hexens x EigenLayer

Security Review Report For EigenLayer

April 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 covered updates to the middleware contracts of EigenLayer, which are developed for an AVS as base contracts. The update included support for the EigenLayer core slashing release.

Our security assessment was a full review of the smart contracts spanning a total of 2 weeks.

During our audit, we have identified three medium severity vulnerabilities, which could have resulted in minor impact in some edge cases.

We have also identified 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

Kasper Zwijsen, Head of Audits

Scope

The analyzed resources are located on:

🔗 https://github.com/Layr-Labs/eigenlayer-middleware

📌 **Commit:** `f5adbcac55d9336cd646ce71bc467aa7e20f1a12`

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

🔗 https://github.com/Layr-Labs/eigenlayer-middleware

📌 **Commit:** `82d91bf96e8c360041983012574dd2b82377969e`

Changelog

07 April 2025	Audit start
22 April 2025	Initial report
02 May 2025	Revision received
06 May 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.

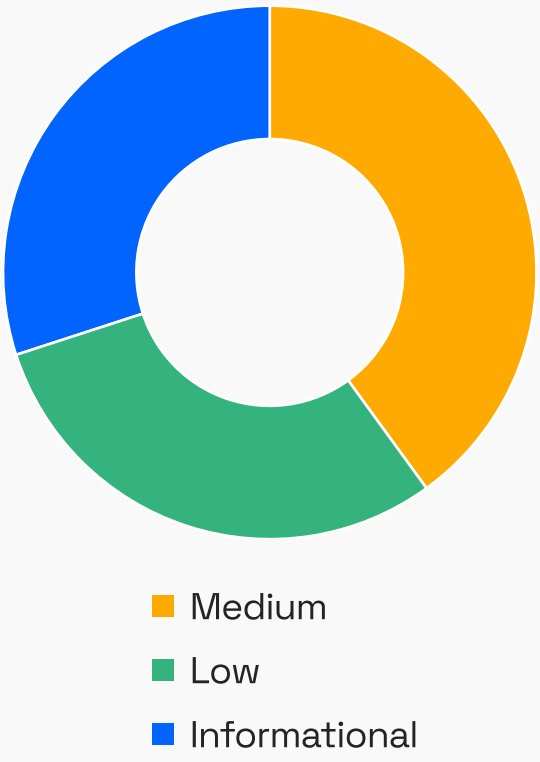
- **Issue Symbolic Codes**

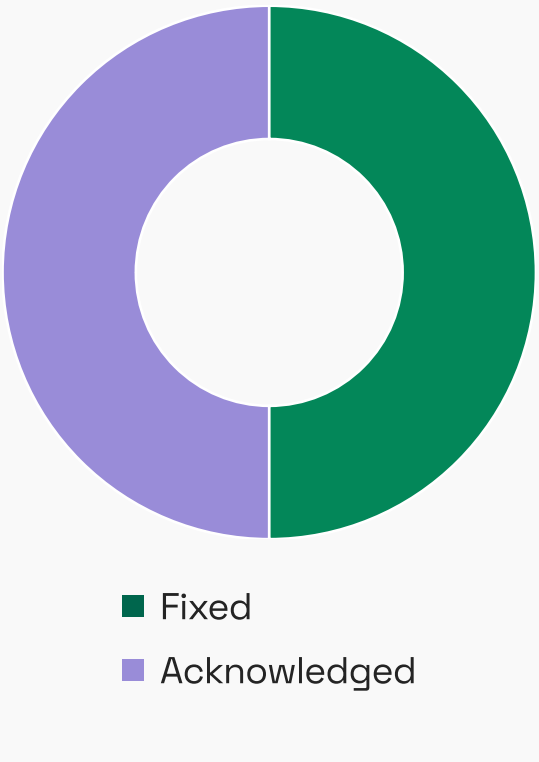
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	4
Low	3
Informational	3
Total:	**10**





6. Weaknesses

This section contains the list of discovered weaknesses.

EIGEN2-2 | Current rate limit logic allows ejection beyond allowed limit

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

EjectionManager.sol:ejectOperators

Description:

The owner or an ejector can eject operators. Only the ejector has a rate limit, which restricts how much can be totally rejected during a quorum, that is based on a limit set by the owner.

When an ejector calls **ejectOperators** to eject an operator, it checks the **amountEjectable** for the quorum. This is used as the total amount ejectors are allowed to remove.

The issue occurs in the following condition:

```
function ejectOperators(
    bytes32[][] memory operatorIds
) external {

-- SNIP --

  if (
      isEjector[msg.sender]
          && quorumEjectionParams[quorumNumber].rateLimitWindow > 0
          && stakeForEjection + operatorStake > amountEjectable
  ) {

      ratelimitHit = true;

      stakeForEjection += operatorStake;
      ++ejectedOperators;

      slashingRegistryCoordinator.ejectOperator(
```

```
            slashingRegistryCoordinator.getOperatorFromId(operatorIds[i][j]),
            abi.encodePacked(quorumNumber)
        );

        emit OperatorEjected(operatorIds[i][j], quorumNumber);

        break;
    }
```

If the rate limit is hit, it will still eject the operator and only then break the loop.

Consider the following scenario:

- **amountEjectable** = 1e16 (the allowed amount for this quorum to be removed by the ejector)
- **OperatorXAmount** = 10e18.

```
stakeForEjection + operatorStake > amountEjectable
0 + 10e18 > 1e16
```

It will eject the operator that had a value of **10e18**, even though the ejector is only allowed to eject up to 1e16.

Remediation:

Considering removing the ejecting logic in the case where the rate limit would have been hit, so it will only make ratelimitHit true and break the loop.

```
function ejectOperators(
    bytes32[][] memory operatorIds
) external {

-- SNIP --

  if (
      isEjector[msg.sender]
          && quorumEjectionParams[quorumNumber].rateLimitWindow > 0
          && stakeForEjection + operatorStake > amountEjectable
  ) {

      ratelimitHit = true;

--    stakeForEjection += operatorStake;
--    ++ejectedOperators;

--    slashingRegistryCoordinator.ejectOperator(
--         slashingRegistryCoordinator.getOperatorFromId(operatorIds[i][j]),
--         abi.encodePacked(quorumNumber)
--     );

--     emit OperatorEjected(operatorIds[i][j], quorumNumber);

      break;
  }
```

EIGEN2-4 | Missing constraint check for modification of lookahead time of slashable quorum

Fixed

Severity:	Probability:	Impact:
Medium	Rare	Medium

Path:

StakeRegistry.sol:_setLookAheadPeriod#L794-L802

Description:

The StakeRegistry exposes the external function **setLookAheadPeriod**, which calls the internal variant and allows the CoordinatorOwner to set the lookahead time for a slashable quorum directly.

However, this modifying function is missing the constraint check that exists in **SlashingRegistryCoordinator.sol:_createQuorum**:

```
require(
    AllocationManager(address(allocationManager)).DEALLOCATION_DELAY() >
lookAheadPeriod,
    LookAheadPeriodTooLong()
);
```

As a result, the CoordinatorOwner can set a look ahead period time that is greater than the deallocation delay of the AllocationManager, which would allow for voting power to be used that cannot be slashed, breaking the governance invariant.

```
    function _setLookAheadPeriod(uint8 quorumNumber, uint32 _lookAheadBlocks)
internal {
        require(
            stakeTypePerQuorum[quorumNumber] ==
IStakeRegistryTypes.StakeType.TOTAL_SLASHABLE,
            QuorumNotSlashable()
        );
        uint32 oldLookAheadDays =
slashableStakeLookAheadPerQuorum[quorumNumber];
        slashableStakeLookAheadPerQuorum[quorumNumber] = _lookAheadBlocks;
        emit LookAheadPeriodChanged(oldLookAheadDays, _lookAheadBlocks);
    }
```

Remediation:

The function **_setLookAheadPeriod** should include the same constraint check as when creating the quorum.

EIGEN2-5 | Max operator limit can be exceeded

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

SlashingRegistryCoordinator.sol:_kickOperator#L395-L409

Description:

The function **_kickOperator** is used to forcefully remove an operator from a set of quorum numbers. It is called from **ejectOperator**, **_updateOperatorsStakes** and **_registerOperatorWithChurn**. The latter has an invalid assumption on the function in that it does not guarantee the removal of an operator and the decrease of the total operator count.

In **_registerOperatorWithChurn**, it will only execute the churn if the max operator count was reached using the provided kick parameters on lines 520-533:

```
if (results.numOperatorsPerQuorum[i] > operatorSetParams.maxOperatorCount) {
    _validateChurn({
        quorumNumber: uint8(quorumNumbers[i]),
        totalQuorumStake: results.totalStakes[i],
        newOperator: operator,
        newOperatorStake: results.operatorStakes[i],
        kickParams: operatorKickParams[i],
        setParams: operatorSetParams
    });

    bytes memory singleQuorumNumber = new bytes(1);
    singleQuorumNumber[0] = quorumNumbers[i];
    _kickOperator(operatorKickParams[i].operator, singleQuorumNumber);
}
```

However, in the function **_kickOperator** on lines 406-408 you can see that it does not remove the operator if the quorum numbers to be removed are not a subset of the operator's quorum bitmap, in other words if it is not registered fully to those quorums:

```
if (quorumsToRemove.isSubsetOf(currentBitmap)) {
    _forceDeregisterOperator(operator, quorumNumbers);
}
```

This edge case can be triggered if the operator to be churned has left at least one of the to be removed quorums and a new operator has taken its place using a normal registration, bringing the total operator count again to the maximum and brining code execution into **_kickOperator**, while skipping the removal.

As a result, the total operator count can be increased for every signed churn parameters, as each one can be used to increase the count by one using the edge case. It can be abused by the caller if they own both the churn operator and the to be churned operator.

```
function _kickOperator(address operator, bytes memory quorumNumbers) internal
virtual {
    OperatorInfo storage operatorInfo = _operatorInfo[operator];
    // Only proceed if operator is currently registered
    require(operatorInfo.status == OperatorStatus.REGISTERED,
OperatorNotRegistered());

    bytes32 operatorId = operatorInfo.operatorId;
    uint192 quorumsToRemove =
        uint192(BitmapUtils.orderedBytesArrayToBitmap(quorumNumbers,
quorumCount));
    uint192 currentBitmap = _currentOperatorBitmap(operatorId);

    // Check if operator is registered for all quorums we're trying to remove
them from
    if (quorumsToRemove.isSubsetOf(currentBitmap)) {
        _forceDeregisterOperator(operator, quorumNumbers);
    }
}
```

Remediation:

While the soft-failure inside of **_kickOperator** is not necessarily a bug by itself, the misassumption of **_registerOperatorWithChurn** on this function is the problem. The function **_registerOperatorWithChurn** should validate the invariant that the total operator count was not exceeded at the end of the function regardless of what happens in **_kickOperator**.

Proof of concept:

```
function test_churn_over_max_poc() public {
    IStakeRegistryTypes.StrategyParams[] memory strategyParams = new
IStakeRegistryTypes.StrategyParams[](1);
    strategyParams[0] = IStakeRegistryTypes.StrategyParams({strategy:
mockStrategy, multiplier: 1 ether});
    vm.prank(proxyAdminOwner);
    slashingRegistryCoordinator.createTotalDelegatedStakeQuorum(
        operatorSetParams,
        1 ether,
        strategyParams
    );

    _setOperatorWeight(testOperator.key.addr, registeringStake);
    _setOperatorWeight(operatorToKick.key.addr, operatorToKickStake);

    ISlashingRegistryCoordinatorTypes.OperatorKickParam[] memory
operatorKickParams =
        new ISlashingRegistryCoordinatorTypes.OperatorKickParam[](1);
    operatorKickParams[0] =
ISlashingRegistryCoordinatorTypes.OperatorKickParam({
        operator: operatorToKick.key.addr,
        quorumNumber: 1
    });

    ISignatureUtilsMixinTypes.SignatureWithSaltAndExpiry memory
churnApproverSignature = _signChurnApproval(
        testOperator.key.addr, testOperatorId, operatorKickParams, defaultSalt,
defaultExpiry
    );

    uint32[] memory operatorSetIds = new uint32[](1);
    operatorSetIds[0] = 1;
    IAllocationManagerTypes.DeregisterParams memory deregisterParams =
IAllocationManagerTypes
        .DeregisterParams({
        operator: operatorToKick.key.addr,
        avs: address(serviceManager),
        operatorSetIds: operatorSetIds
    });

    vm.prank(operatorToKick.key.addr);
```

```
IAllocationManager(coreDeployment.allocationManager).deregisterFromOperatorSet
s(
        deregisterParams
    );
    operatorSetIds[0] = 2;
    registerOperatorInSlashingRegistryCoordinator(operatorToKick, "",
operatorSetIds);

    console.log("Total Operators: %d",
indexRegistry.totalOperatorsForQuorum(1));

    Operator memory extraOperator4 = operatorsByID[operatorIds.at(5)];
    operatorSetIds[0] = 1;
    registerOperatorInSlashingRegistryCoordinator(extraOperator4, "",
operatorSetIds);

    console.log("Total Operators: %d",
indexRegistry.totalOperatorsForQuorum(1));

    _registerOperatorWithChurn(
        testOperator, operatorKickParams, "", churnApproverSignature
    );

    console.log("Total Operators: %d",
indexRegistry.totalOperatorsForQuorum(1));
}
```

EIGEN2-14 | Stake deviations due to adding/removing strategy may allow for ejector manipulation

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

src/StakeRegistry.sol#L128-L174, src/StakeRegistry.sol#L727-L731, src/EjectionManager.sol#L135-L163

Description:

In the StakeRegistry contract, the **updateOperatorsStake()** function is triggered by the **updateOperators()** function of the RegistryCoordinator contract to update the stakes of some specific operators. It will calculate the new stake of each operator based on the current strategies of a quorum and update the new total stake of the StakeRegistry.

```
(uint96[] memory stakeWeights, bool[] memory hasMinimumStakes) =
    _weightOfOperatorsForQuorum(quorumNumber, operators);

int256 totalStakeDelta = 0;
// If the operator no longer meets the minimum stake, set their stake to zero
and mark them for removal
/// also handle setting the operator's stake to 0 and remove them from the
quorum
for (uint256 i = 0; i < operators.length; i++) {
    if (!hasMinimumStakes[i]) {
        stakeWeights[i] = 0;
        shouldBeDeregistered[i] = true;
    }

    // Update the operator's stake and retrieve the delta
    // If we're deregistering them, their weight is set to 0
    int256 stakeDelta = _recordOperatorStakeUpdate({
        operatorId: operatorIds[i],
        quorumNumber: quorumNumber,
        newStake: stakeWeights[i]
    });
```

```
        totalStakeDelta += stakeDelta;
    }

    // Apply the delta to the quorum's total stake
    _recordTotalStakeUpdate(quorumNumber, totalStakeDelta);
```

However, when adding or removing a strategy from a quorum, not all operators of that quorum will have their stakes updated immediately. Some operators may be updated first, leading to unfair competition when large stake deviations occur. The ejector role can take advantage of this to manipulate ejections as they wish.

For example, right after a strategy is added to a quorum, an ejector might post-run to update specific operators, increasing their stakes and the total stake of that quorum. After that, the **amountEjectableForQuorum** of that quorum will increase, allowing the ejector to eject other operators who still have outdated stakes. The operator selection for that ejection depends on the ejector.

Remediation:

After adding or removing a strategy in a quorum, it should be ensured that all operators of that quorum update their stakes right away, or a validation should be created to check that all operators in the quorum have been updated before executing an ejection.

Commentary from the client:

"This is an AVS-sync concern with respect to calling updateOperators to ensure liveness of stakes."

EIGEN2-8 | Redundant double check of max operator count in register operator

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

SlashingRegistryCoordinator.sol:registerOperator#L151-L225

Description:

The **registerOperator** function is called from the AllocationManager. The operator can specify a few parameters, such as the registration type.

In the case of the **NORMAL** registration type, the function simply calls the internal **_registerOperator** with **checkMaxOperatorCount: true**. This will trigger a check at the end of **_registerOperator** where the new total operator count is checked against the quorum's **maxOperatorCount**.

However, after calling **_registerOperator**, the function will again perform the same check by looping over the quorum numbers and checking the result against **maxOperatorCount**:

```
for (uint256 i = 0; i < quorumNumbers.length; i++) {
    uint8 quorumNumber = uint8(quorumNumbers[i]);

    require(
        numOperatorsPerQuorum[i] <=
_quorumParams[quorumNumber].maxOperatorCount,
        MaxOperatorCountReached()
    );
}
```

This second check is exactly the same and performed right afterwards, making it redundant and a waste of gas.

Remediation:

Remove the redundant check in **registerOperator**, as the parameter **checkMaxOperatorCount**: **true** already covers this.

EIGEN2-10 | A new operator could replace another operator with less staking amount

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

SlashingRegistryCoordinator.sol:_setOperatorSetParams

Description:

The function **_setOperatorSetParams** sets the configuration for a quorumNumber for the **SlashingRegistryCoordinator**.

Currently, there are no checks for the **kickBIPsOfOperatorStake** and **kickBIPsOfTotalStake**. However it would be better to implement some limitations on these values.

For example. in the function **_validateChurn**, these values are used to determine if the registrering user's stake is greater than the **operatorToKickStake** times **kickBIPsOfOperatorStake** its value:

```
_validateChurn
    require(
          newOperatorStake > _individualKickThreshold(operatorToKickStake,
setParams),
          InsufficientStakeForChurn()
      );
```

```
function _individualKickThreshold(
    uint96 operatorStake,
    OperatorSetParam memory setParams
) internal pure returns (uint96) {
    return operatorStake * setParams.kickBIPsOfOperatorStake /
BIPS_DENOMINATOR;
}
```

To prevent a case where a lower stake could kick a higher stake, the value should always be higher than 100% expressed in BIPS (**10_000**). It would be an unfair situation to kick out another operator while the **newOperatorStake** is lower than the **operatorToKickStake**.

```
    function _setOperatorSetParams(
        uint8 quorumNumber,
        OperatorSetParam memory operatorSetParams
    ) internal {
        _quorumParams[quorumNumber] = operatorSetParams;
        emit OperatorSetParamsUpdated(quorumNumber, operatorSetParams);
    }
```

Remediation:

We recommend to add a check to ensure **kickBIPsOfOperatorStake** is higher than 100% (**10_000**).

Commentary from the client:

“ This is up to the churn approver to properly gate this.”

EIGEN2-15 | Ejection timestamp is not updated when operator is kicked out in update loop

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

An operator could be kicked by ejection, which invokes the kick functions in **SlashingRegistryCoordinator.sol** It sets a **lastEjectionTimestamp** to the current block.timestamp, which disallows the same operator to register immediately again .

```
function ejectOperator(
    address operator,
    bytes memory quorumNumbers
) public virtual onlyEjector {
    lastEjectionTimestamp[operator] = block.timestamp;
    _kickOperator(operator, quorumNumbers);
}
```

However **_kickOperator** is also called when a operator gets kicked by someone registering with churn and during **updateOperators** that kicks operators that don't meet the minimum threshold.

```
function _updateOperatorsStakes(... ) internal virtual {
      -- snip
      if (doesNotMeetStakeThreshold[j]) {
          _kickOperator(operators[j], singleQuorumNumber);
      }
  }
}
```

It doesn't update the **lastEjectionTimestamp**, although they are kicked out by not having enough threshold.

```
function ejectOperator(
    address operator,
    bytes memory quorumNumbers
) public virtual onlyEjector {
    lastEjectionTimestamp[operator] = block.timestamp;
    _kickOperator(operator, quorumNumbers);
}
```

Remediation:

Consider updating the operator's **lastEjectionTimestamp** when they are kicked out for not having a minimum threshold, so they can't register immediately again.

Commentary from the client:

" Ejection has a different pathway from churning and we want a cooldown there."

EIGEN2-6 | Missing check for strategy modifier value

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

As specified in **StakeRegistry.md** a strategy's multiplier cant be set to 0,

```
* For each `_strategyParams` being added, the `multiplier` MUST NOT be 0
```

The function **_addStrategyParams** has the following check:

```
function _addStrategyParams(...) internal {
 - snip -
require(_strategyParams[i].multiplier > 0, InputMultiplierZero());
```

However, this is missing in the **modifyStrategyParams** function, so an active strategy's multiplier could be changed to 0 at later point in time.

```
function modifyStrategyParams(
    uint8 quorumNumber,
    uint256[] calldata strategyIndices,
    uint96[] calldata newMultipliers
) public virtual onlyCoordinatorOwner quorumExists(quorumNumber) {
    uint256 numStrats = strategyIndices.length;
    require(numStrats > 0, InputArrayLengthZero());
    require(newMultipliers.length == numStrats, InputArrayLengthMismatch());


    StrategyParams[] storage _strategyParams = strategyParams[quorumNumber];


    for (uint256 i = 0; i < numStrats; i++) {
        // Change the strategy's associated multiplier
        _strategyParams[strategyIndices[i]].multiplier = newMultipliers[i];
        emit StrategyMultiplierUpdated(
            quorumNumber, _strategyParams[strategyIndices[i]].strategy,
newMultipliers[i]
```

```
            );
        }
    }
```

Remediation:

Add the non zero check to **modifyStrategyParams**.

EIGEN2-7 | Unused imports

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Description:

We identified the following contracts/libraries that are imported but never used:

- LibMergeSort in ServiceManagerBase.sol

```
import {LibMergeSort} from "./libraries/LibMergeSort.sol";
```

Remediation:

Remove the redundant imports.

EIGEN2-11 | Quorum Total Stake Becomes Inaccurate After Changing minimumStakeForQuorum Without Updating Operators

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

StakeRegistery.sol:setMinimumStakeForQuorum

Description:

The **setMinimumStakeForQuorum** function updates the **minimumStakeForQuorum** value in **StakeRegistery** for a specific **quorumNumber**. However, it does not automatically update the state of the **_totalStakeHistory**, even though some operators may have become cut off in case of a higher minimum stake.

For example, if there are 100 operators total, and 10 of them have a minimum stake of 1 ETH, and the **minimumStakeForQuorum** is increased to 2 ETH, then it should only count the weight of the 90 valid operators. However, it doesn't update the total weight after calling **setMinimumStakeForQuorum** making it potentially out of sync.

```
/// @inheritdoc IStakeRegistry
function setMinimumStakeForQuorum(
    uint8 quorumNumber,
    uint96 minimumStake
) public virtual onlyCoordinatorOwner quorumExists(quorumNumber) {
    _setMinimumStakeForQuorum(quorumNumber, minimumStake);
}
```

Remediation:

After calling **setMinimumStakeForQuorum** it needs to update all operators through the function **updateOperatorsStake** to apply the new stake delta to the quorum's total stake without operators below the minimumStake threshold.

hexens x Eigen Layer