to the user.

While this may be equated to users self-griefing, it may not be obvious to users that the options are not supported on TON.

We suggest that OApp developers parse the message options and reject any composed-message options before sending a message to the TON chain. Additionally, if enforced options on other chains include composed message options, ensure the TON chain does not receive these options.

4.4. Risks of flexible object structure

We warned LayerZero Labs in a few conversations that the flexible structure of funC++ objects can be risky. After investing significant time looking into different cases where they may be problematic, we identified at least one such case, which is documented in Finding 3.5. ↗.

Specifically, whenever a user can pass in an object, there is no guarantee that the fields in that object are not overlapping or at unexpected offsets in the object (if accessing the object as a byte array instead of following pointers). That is, writing invalidates all writes and reads (and therefore checks on the object).

Because of this, we recommend only reading from objects passed in as arguments and never writing to them. There are a couple of easy ways to approach this:

1. Copy all of the fields into a newly created object if needing to modify an object, and modify as necessary.
2. Wrap user-inputted objects with a new object. That is, make the user-inputted object a field of a new object, which also contains the updated fields needed.

LayerZero Labs opted to implement the first option (sanitization) in many places to reduce risk.

4.5. Handler reuse in channel

The contract structure encouraged by the baseMain library involves a `runHandler` entry point that passes the message data to different handler functions, dispatching based on the opcode. This is how nearly all contracts are currently laid out.

One observation is that the provided library functions make these handlers semi-effectful. In particular, the handlers often use the `preamble` helper to obtain the current state of contract storage and an empty tuple of actions.

```
(cell, tuple) preamble() impure inline {
    return (getContractStorage(), emptyActions());
}
```

These helpers then *mutate* the storage by modifying that object and calling `setContractStorage` themselves, while *returning* the tuple of actions they wish to perform.

A few of the channel contract handlers call other handlers. The fact that these handlers are somewhat effectful means that the control flow is sometimes obscured or harder to understand. For a simple example, consider the implementation of `nilify`.

```
tuple nilify(cell $packetId) impure inline method_id {
    (cell $storage, tuple actions) = preamble();

    cell $receivePath =
    $storage.cl::get<objRef>(Channel::path).lz::Path::optimizedReverse();
    _assertEqualPaths($receivePath,
    $packetId.cl::get<objRef>(md::PacketId::path));

    int incomingNonce = $packetId.cl::get<uint64>(md::PacketId::nonce);
    _assertNonceCommittable(incomingNonce);

    _commitFakePacket($storage, incomingNonce, $receivePath);

    setContractStorage(
        getContractStorage().cl::set(
            Channel::executionQueue,
            DeterministicInsertionCircularQueue::delete(
                $storage.cl::get<cellRef>(Channel::executionQueue),
                incomingNonce
            )
        )
    );

    actions~pushAction<event>(Channel::event::PACKET_NILIFIED, $packetId);
    actions~pushAction<call>(
        $receivePath.cl::get<uint256>(lz::Path::dstOApp),
        OApp::OP::NILIFY_CALLBACK,
        md::MdObj::new($packetId, getInitialStorage())
    );

    return actions;
}
```

While it obtains a `$storage` object from `preamble`, this object becomes stale because it calls the `_commitFakePacket` function, which then calls the `channelCommitPacket` handler. That handler itself fetches the state and modifies it, which leaves `$storage` out of date.

Thus, at the end of the function, the `cl::set` call cannot be performed on the out-of-date `$storage` object, because there are changes to storage from `_commitFakePacket` that are desired to be kept. So the contract calls `getContractStorage` to fetch that latest state. However, when it actually sets the `Channel::executionQueue` field, it performs an insertion on the stale `$storage.cl::get<cellRef>(Channel::executionQueue)` object.

In this case, the behavior may be desired because the updates to the execution queue made by the `channelCommitPacket` should be overwritten. But if handlers are meant to be reused in other handlers, the way they are structured makes reasoning about the control flow more confusing. In contrast, the actions emitted by `channelCommitPacket` are discarded.

One possible API change for clarifying this behavior would be to make handlers pure with respect to storage and actions. They could all take initial state and initial actions objects as arguments and return the new state and new actions.

4.6. Type alias error in `Channel::sendRequestQueue`

`Channel::sendRequestQueue` is currently used as a DICQ, but in the storage definition it is declared as `cl::t::dict256`. While this discrepancy does not pose immediate functional issues—since the variable is both initialized and used as a DICQ.

Although the mismatch doesn't cause immediate issues, it could lead to errors if future updates mistakenly rely on the type alias and treat the field as a dict256. To prevent this, it’s best to correct the definition so it consistently reflects its actual usage as a DICQ.

```
;; @owner manager
cell Channel::New(int owner, cell $path, int endpointAddress)
    inline method_id {
    return cl::declare(
        Channel::NAME,
        unsafeTuple([
            [cl::t::objRef, BaseStorage::New(owner)],           ;;
    Channel::baseStorage
            [cl::t::objRef, $path],                             ;; Channel::path
            [cl::t::address, endpointAddress],                   ;;
    Channel::endpointAddress
            [cl::t::objRef, lz::EpConfig::NewWithDefaults()],    ;;
    Channel::epConfigOApp
            [cl::t::uint64, 0],                                  ;;
    Channel::outboundNonce
            [cl::t::dict256, cl::dict256::New()],                ;;
    Channel::sendRequestQueue
            [cl::t::uint64, 0],                                  ;;
    Channel::sendRequestId
            [cl::t::objRef, POOO::New()],                        ;;
    Channel::commitPOOO
            [cl::t::objRef, POOO::New()],                        ;;
    Channel::executePOOO
            [cl::t::cellRef, empty_cell()],                      ;;
    Channel::executionQueue (DICQ)
            [cl::t::coins, 0]                                    ;;
    Channel::zroBalance
        ])
    );
}
```

```
;;; ================INTERFACE FUNCTIONS=====================

(cell, tuple) _initialize(cell $md) impure inline {
    (cell $storage, tuple actions) = preamble();
    cell $path = $storage.cl::get<objRef>(Channel::path);

    throw_if(
        Channel::ERROR::wrongPath,
        ($path.cl::get<uint32>(lz::Path::srcEid) == 0)
        | ($path.cl::get<address>(lz::Path::srcOApp) == NULLADDRESS)
        | ($path.cl::get<uint32>(lz::Path::dstEid) == 0)
        | ($path.cl::get<address>(lz::Path::dstOApp) == NULLADDRESS)
    );

    return (
        $storage
            .cl::set(Channel::executionQueue,
    DeterministicInsertionCircularQueue::create())
            .cl::set(Channel::sendRequestQueue,
    DeterministicInsertionCircularQueue::create()),
        actions
    );
}
```

4.7. Differences from EVM specification

Note that the TON implementation differs from the white paper, EVM specification, and Aptos specification in a couple of ways:

- In the TON implementation, neither `burn` nor `nilify` functions take the `payloadHash` as an argument (compare this to the EVM spec ↗).
- The `skip` function is not implemented.

LayerZero Labs noted that these differences were conscious decisions.

4.8. Disclaimer regarding send-queue congestion

If the send queue becomes congested (e.g., if a msglib gets bricked) for any reason, users will be unable to send further messages unless the OApp manually `forceAborts` the stuck messages or otherwise fixes the congestion.

This is the intended design.

4.9. Gas optimizations

Below are a few small gas optimizations we made note of during the audit.

Unnecessary bitwise `OR` operations

Note that these two bitwise `OR` operations in `_viewExecutionStatus` are unnecessary, as the conditions to reach them require that the values are `0`.

```
int executed = incomingNonce < firstUnexecutedNonce;
if ((~ executed) & (incomingNonce < (firstUnexecutedNonce + MAX_CELL_BITS))) {
    executed = executed | $executePOOO.POOO::isBitSet(incomingNonce);
}

int committed = incomingNonce < firstUncommittedNonce;
```

```
if ((~ committed) & (incomingNonce < (firstUncommittedNonce + MAX_CELL_BITS)))
    {
    committed = committed | $commitPOOO.POOO::isBitSet(incomingNonce);
}
```

Unnecessary call to `_assertNonceCommittable`

In the `nilify` function, consider simply asserting the `isCommittable` (from the first value in the tuple) return value from `_nonceCommittable`:

```
_assertNonceCommittable(incomingNonce);

(_, cell $previousPacket) = _nonceCommittable(incomingNonce);
```

Unnecessary call to `_viewInboundNonce`

In `_viewExecutionStatus` in protocol/channel/handler.fc, consider replacing the call to `_viewInboundNonce()` with `firstUncommittedNonce - 1`:

```
int _viewExecutionStatus(int incomingNonce) impure method_id {
    cell $storage = getContractStorage();

    cell $executePOOO = $storage.cl::get<objRef>(Channel::f::executePOOO);
    int firstUnexecutedNonce =
    $executePOOO.cl::get<uint64>(POOO::f::nextEmpty);
    cell $commitPOOO = $storage.cl::get<objRef>(Channel::f::commitPOOO);
    int firstUncommittedNonce = $commitPOOO.cl::get<uint64>(POOO::f::
        nextEmpty);
    int inboundNonce = _viewInboundNonce();
```

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Risk of collusion

In LayerZero V1, the soundness of the protocol depended on two independent actors (the relayer and oracle) providing correct information to the receiver chain. If only one entity provided false, malicious information, the UA deployer could change the entity before the protocol would be harmed. But if both entities provided false information, false messages could be delivered before the UA deployer could change the configuration.

> Any protocol that relies on any number of entities to provide true information — regardless of how and where it sources the information — carries this risk. It is inherent to the design, not the protocol.

LayerZero V2 reduces this risk by abstracting the verifier entities required to deliver a message. There can be more than two entities as well. The new verification rules provide configuration to allow any EOA or contract to be a verifier (called a "DVN"), which gives OApps the flexibility to decide how they want to perform validation.

A DVN implementation example is a multi-sig, where there is a quorum of any number of entities (configured by the deployer) so that message approvals (i.e., the act of a verifier approving of a message) can be done off chain by signing a specific payload and the signatures can be submitted in a single transaction.

> In general, the more entities required to assert the validity of a message, the lower the risk of a collusion attack.

5.2. Risks of default configurations

Zellic would like to clarify the default configurations aspect of LayerZero V2's threat model for those who are not intimately familiar with the protocol.

We want to dispel any unreasonable concerns about centralization risks but plainly document any true risks for the benefit of protocol developers and users.

Types of configuration

LayerZero V2 provides the following general types of configuration to OApps:

Messaging libraries: OApps' messaging-library choices are restricted to those approved by LayerZero Labs. Note that messaging libraries may have specific configuration options such as the number of inbound confirmations required to deliver a message.

- **Send library:** This library is responsible for fee calculation and interacting with off-chain entities.
- **Receive library:** This library is responsible for message validation (i.e., ensuring all proper DVNs approve of the message).

DVNs/verifiers: In the receive library, the DVNs can be configured to some extent depending on the library implementation. DVNs are responsible for ensuring packets are valid, but because they are abstracted, the logic or process for determining what constitutes a valid packet is up to the DVN (i.e., it is configurable). In all receive libraries as of the time of this writing, the OApp may manually configure the DVN to whatever they desire.

- **Required DVNs:** All of these types of configured DVNs must approve of the packet, or the packet will be rejected.
- **Optional DVNs:** A quorum of these types of configured DVNs must approve of the packet, or the packet will be rejected. Note that the quorum number is also configurable by the OApp.

Abilities of LayerZero Labs

LayerZero Labs has the following abilities and limitations relating to configuration:

- All configuration options have defaults that can be set by LayerZero Labs at any time.

  > For all configuration options, LayerZero Labs does not have the ability to override user-set configuration. However, if the values are unset by the user, the LayerZero Labs–specified defaults — which can be changed at any time — will be used.

- LayerZero Labs has the ability to add permitted messaging libraries at any time and may permit malicious libraries. However, as previously mentioned, user configuration overrides default configuration. If the user has configured a specific library version, LayerZero Labs does not have the ability to change the OApp's receive library to a malicious one.
- LayerZero Labs does not have the ability to remove messaging libraries. An implication of this fact is that it is not possible for LayerZero Labs to deny service by disabling or preventing execution of a user-configured library.
- Contracts are immutable; that is, core LayerZero V2 contracts cannot be upgraded by LayerZero Labs or any entity.

Recommendations to OApp deployers

Based on these abilities of LayerZero Labs, if an OApp does not desire a specific custom configuration, we recommend fixing the configuration to the current default configuration at the time of deployment.

Note that the purpose of LayerZero Labs's ability to change the default configuration is to make the protocol future-proof without upgradability. As time passes, vulnerabilities or functionality bugs could potentially be discovered in the messaging libraries. Similarly, new technology may be developed to increase the efficiency of verification (i.e., gas savings).

To that end, we recommend that OApp deployers evaluate their present configuration and LayerZero Labs's new default configuration whenever their default configuration changes.

5.3. Threat model regarding channels

During core message processing, the channel is responsible for handling the

- `CHANNEL_SEND`,
- `CHANNEL_COMMIT_PACKET`,
- `LZ_RECEIVE_PREPARE`,
- `LZ_RECEIVE_LOCK`, and
- `LZ_EXECUTE_CALLBACK`

opcodes.

The data structures held in the channel state are the

- commit pipeline,
- execution pipeline,
- execution queue, and
- send queue.

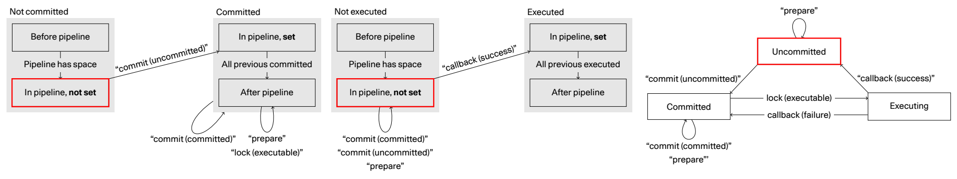
Most operations act upon a single nonce, and these data structures essentially hold state for each nonce. So we can describe the effect of channel operations in terms of how a given nonce's state is updated, in each of the data structures.

We will describe these transitions in terms of the following operations, which correspond to input and output messages as follows.

Operation	Input message	From	Output message	To
commit (committed)	`CHANNEL_COMMIT_PACKET`	endpoint	`COMMIT_PACKET_CALLBACK`	connection
commit (uncommitted)	`CHANNEL_COMMIT_PACKET`	endpoint	`COMMIT_PACKET_CALLBACK`	connection
prepare	`LZ_RECEIVE_PREPARE`	executor	`LZ_RECEIVE_PREPARE`	oapp
lock (executable)	`LZ_RECEIVE_LOCK`	oapp	`LZ_RECEIVE_EXECUTE`	oapp
lock (not executable)	`LZ_RECEIVE_LOCK`	oapp		
callback (success)	`LZ_EXECUTE_CALLBACK`	oapp		
callback (failure)	`LZ_EXECUTE_CALLBACK`	oapp		

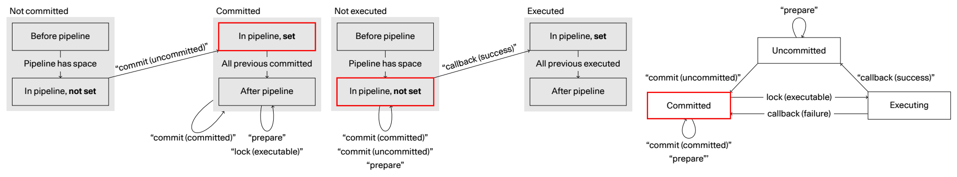
The following diagrams approximate the transitions of the commit pipeline, execution pipeline, and execution queue. To give a more concrete example, let us consider the trajectory of a single nonce in a complete interaction with the channel. A nonce might begin

- **not set** (commit pipeline), **not set** (execution pipeline), and **uncommitted** (execution queue).



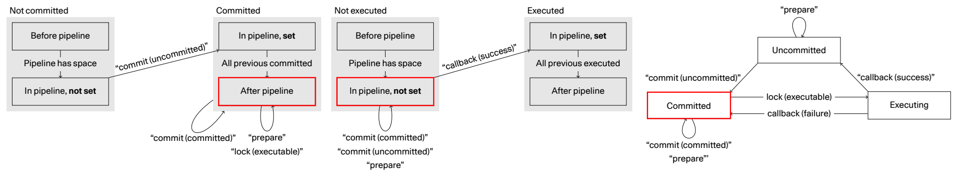
The endpoint issues a `CHANNEL_COMMIT_PACKET` message, which in this case corresponds to "commit (uncommitted)" in our diagram. The `COMMIT_PACKET_CALLBACK` message is sent to the connection, and the nonce becomes

- **set** (commit pipeline), **not set** (execution pipeline), and **committed** (execution queue).



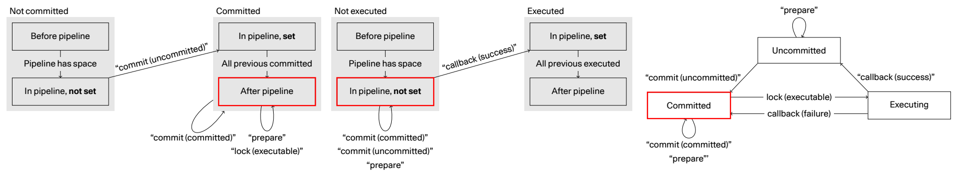
We wait some time for all previous nonces to be committed, so the nonce is

- **"after"** the commit pipeline, **not set** (execution pipeline), and **committed** (execution queue).



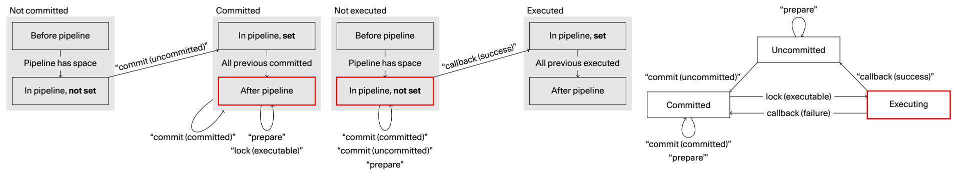
At this point, the nonce's state allows the `LZ_RECEIVE_PREPARE` message ("prepare" in our diagram) to be issued by the executor. This results in the `LZ_RECEIVE_PREPARE` message being forwarded to the OApp, and the nonce remains

- **"after"** the commit pipeline, **not set** (execution pipeline), and **committed** (execution queue)



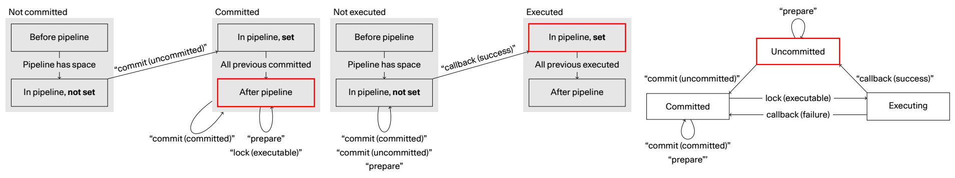
When the OApp receives the message, it can itself send `LZ_RECEIVE_LOCK` ("lock (executable)"). The channel will respond to the OApp with `LZ_RECEIVE_EXECUTE`, and the nonce will be moved to

- **"after"** the commit pipeline, **not set** (execution pipeline), and **executing** (execution queue).



Finally, suppose the message execution succeeds. The OApp will respond with `LZ_EXECUTE_CALLBACK`, which corresponds to “callback (success)” in our diagram. The nonce will then be

- **“after”** the commit pipeline, **set** (execution pipeline), and **uncommitted** (execution queue).



This means that the execution pipeline can roll forward, and the slot in the execution queue is freed up for a future nonce.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed.

During our assessment on the scoped LayerZero TON contracts, we discovered 14 findings. Three critical issues were found. Three were of high impact, two were of medium impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.