LayerZero Eigen DVN

Security Assessment

September 18th, 2025 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

LayerZero engaged OtterSec to assess the `eigen-dvn` program. This assessment was conducted between September 15th and September 16th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 1 finding throughout this audit engagement.

We made recommendations for implementing proper validations to ensure adherence to coding best practices (OS-EVN-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/DVN-AVS. This audit was performed against commit 450aa06.

A brief description of the program is as follows:

Name	Description
eigen-dvn	Provides a network of staked operators that perform off-chain verification tasks. It leverages EigenLayer's restaking to ensure operators are economically secured and can be slashed for misbehavior.

02 — 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-EVN-SUG-00	There are several instances where proper validation is not performed, resulting in unexpected failures and potential security issues.

Missing Validation Logic

OS-EVN-SUG-00

Description

1. `LayerZeroSlasher::queueRequest` allows unbounded description strings, which may bloat storage and increase gas costs. Add a maximum length check to limit the length of the request description, improving efficiency.

   >_ *eigenlayer/LayerZeroSlasher.sol* SOLIDITY

   ```
   /// @inheritdoc ILayerZeroSlasher
   function queueRequest(string memory _description) external {
       uint256 id = nextRequestId++;
       _requests[id] = Request({
           description: _description,
           expiry: block.timestamp + APPROVAL_WINDOW,
           status: Status.QUEUED,
           bondToken: bondToken,
           bondAmount: bondAmount,
           bonder: msg.sender,
           slashId: 0, // EigenLayer slash ID starts at 1, can safely initialize this to 0
           response: ""
       });
       bondToken.safeTransferFrom(msg.sender, address(this), bondAmount);
       emit RequestQueued(id, msg.sender, _description);
   }
   ```

2. Currently, `LayerZeroSlasher::getRequest` just returns the structure from `_requests[_id]` even if that ID has never been queued. This is misleading because a caller may think they are fetching a real slashing request when, in fact, no such request exists. Add a revert if `status == null` to prevent this ambiguity and ensure only real requests may be queried.

   >_ *eigenlayer/LayerZeroSlasher.sol* SOLIDITY

   ```
   /// @inheritdoc ILayerZeroSlasher
   function getRequest(uint256 _id) external view returns (Request memory) {
       return _requests[_id];
   }
   ```

3. The constructor in `LayerZeroSlasher` accepts `_avsRegistrar`, `_allocationManager`, and `_operatorSetId` but does not verify they are consistent. If the registrar is bound to a different operator set or allocation manager, the slasher will be misconfigured. Validate the passed-in `AVSRegistrar` has the same values for `operatorSetId` and `allocationManager` as the ones passed to the constructor.

Remediation

Add the missing validations.

Patch

1. Issue #1 was fixed in PR#26.
2. Issue #2 was acknowledged
3. Issue #3 was acknowledged.

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.