Hyperliquid Composer

Security Assessment

August 4th, 2025 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

LayerZero engaged OtterSec to assess the `hyperliquid-composer` program. This assessment was conducted between July 28th and July 30th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, we identified an issue where, due to the asynchronous HyperCore balance reads, multiple lzCompose calls in the same transaction may utilize a stale balance, breaking capping logic (OS-HLC-SUG-01). We also suggested adding a minimum gas check to prevent malicious executors from deliberately failing refund calls via gas exhaustion (OS-HLC-SUG-00). We also made recommendations for modifying the codebase to improve consistency and functionality (OS-HLC-SUG-03).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/devtools. This audit was performed against commit bd7f2e7. Follow-up reviews wer performed against commits 8950c71, da5a74c, 4c0c4c7, and badaf83. We also reviewed an additional scope at repository https://github.com/LayerZero-Labs/hyperliquid-composer-deployment against commit 32a65b4.

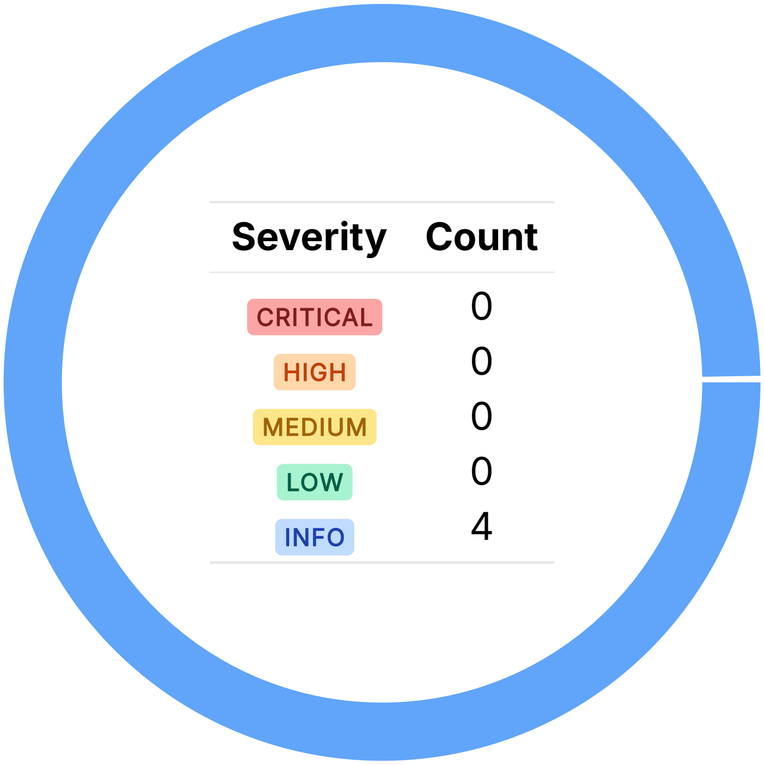
A brief description of the program is as follows:

Name	Description
hyperliquid-composer	It receives cross-chain OFT messages from LayerZero and bridges assets from HyperEVM to HyperLiquid L1 (HyperCore).

02 — Findings

Overall, we reported 4 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	0
INFO	4

03 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-HLC-SUG-00	Suggestion to add a minimum gas check to prevent failure of refund calls via gas exhaustion.
OS-HLC-SUG-01	Due to the asynchronous `HyperCore` balance reads, multiple `lzCompose` calls in the same transaction may utilize a stale balance, breaking capping logic.
OS-HLC-SUG-02	Suggestion to maintain code consistency regarding the expected behavior when the transferred amount exceeds the available bridge supply.
OS-HLC-SUG-03	Recommendation for modifying the codebase to improve consistency and functionality.

Minimum Gas Check to Prevent Gas Exhaustion OS-HLC-SUG-00

Description

There is no minimum gas check before executing the refund logic in the composer, allowing malicious executors to deliberately exhaust gas in a try-catch block and trigger refund failures. This may be exploited to manipulate the fallback path and steal the dust amount.

Remediation

Add a minimum gas check before executing the refund logic in the composer.

Patch

Resolved in PR#1642.

Utilization of Stale Balance via Async Check OS-HLC-SUG-01

Description

Currently, due to the asynchronous behavior in HyperLiquid when checking the HyperCore bridge balance via a precompile, the capping logic may fail if two `lzCompose` calls are executed within the same transaction. Both executions read the same stale balance before either transfer is applied on HyperCore, resulting in the first execution to consume the balance while the second still sees the old value. This may result in approving more tokens than the bridge actually holds.

Remediation

Track an in-flight reserved balance locally, reducing the available amount for subsequent calls in the same transaction.

Patch

LayerZero acknowledged this issue.

Unreachable Bridge Supply Check OS-HLC-SUG-02

Description

Currently, both functions `_transferNativeToHyperCore` and `_transferERC20ToHyperCore` contain a check that reverts when the user-provided `amounts.core` exceeds the bridge supply. However, the `amounts.core` value used in this check has already been capped to the bridge supply, making this revert condition unreachable.

Remediation

Either remove this redundant check or move the revert check to occur before the amount is capped.

Patch

Resolved in fb03ebf.

Code Refactoring OS-HLC-SUG-03

Description

1. `FeeToken::activationFee` computes `10 ** CORE_SPOT_DECIMALS`, which may overflow if `CORE_SPOT_DECIMALS` becomes too large due to mismatched token decimals. Validate that `quoteAssetInfo.weiDecimals <= baseAssetInfo.szDecimals + 11` to ensure the decimal difference stays within a safe range, preventing overflow.
2. Native transfers to `HyperCore` require an `activationFee` if the user account is not yet activated. This should be addressed by blocking activation attempts when the transfer involves only the native token (`HYPE`) without any ERC20 tokens.
3. The maximum user activation per block check may not be strict enough because while it's based on the expected maximum transactions per block, multiple `lzCompose` calls within a single transaction could bypass this rate limiting. Since each `lzCompose` call can trigger user activation, a single transaction containing multiple such calls could exceed the intended activation limit per block. Consider implementing additional safeguards to handle this scenario properly.

Remediation

Incorporate the above suggestions.

Patch

Resolved in PR#1774.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.