PUBLIC

# Code Assessment

## of the LayerZeroReceiver Smart Contract

September 09, 2025

Produced for

by

# Contents


1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Open Findings 10
6 Resolved Findings 11
7 Informational 15
8 Notes 16

# 1 Executive Summary

Dear all,

Thank you for trusting us to help Enso with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of LayerZeroReceiver according to Scope to support you in forming an opinion on their security risks.

Enso implements a LayerZero v2 compose receiver that allows users to execute Shortcuts through OFT bridging.

The most critical subjects covered in our audit are functional correctness, correct integration with LayerZero, and the security of funds. The general subjects covered are gas efficiency and operational considerations.

The most significant issues uncovered include:

- Ineffective griefing protection, demonstrated how the griefing protection was bypassable under certain conditions.
- Risky sweep mechanism, highlighted the risk that swept fund could be moved without preventing the corresponding message from later being executed.

Both above mentioned issues were resolved accordingly.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 4 |
| • Code Corrected | 4 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Shortcuts Client Contracts repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 18 August 2025 | 2a4959050cd14de534f2b02486135a177c9acced | Initial Version |
| 2 | 04 September 2025 | 12c05d29c4fab5829c68c692d6006101d179e78c | After Intermediate Report |
| 3 | 08 September 2025 | 6fb165ee59312a106f2a293f681032f222478d55 | Final Version Report |

For the solidity smart contracts, the compiler version `0.8.28` and evm version `cancun` was chosen.

The files in scope for this review were:

```
src/bridge/LayerZeroReceiver.sol
```

### 2.1.1 Excluded from scope

Any other file in the shortcuts client contracts repository was out of scope for this review. All dependencies and libraries are out of scope and expected to work correctly as documented.

LayerZero V2 itself is out of scope and assumed to function correctly as per its documentation.

The OFT implementations are out of scope and are expected to follow the standard. Their configurations and settings (e.g. libraries, DVN, executor) are out of scope.

Tests and imports, including the imported LayerZero contracts, are out of scope and treated as external dependencies for this review.

## 2.2 System Overview

This system overview describes the latest received version (Version 2) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Enso implements a LayerZero v2 compose receiver that allows users to execute Shortcuts through OFT bridging.

### 2.2.1 Context

LayerZero v2 is a cross-chain messaging bridge that allows sending messages from source chains to destination chains. Generally, the bridging process can be described as follows:

1. User interacts with an OApp (Omnichain Application) on chain X.
2. OApp initiates a cross-chain message through the LayerZero v2 endpoint. Note that fees for gas and native token drops are paid when sending the message.
3. LayerZero v2 off-chain mechanisms validate the message (e.g., DVN).
4. Eventually, the message becomes executable on chain Y. It can be executed by arbitrary addresses but is typically executed by the executor.
5. When executed, the LayerZero v2 endpoint on chain Y calls the receiving OApp.

Note that some OApps support composed messages. These are custom messages specified by users to target further integrations and can be described as follows:

1. The user provides the composed message to the OApp on the sending chain X.
2. The full message is bridged as above.
3. The OApp handles its message accordingly. If the message contains a composed message, the OApp queues the message for execution. Note that the composed message becomes immediately executable.
4. The execution can be triggered by arbitrary addresses, but is triggered by the executor. Note that the endpoint invokes a call to the target contract as a consequence.

The most prominent use case for composing messages is pairing regular token bridging with functionality that directly handles the tokens further. LayerZero v2 OFTs (Omnichain Fungible Tokens) typically support such composing. More concisely, after allocating tokens to the recipient, a call is composed if necessary and desired.

### 2.2.2 LayerZeroReceiver

The `LayerZeroReceiver` implements a call composer for OFTs that allows composing token bridging with Shortcut executions.

More specifically, the contract allows whitelisted OFTs to compose calls to it. Hence, `ILayerZeroComposer.lzCompose` is implemented.

The expected compose message corresponds to `address receiver, uint256 nativeDrop, uint256 estimatedGas, bytes memory shortcutData` where the receiver corresponds to the recipient of bridged funds in case of shortcut execution failure, and where the shortcut data corresponds to the data defining the shortcut execution.

`lzCompose` can be outlined as follows:

1. Note that when the function is called, the funds have already been allocated to the contract (not including native drops, which are attached as value on the call).
2. Access control and decoding. Additionally, it is validate that the message key has not been swept.
3. Self-call to `execute` which ensures that the `EnsoRouter` can be called accordingly through `routeSingle` or `routeMulti` (latter in case of native drop when bridging ERC-20s). Note that the router moves the funds to shortcuts and runs the script according to `shortcutData`.
4. If the self-call to `execute` succeeds, the execution finishes. If the execution was unsuccessful, the bridged amounts are sent to the `receiver`. Note that as long as the `estimatedGas` at the right location is satisfied, the user accepts the terms that funds will be automatically forwarded.

Further, the following functionality is available:

- Functionality defined in `Ownable`.
- The owner can retrieve any amount of any token from the contract with `sweep`. Note that double-sweeping the same message key is intended to allow for maximum flexibility and handling corner cases.
- The owner can assign/revoke the registrar role with `setRegistrar` and `removeRegistrar`.
- Registrars can add/remove support for OFTs with `setOFTs` and `removeOFTs`.

### 2.2.3 Changelog

In Version 2, the following changes were made:

- In previous version, the compose message had fewer parameters. More specifically, it was `<address receiver><bytes shortcutData>`.
- To resolve a double-spending issue, messages are marked now as swept when `sweep` is called. In prior versions, that was not the case.
- In prior versions, the notion of "reserveGas" existed. However, due to an alternative mechanism being implemented, the setter `setReserveGas` and related functionality was removed.

## 2.3 Trust Model

This section outlines the trust model considered for this security review. Below, we list the roles with the level of trust.

The following general roles are defined:

- *Owner*: Fully trusted. Can drain any funds bridged to the contract for composing purposes by sweeping or through registrar capabilities.
- *Registrars*: Fully trusted. Can drain any funds bridged to the contract for composing purposes by adding support for malicious OFTs.
- *Whitelisted OFT's*: Fully trusted. Note that this includes the respective OFTs for all chains the OFT supports. Additionally, this includes the OFT's off-chain components (e.g. DVN). Malicious OFTs could, in the worst case, drain all funds similar to registrars.
- *OFT's underlying token*: Trusted to work as intended. However, note that special tokens, such as rebasing tokens and tokens with transfer fees, are unsupported.
- *LayerZero v2 endpoint*: Trusted but expected to work as documented. Issues may lead to draining of funds in the worst case.
- *Enso Shortcuts and Router*: Trusted but expected to work correctly. Incorrect execution may lead to loss of funds. Expected to not revert if too much `msg.value` is provided.
- *Users*: Untrusted.

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 4 |

- Drop Manipulation May Lead to Reverts Code Corrected
- Ineffective Griefing Protection Code Corrected
- Missing Constructor Event Code Corrected
- Risky Sweep Mechanism Code Corrected

| | |
|---|---|
| Informational Findings | 2 |

- Unused Events and Errors Code Corrected
- Gas Optimizations Code Corrected

## 6.1 Drop Manipulation May Lead to Reverts

Design Low Version 1 Code Corrected

*CS-ENSO-LZR-001*

The caller of the composed message execution in the LayerZero Endpoint can provide a non-zero `msg.value` when invoking `lzCompose`. This amount is added to the native drop when building the router call. Although the code aggregates both amounts downstream execution may revert upon receiving this unexpected value (e.g. insufficient amount, excessive amount).

Ultimately, providing undesired `msg.value` to `lzCompose` can lead to execution griefing, which, ultimately, leads to skipping shortcuts and transferring tokens to the receiver by manipulating execution variables.

---

**Code corrected:**

The composed message now encodes the expected native drop amount and the contract enforces that `msg.value` of the call to `lzCompose()` is at least the encoded amount, removing the possibility of drop manipulation (assuming that `msg.value` higher than expected is not problematic in the scripts).

## 6.2 Ineffective Griefing Protection

Design Low Version 1 Code Corrected

*CS-ENSO-LZR-002*

The protection mechanism against griefing attacks in `lzCompose()` can be trivially bypassed, allowing attackers to force token transfers without executing the shortcut logic.

The code attempts to detect out-of-gas griefing with two checks:

First, execution is only attempted if there is a minimum amount of gas:

```
uint256 availableGas = gasleft();
if (availableGas < reserveGas) revert InsufficientGas(_guid);
```

After the try/catch execution:

```
if (err.length == 0 && gasleft() < reserveGas - _TRY_CATCH_GAS) {
    revert InsufficientGas(_guid);
}
```

Note that various griefing possibilities and problems exist. The various problems are outlined below:

**Try-catch gas overestimation.** Since the try/catch gas overhead is overestimated by `_TRY_CATCH_GAS = 2,000`, there is a window to bypass this check. Consider the following example:

1. Attacker provides sufficient gas to pass the initial check: `availableGas >= reserveGas`.
2. Execution receives `availableGas - reserveGas` and consumes all.
3. After try/catch: `gasleft() = reserveGas - actual_overhead`.
4. Since `actual_overhead < _TRY_CATCH_GAS` holds, `reserveGas - actual_overhead >= reserveGas - 2,000` holds.
5. *Tokens are transferred without the shortcut execution*.

**Opcode Pricing Might Change.** Note that opcode pricing may also change over time. As a consequence, a fixed `_TRY_CATCH_GAS` may not lead to the desired results. Depending on the scenario, this may lead to *always reverting or token transfers without shortcut execution* similar to the above case.

**63/64 rule.** The gas forwarded to a call is `min(availableGas - reserveGas, gasleft()*63/64)`. In case the second term is forwarded, griefing can occur. Ultimately, that leads to a scenario where `> reserveGas` is available directly after the call. If the delta of gas above `reserveGas` is sufficiently high, then the if-condition will not be satisfied. *Tokens will be transferred without the shortcut execution* similar to the above cases.

**OOG in nested calls.** OOG cases in nested calls will lead to bubbling up of revert reasons. In these cases, due to the 63/64 rule, the top-level call `this.execute` will not use all gas but revert with OOG. Similarly, to the previous item, if the delta is sufficiently high, the *tokens will be transferred without the shortcut execution* similar to the above cases.

**OOG with custom errors.** Some OOGs might have an error length greater than 0. For example, a call could wrap all reverts from its calls in custom errors for bubbling them up (e.g., Mangrove). In such cases, OOG would not be detected. Consider the following example:

1. Attacker provides sufficient gas to pass the initial check: `availableGas >= reserveGas`.
2. Execution receives `availableGas - reserveGas`.
3. Eventually a call to contract X is made. That contract performs a low-level call to contract Y. The call to contract Y reverts. X detects the revert and reverts with `CustomError`.
4. The error's length is greater than 0. However, OOG occurred.
5. *Tokens are transferred without the shortcut execution.*

**Other 0-length errors.** Alternatively, note that other reverts lead to no revert data (e.g. `abi.decode`) and thus potentially to cases where the `_TRY_CATCH_GAS` window will detect an OOG where no OOG is. As a consequence, the execution could potentially *always revert*.

**Summary.** The logic aiming to protect against griefing can be bypassed or can break in various ways.

---

**Code corrected:**

The code has been adjusted to always revert if the new parameter `estimatedGas` specified by the user is violated. For any other revert, the user accepts that the amounts will be sent to the receiver as it is in an input mistake.

## 6.3 Missing Constructor Event

Correctness | Low | Version 1 | Code Corrected

*CS-ENSO-LZR-003*

The constructor sets the initial registrar but does not emit the event `RegistrarAdded`.

---

**Code corrected:**

The constructor now emits the `RegistrarAdded` event.

## 6.4 Risky Sweep Mechanism

Design | Low | Version 1 | Code Corrected

*CS-ENSO-LZR-004*

The contract provides a sweep function that allows the owner to recover tokens from the contract.

In normal operation, token transfers arrive at this receiver contract and are later forwarded through `lzCompose` message execution.

The sweep function must only be used for messages that are certain never to become executable. If tokens are swept and the corresponding message later becomes executable, execution may succeed and move unintended funds from the contract. This makes the sweep operation risky and requires extreme caution in its use.

Otherwise, scenarios as outlined below might be possible:

1. Alice bridges 1M USDC with a composed message.
2. The composed message execution always reverts, and funds are stuck.
3. Governance decides to sweep and reimburse Alice.
4. Bob bridges 1M USDC with a composed message.
5. Suddenly, Alice's composed message can be executed. Alice executes it, consuming Bob's funds.
6. Bob's funds are lost.

Ultimately, there is no mechanism to invalidate a refunded message, so if it later becomes executable it may still be executed.

---

**Code corrected:**

A `messageExecuted` mapping was added to track swept messages:

- In case of `lzCompose`, the message key is validated not to be swept.
- In case of `sweep`, the same key can be swept multiple times to allow handling corner cases but is marked as used in the mapping.

Effectively, that prevents any composing after a message is swept. However, the mechanism may lead to sweeping more often. For example, sweeping an unexecutable message with a given key, could lead to requiring to sweep also for a message with the same key (index is different).

## 6.5 Unused Events and Errors

**Informational** **Version 2** **Code Corrected**

*CS-ENSO-LZR-007*

The changes introduced in **Version 2** removed the need for some events and errors. However, they still remain declared:

1. Event `ReserveGasUpdated` is unused.
2. Error `InvalidArrayLength` is unused.

---

**Code corrected:**

The unused code was removed.

## 6.6 Gas Optimizations

**Informational** **Version 1** **Code Corrected**

*CS-ENSO-LZR-005*

Some computations can be optimized to reduce the gas consumption. Below is a non-exhaustive list of potential meaningful optimizations:

1. In `lzCompose` the storage variable `reserveGas` is read 3 times. However, it could be cached in memory to save gas.

---

**Code corrected:**

The `reserveGas` variable has been removed in the updated code.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 No-op Events

**Informational** **Version 1**

*CS-ENSO-LZR-006*

The owner and registrars can configure the contract. Events are emitted upon updates to track state changes. However, events are also emitted even when no actual state change occurs, resulting in no-op informational events.

```
event OFTAdded(address oft);
event OFTRemoved(address oft);
event RegistrarAdded(address account);
event RegistrarRemoved(address account);
event ReserveGasUpdated(uint256 amount);
```

# 8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

## 8.1 Alternative Griefing Possibilities

Note Version 2

Note that this was initially highlighted as part of the resolved issue Ineffective griefing protection. This note aims to highlight that other way of forcing transfers to users exist. In some cases, `lzCompose` could be wrapped with manipulations to create regular reverts (error length greater than 0). Consider the following example:

1. An `lzCompose` to swap USDC to USDT on a Uniswap V3 pool is queued.
2. An attacker calls `flash` on the respective pool.
3. Within the callback, they call `lzCompose`.
4. The swap reverts due to the reentrancy lock.
5. Tokens are transferred without the shortcut execution.

To summarize, users should be aware that their execution could be griefed.

## 8.2 Arbitrary Shortcut Scripts

Note Version 1

Users, frontends and governance should be aware that the provided shortcut scripts can be arbitrary and that no checks within the execution of `LayerZeroReceiver` are enforced. Checks (e.g. slippage protection, amount of received tokens), should be ensured to be part of the respective scripts. Bad scripts (e.g. lack of checks, reenterable by an attacker) may lead to a loss of funds.

In the context of `LayerZeroReceiver`, the abuse of bad scripts is simplified since `lzCompose` can be initiated by arbitrary parties and, thus, could be, for example, sandwiched by potential attackers.

## 8.3 Possibility to Craft Unexecutable Message

Note Version 1

Input parameters of a compose message cannot be validated on the source chain, making it possible to craft an unexecutable message (e.g. decoding of message reverts, recipient being `0x0`). In such a case, the sender would only lock their own funds. The contract provides a `sweep()` function that allows the admin to recover these funds if needed.

This note only highlights the theoretical possibility. Users are not expected to do this, and the situation is comparable to other incorrect user actions that may result in lost or locked funds in such systems.

## 8.4 Sweep Considerations

Note Version 2

Governance should be aware of the details of the sweep mechanism and the blocking of messages as a consequence.

Generally, the mechanism can be described as follows: if a message key has been swept, it cannot be executed anymore through `lzCompose`. Note that the message key is defined as follows:

```
function getMessageKey(address from, bytes32 guid, bytes calldata message) public pure returns (bytes32) {
    return keccak256(abi.encode(from, guid, message));
}
```

Since compose messages have an index, the message key may not be unique. As a consequence, some compose messages could share the same key:

- are part of the same cross-chain message (same GUID but different index)
- and if they their message has the same content.

That leads to corner cases for governance to consider:

1. If two messages share the same key and one is executable and one is unexecutable, sweeping the unexecutable message before the executable one is executed, may lead to additional sweeping. Namely, the executable one would also become unexecutable. If the message is executed first, sweeping should be acceptable afterwards.
2. If two messages share the same key and both are unexecutable, both can be swept in separate calls.

Additionally, governance should be aware that the sweep mechanism is designed to be as flexible as possible:

1. No requirements on token or amount to sweep.
2. Double-sweeping possible. Additionally, sweeping arbitrary keys is possible.
3. Sweeping executed messages possible.

However, that is in accordance with the Trust Model.