hexens x DIN Decentralized Infrastructure Network

Security Review Report for DIN

October 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review of DIN AVS, which is an AVS implementation for EigenLayer, developed by Infura.

Our security assessment was a full review of the code, spanning a total of 1 week.

During our review, we did not identify any major severity vulnerabilities.

We did identify several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

- **Review Led by**

Jahyun Koo, Lead Security Researcher

- **Scope**

The analyzed resources are located on:

🔗 https://github.com/DIN-center/din-avs/tree/3200004d10d3a1b10b9a40bf3adecb870906b036

The issues described in this report were fixed in the following commit:

🔗 https://github.com/DIN-center/din-avs/commit/6e8deb6777368d74f9b7169cc23d875cb09a446d

- **Changelog**

13 October 2025	Audit start
20 October 2025	Initial report
31 October 2025	Revision received
6 November 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

▪ Severity Characteristics

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical	Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.
High	Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

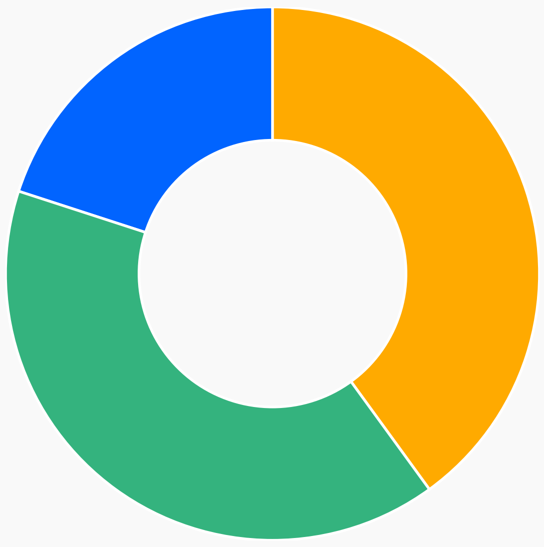
▪ Issue Symbolic Codes

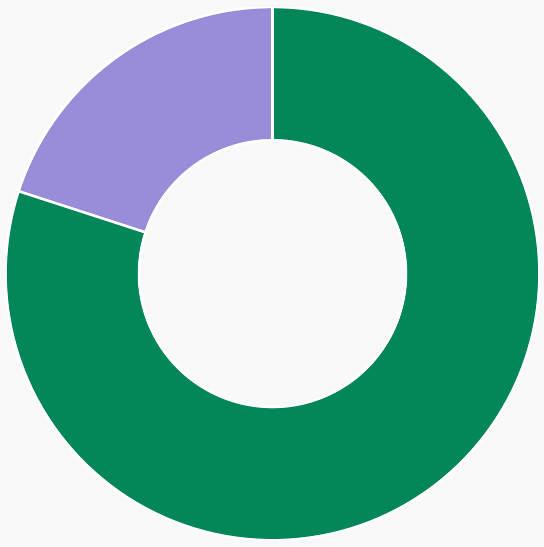
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
🟥 Critical	0
🟥 High	0
🟧 Medium	2
🟩 Low	2
🟦 Informational	1
Total:	**5**





6. Weaknesses

This section contains the list of discovered weaknesses.

INFU1-1 | Pre-veto operator removal on slashing queue lacks automatic rollback

Acknowledged

Severity:	Medium	Probability:	Rare	Impact:	High

Path:

src/ServiceManager.sol

Description:

In **ServiceManager.queueSlashingRequest()**, the function immediately removes the operator from the allowlist, disqualifies them, and deregisters them from operator sets before the veto period concludes. When the veto committee cancels a slashing request via **VetoableSlasher.cancelSlashingRequest()**, only the slashing request status is updated to **Cancelled**, but the operator's state changes (allowlist removal, disqualification, deregistration) are not reverted.

This defeats part of the purpose of the veto mechanism, which is designed to prevent incorrect slashing actions. As a result, an authorized watcher can cause disruptive state changes that persist even if the veto committee cancels the request.

- Example scenario:

1. A Watcher queues a slashing request (whether by mistake or misjudgment)
2. The operator is immediately removed from allowlist, disqualified, and deregistered
3. The operator cannot provide services during the veto period
4. The veto committee identifies the error and cancels the request
5. The operator's funds are protected, but the operator remains in a disabled state
6. Manual recovery is required: Allowlister must re-add to allowlist, Owner must re-qualify, and operator must re-register

```
    function queueSlashingRequest(IAllocationManager.SlashingParams calldata params) external override
onlyWatcher {
        // Ensure the operator is not a watcher or router
        require(params.operatorSetId > WATCHER_OPERATOR_SET_ID, "can't slash watchers or routers");

        // Validate slashing parameters to prevent precision loss
        for (uint256 i = 0; i < params.strategies.length; i++) {
            // Calculate percentage from wads (params.wadsToSlash is in 18 decimal precision)
            uint256 slashPercentage = params.wadsToSlash[i];
            validateSlashingParams(params.operator, params.strategies[i], slashPercentage);
        }

        _queueSlashingRequest(params);

        _removeOperatorFromAllowlist(uint8(params.operatorSetId), params.operator);
        _disqualify(uint8(params.operatorSetId), params.operator);

        // Deregister operator from operator sets
        uint32[] memory operatorSetIds = new uint32[](1);
        operatorSetIds[0] = params.operatorSetId;
        IAllocationManagerTypes.DeregisterParams memory deregisterParams =
IAllocationManagerTypes.DeregisterParams({
            operator: params.operator,
            avs: address(this),
            operatorSetIds: operatorSetIds
        });

        allocationManager.deregisterFromOperatorSets(deregisterParams);
    }
```

Remediation:

- Defer side effects to fulfillment: move allowlist removal, disqualification, and deregistration to the slashing fulfill path so they occur only after the veto window has passed and the request is fulfilled.
- Or add rollback on cancel: record the operator’s pre-queue allowlist/qualification/registration state (and any data needed to restore it) and, upon cancellation, restore those states; only apply permanent removal during fulfillment.

Commentary from the client:

“Intended design choice.”

INFU1-4 | Registry not refreshed after slashing fulfillment, causing delayed enforcement

Fixed ✓

Severity:	Medium	Probability:	Likely	Impact:	Low

Path:

`src/ServiceManager.sol`

Description:

The **ServiceManager.fulfillSlashingRequest()** function overrides **VetoableSlasher** but omits the call to **slashingRegistryCoordinator.updateOperators()** that synchronizes registry state after slashing execution. The base implementation performs this update atomically to ensure **StakeRegistry** and **IndexRegistry** reflect post-slashing magnitudes and trigger automatic enforcement actions such as removing operators who fall below minimum stake thresholds.

This creates a state inconsistency where registry-coordinator–derived views lag behind the operator's actual post-slash status. As a result, follow-on enforcement actions that depend on these views, such as minimum stake validation for a quorum, are not triggered immediately.

```
function fulfillSlashingRequest(uint256 requestId) external override onlyWatcher {
    IVetoableSlasher.VetoableSlashingRequest storage request = slashingRequests[requestId];
    require(block.number >= request.requestBlock + vetoWindowBlocks, VetoPeriodNotPassed());
    require(request.status == SlashingStatus.Requested, SlashingRequestIsCancelled());

    request.status = SlashingStatus.Completed;

    _fulfillSlashingRequest(requestId, request.params);
}
```

Remediation:

After fulfilling a slashing request in **ServiceManager**, perform an immediate registry refresh consistent with the base behavior by calling **slashingRegistryCoordinator.updateOperators** (or an equivalent quorum-scoped update) for the affected operator(s). Ensure all slashing paths apply a consistent post-slash refresh so enforcement dependent on registry state occurs without delay.

INFU1-2 | Pinned eigenlayer-middleware v1.3.0 may lack audit fixes and includes not-fully-audited modules

Fixed ✓

Severity:	Low	Probability:	Rare	Impact:	Low

Description:

The project pins **eigenlayer-middleware** to v1.3.0, while v1.3.1 is available with slashing audit fixes. The v1.3.1 release addresses one Medium severity issue and several Low severity issues in the upstream middleware contracts.

Since **DINRegistryCoordinator** inherits from **SlashingRegistryCoordinator** and configures a **churnApprover**, the project potentially inherits these issues through the dependency chain. The risk is inherited at the dependency level and requires a buggy churn approver to materialize.

Remediation:

Update the **eigenlayer-middleware** submodule to v1.3.1 or later, adjust Foundry remappings accordingly, and verify compatibility with any nested dependencies such as **eigenlayer-contracts.** Recompile and test to ensure the update does not introduce breaking changes.

INFU1-5 | Mismatch in Slashing Validation Logic Can Lead to False Negatives

Fixed ✓

Severity:	Low	Probability:	Unlikely	Impact:	Low

Path:

`src/ServiceManager.sol`

Description:

The pre-flight validation logic in **ServiceManager.calculateExpectedSlash** is inconsistent with the execution logic in **AllocationManager.slashOperator.** This discrepancy can cause the validation to incorrectly reject valid slashing transactions.

The two key differences are:

1. Calculation Basis: The validation uses an operator's **getMaxMagnitude**, whereas the core execution logic uses the **currentMagnitude** of the allocation.
2. Rounding Policy: The validation's division implicitly rounds down, while the **AllocationManager** explicitly rounds up using **mulWadRoundUp**.

As a result, the validation logic is overly conservative. This can lead to a "false negative" where a legitimate slash - particularly one involving a small percentage near the precision boundary - is calculated to have an expected amount of zero and is reverted, even though it would have been successfully processed by the **AllocationManager**.

```
function calculateExpectedSlash(address operator, IStrategy strategy, uint256 slashPercentage)
    internal
    view
    returns (uint256 expectedSlash)
{
    // slither-disable-next-line calls-loop
    uint64 maxMagnitude = allocationManager.getMaxMagnitude(operator, strategy);
    expectedSlash = (uint256(maxMagnitude) * slashPercentage) / 1e18;
}
```

Remediation:

Update **calculateExpectedSlash()** to query the operator's current allocation for the specific operator set and strategy using **getAllocation()**, then apply the same round-up multiplication logic that **AllocationManager** uses during actual slash execution.

INFU1-3 | Unused Import and Custom Error in ServiceManager

Fixed ✓

Severity:	Informational	Probability:	Rare	Impact:	Informational

Description:

The **ServiceManager.sol** contract contains an unused import and an unused custom error definition.

- The import **ISlashingRegistryCoordinatorTypes** is included but is not referenced anywhere in the contract.
- The custom error **SlashingPeriodOver** is defined but is never reverted.

Remediation:

Remove the unused import statement for **ISlashingRegistryCoordinatorTypes** and the unused custom error declaration **SlashingPeriodOver** from **ServiceManager.sol.**

hexens x DIN

Decentralized
Infrastructure
Network