Layr-Labs

EigenLayer Slashing Review

Security Assessment Report

Version: 2.0

February, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Layr-Labs smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Layr-Labs smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Layr-Labs smart contracts in scope.

Overview

EigenLayer is a restaking platform where users delegate assets to operators, who secure off-chain processes via Actively Validated Services (`AVSs`). Current functionality includes operator/AVS registration, deregistration, and reward distribution.

The Q1 2025 protocol upgrade introduces slashing mechanisms to penalise misbehaviour, requiring major updates to core contracts and new slashing management systems. This security review focuses on the v1.0.0 release, prioritizing compatibility with existing mainnet contracts during the upgrade.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Layr-Labs/eigenlayer-contracts repository.

The scope of this time-boxed review was strictly limited to files at PR #679.

The fixes of the identified issues were assessed at commit f7413d9.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- High: 2 issues.

- Medium: 1 issue.
- Low: 6 issues.
- Informational: 10 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Layr-Labs smart contracts in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EGSL-01	Incorrect Withdrawable Shares Reduction After `AVS` And Beacon Chain Slashing	High	Closed
EGSL-02	Over-Slashing Of Withdrawable `beaconChainETHStrategy` Shares	High	Closed
EGSL-03	`beaconChainETHStrategy` Queued Withdrawals Excluded From Slashable Shares	Medium	Resolved
EGSL-04	Unsafe Casting in `_addInt128()`	Informational	Resolved
EGSL-05	Slashing A Pending Deallocation Impacts Deallocation Queue	Low	Resolved
EGSL-06	Incorrect Event Emission When Completing Queued Withdrawal As Shares	Low	Resolved
EGSL-07	`getMinimumSlashableStake()` Does not Account For Removed Strategies	Low	Closed
EGSL-08	Global Deallocation Delay May Not Be Appropriate For Different Operator Sets	Low	Closed
EGSL-09	Denial Of Service Due To Unbounded Allocation Delay	Low	Closed
EGSL-10	Operators Under Immediate Slashing Risk Upon Reactivation Of Previous Strategy	Low	Closed
EGSL-11	Incorrect `_addShares()` Can Lead to Over-Delegation After Slashing Upgrade	Informational	Resolved
EGSL-12	`beaconChainSlashingFactor` Is Negated After Delegation	Informational	Resolved
EGSL-13	Minimum Slashing Amount Can Result In No Token Burnt Due To Rounding	Informational	Resolved
EGSL-14	Lack of Whitelist Validation in `addStrategiesToOperatorSet()`	Informational	Closed
EGSL-15	Function Selector Based Permissions May Break During Contract Upgrades	Informational	Resolved
EGSL-16	Rounding Of `slashingFactor` Can Result In Complete Loss Of Withdrawable Shares	Informational	Resolved
EGSL-17	Completing Queued Withdrawal Can Result In Revert When `sharesToWithdraw` Is Zero	Informational	Resolved
EGSL-18	Unsafe Casting In `OperatorSetLib.decode()`	Informational	Closed
EGSL-19	Miscellaneous General Comments	Informational	Closed

EGSL-01	Incorrect Withdrawable Shares Reduction After AVS And Beacon Chain Slashing		
Asset	`EigenPodManager.sol, DelegationManager.sol, SlashingLib.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `beaconChainSlashingFactor` calculation fails to account for operator `maxMagnitude` and staker `depositScalingFactor`, leading to incorrect withdrawable share reductions when multiple slashing events occur. This allows stakers to withdraw more funds than should be available after combined operator and beacon chain slashing.

The core issue resides in `EigenPodManager._reduceSlashingFactor()` which calculates the beacon chain slashing factor (`bcsf`) using unscaled restaked balances rather than accounting for existing operator slashing (`maxMagnitude`) and deposit scaling factors (`dsf`). This results in an edgecase scenario where a user's withdrawable shares decreases less than the amount of ETH slashed on the beacon chain.

Consider the following example:

1. Stake and verify a validator (`withdrawableShares` = 32 ETH)
2. Slash operator for 50% of allocation (`withdrawableShares` = 16 ETH, `maxMagnitude` = 0.5)
3. Slash validator for 16 ETH
4. Start and finalize an `EigenPod` checkpoint (`withdrawableShares` = 8 ETH, `bcsf` = 0.5)

where:

$$\begin{aligned} finalWithdrawableShares &= depositShares \times dsf \times slashingFactor \\ &= depositShares \times dsf \times (maxMagnitude \times bcsf) \\ &= 32 \times 1 \times (0.5 \times 0.5) \\ &= 8 \end{aligned}$$

In the example above, the staker's `withdrawableShares` only decreases by 8 ETH (from 16 to 8) from step 2 to 4 even though they have been slashed on the beacon chain for 16 ETH, since the calculation of `withdrawableShares` is multiplicative based on the `maxMagnitude` and `bcsf`. Since the `bcsf` calculation does not take into account `maxMagnitude`, 8 ETH has effectively been slashed twice.

Recommendations

Modify the beacon chain slashing factor calculation to scale `prevRestakedBalanceWei` by `maxMagnitude`.

To also incorporate the recommended change in EGSL-02, scale `prevRestakedBalanceWei` by `maxMagnitude * dsf`:

EigenPodManager.sol::_reduceSlashingFactor()

```
prevRestakedBalanceWei = prevRestakedBalanceWei.mulWad(maxMagnitude).mulWad(dsf);
```

Resolution

The EigenLayer team has acknowledged this issue in a response available as a Notion document.

A summary of the response is as follows:

> *"We define restaking as reusing staked ETH as security for* `AVSs`*. Thus, the same Native ETH that is securing the beacon chain can also be slashed by an* `AVS`*, with priority burning rights going to the beacon chain. Hence, the behavior described in this issue is an intentional design choice."*

The EigenLayer team will also update documentation to inform users of the accounting intricacies of the system.

EGSL-02	Over-Slashing Of Withdrawable `beaconChainETHStrategy` Shares		
Asset	`EigenPodManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `_reduceSlashingFactor()` function in `EigenPodManager` incorrectly calculates the new beacon chain slashing factor by using unscaled balances, leading to inconsistent withdrawable share calculations depending on when validators are verified relative to checkpoints.

EigenPodManager.sol::_reduceSlashingFactor()

```
function _reduceSlashingFactor(
    address podOwner,
    uint256 prevRestakedBalanceWei,
    uint256 balanceDecreasedWei
) internal returns (uint64) {
    uint256 newRestakedBalanceWei = prevRestakedBalanceWei - balanceDecreasedWei;
    uint64 prevBeaconSlashingFactor = beaconChainSlashingFactor(podOwner);
    // newBeaconSlashingFactor is less than prevBeaconSlashingFactor because
    // newRestakedBalanceWei < prevRestakedBalanceWei
    uint64 newBeaconSlashingFactor =
        uint64(prevBeaconSlashingFactor.mulDiv(newRestakedBalanceWei, prevRestakedBalanceWei));
    uint64 beaconChainSlashingFactorDecrease = prevBeaconSlashingFactor - newBeaconSlashingFactor;
    _beaconChainSlashingFactor[podOwner] =
        BeaconChainSlashingFactor({slashingFactor: newBeaconSlashingFactor, isSet: true});
    emit BeaconChainSlashingFactorDecreased(podOwner, prevBeaconSlashingFactor, newBeaconSlashingFactor);
    return beaconChainSlashingFactorDecrease;
}
```

The `_reduceSlashingFactor()` function calculates the new beacon chain slashing factor (`bcsf`) without scaling `prevRestakedBalanceWei` by the deposit scaling factor (`dsf`). This causes the slashing factor to be reduced too aggressively when a validator is slashed.

Consider the following scenario where the 2nd validator is verified before the checkpoint is performed:

1. Stake and verify the first validator (`withdrawableShares` = 32 ETH)
2. Slash operator for 50% of allocation (`withdrawableShares` = 16 ETH, `maxMagnitude` = 0.5)
3. Stake and verify the second validator (`withdrawableShares` = 48 ETH, `dsf` = 1.5)
4. Slash the first validator for 16 ETH on the beacon chain
5. Perform an `EigenPod` checkpoint (`withdrawableShares` = 36 ETH, `bcsf` = 0.75)

where:

$$\begin{aligned} dsf &= \frac{newWithdrawableShares}{newDepositShares \times slashingFactor_{steptwo}} \\ &= \frac{48}{64 \times (0.5 \times 1)} \\ &= 1.5 \end{aligned}$$

$$\begin{aligned} finalWithdrawableShares &= depositShares \times dsf \times slashingFactor \\ &= 64 \times 1.5 \times (0.5 \times 0.75) \\ &= 36 \end{aligned}$$

The final withdrawable shares value is incorrectly calculated as 36 ETH, whereas the correct value is 40 ETH:

$$\begin{aligned} expectedFinalWithdrawableShares &= withdrawableShares_{validatorOne} + withdrawableShares_{validatorTwo} \\ &= (32 \times 0.5 \times 0.5) + 32 \\ &= 8 + 32 = 40 \end{aligned}$$

Recommendations

Modify `_reduceSlashingFactor()` to scale `prevRestakedBalanceWei` by the deposit scaling factor before calculating the new beacon chain slashing factor.

To also incorporate the recommended change in EGSL-01, scale `prevRestakedBalanceWei` by `maxMagnitude * dsf`:

EigenPodManager.sol::_reduceSlashingFactor()

```
prevRestakedBalanceWei = prevRestakedBalanceWei.mulWad(maxMagnitude).mulWad(dsf);
```

Resolution

The EigenLayer team has acknowledged this issue in a response available as a Notion document.

A summary of the response is as follows:

> *"Following the logic from EGSL-01, the attributable slashed amount when an `AVS` slashes decreases in the event of beacon chain slashing. The difference in end state between checkpointing before or after staking a validator is due to the asynchronous nature of the beacon chain proof system. A benefit of the system is that stakers are incentivized to immediately prove beacon chain slashes."*

The EigenLayer team will also update documentation to inform users of this specific edge case.

EGSL-03	`beaconChainETHStrategy` Queued Withdrawals Excluded From Slashable Shares		
Asset	`DelegationManager.sol, EigenPodManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

`DelegationManager` excludes `beaconChainETHStrategy` shares in queued withdrawals when calculating burnable shares. This results in undercounting of burnable shares during operator slashing events.

The `_addQueuedSlashableShares()` function explicitly excludes `beaconChainETHStrategy` from cumulative scaled shares tracking as shown below:

DelegationManager.sol::_addQueuedSlashableShares()

```
/// @dev Add to the cumulative withdrawn scaled shares from an operator for a given strategy
function _addQueuedSlashableShares(address operator, IStrategy strategy, uint256 scaledShares) internal {
    // @audit beaconChainETHStrategy is excluded from slashable shares tracking
    if (strategy != beaconChainETHStrategy) {
        uint256 currCumulativeScaledShares = _cumulativeScaledSharesHistory[operator][strategy].latest();
        _cumulativeScaledSharesHistory[operator][strategy].push({
            key: uint32(block.number),
            value: currCumulativeScaledShares + scaledShares
        });
    }
}
```

However, the `slashOperatorShares()` function relies on this tracking to calculate slashable shares in the withdrawal queue through `_getSlashableSharesInQueue()`:

DelegationManager.sol::_getSlashableSharesInQueue()

```
// We want ALL shares added to the withdrawal queue in the window [block.number - MIN_WITHDRAWAL_DELAY_BLOCKS, block.number]
//
// To get this, we take the current shares in the withdrawal queue and subtract the number of shares
// that were in the queue before MIN_WITHDRAWAL_DELAY_BLOCKS.
uint256 curQueuedScaledShares = _cumulativeScaledSharesHistory[operator][strategy].latest();
uint256 prevQueuedScaledShares = _cumulativeScaledSharesHistory[operator][strategy].upperLookup({
    key: uint32(block.number) - MIN_WITHDRAWAL_DELAY_BLOCKS - 1
});

// The difference between these values is the number of scaled shares that entered the withdrawal queue
// less than or equal to MIN_WITHDRAWAL_DELAY_BLOCKS ago. These shares are still slashable.
uint256 scaledSharesAdded = curQueuedScaledShares - prevQueuedScaledShares;

return SlashingLib.scaleForBurning({
    scaledShares: scaledSharesAdded,
    prevMaxMagnitude: prevMaxMagnitude,
    newMaxMagnitude: newMaxMagnitude
});
```

This omission means that when operators are slashed on the `beaconChainETHStrategy`:

1. Queued beacon chain ETH withdrawals are not included in `scaledSharesAdded` calculation.
2. `totalDepositSharesToBurn` will be undercounted.
3. Less shares will be burnable in `EigenPodManager` than intended.

This issue has a low impact as `beaconChainETHStrategy` currently does not have a mechanism for burning ETH. However, it is still important to correctly account for burnable ETH shares so that this feature can be correctly implemented in the future after the Ethereum Pectra upgrade.

Recommendations

Remove the `beaconChainETHStrategy` exclusion in `_addQueuedSlashableShares()`.

Additionally, consider implementing proper slashing factor handling for beacon chain ETH in `EigenPodManager` to account for any beacon chain slashing that would result in less burnable shares.

Resolution

The EigenLayer team has removed the `beaconChainETHStrategy` exclusion in `_addQueuedSlashableShares()`.

This issue has been resolved in PR #1087.

EGSL-04	Unsafe Casting in `_addInt128()`
Asset	`AllocationManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_addInt128()` function can silently overflow due to unsafe casting, resulting in an attacker being able to allocate more magnitude than intended.

The `_addInt128()` function can overflow due to unsafe casting from `int128` to `uint128` when the casted value is negative, and from `uint128` to `uint64`.

AllocationManager.sol::_addInt128()

```
function _addInt128(uint64 a, int128 b) internal pure returns (uint64) {
    return uint64(uint128(int128(uint128(a)) + b));
}
```

This allows an attacker to manipulate their `currentMagnitude` to be out of bounds and higher than their `maxMagnitude`.

This actual vulnerability is rated high impact and likelihood as it allows the `operator` to allocate without limits. However, this issue is rated as informational severity in the report, as it was made aware to the testing team by the EigenLayer team during the engagement.

Recommendations

Consider using safe casting and reverting in the case of an underflow in `_addInt128()`.

Resolution

The EigenLayer team has used the `SafeCastUpgradeable` library to ensure that the `_addInt128()` function does not overflow when casting to `uint64`.

This issue has been resolved in PR #1027.

EGSL-05	Slashing A Pending Deallocation Impacts Deallocation Queue		
Asset	`AllocationManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When a pending deallocation is slashed the value of the deallocation may round down to zero. Therefore, an operator is able to call `modifyAllocations()` to update this deallocation. As a result, the `deallocationQueue` may no longer be sorted based on `effectBlock`.

The root cause of the issue is that when `slashOperator()` is called, if there is a pending deallocation it will be slashed. That slashed amount may result in the `allocation.pendingDiff` rounding down to zero. This will not clear the pending deallocation from the `deallocationQueue`.

AllocationManager.sol::_slashOperator()

```
if (allocation.pendingDiff < 0) {
    uint64 slashedPending =
        uint64(uint256(uint128(-allocation.pendingDiff)).mulWadRoundUp(params.wadsToSlash[i]));
    allocation.pendingDiff += int128(uint128(slashedPending)); // @audit may be zeroed

    // ...
}
```

Once `allocation.pendingDiff = 0`, the pending deallocation can be modified as the following check now passes:

AllocationManager.sol::modifyAllocations()

```
function modifyAllocations(
    address operator,
    AllocateParams[] memory params
) external onlyWhenNotPaused(PAUSED_MODIFY_ALLOCATIONS) {
    // ...
    for (uint256 i = 0; i < params.length; i++) {
        // ...
        for (uint256 j = 0; j < params[i].strategies.length; j++) {
            (StrategyInfo memory info, Allocation memory allocation) =
                    _getUpdatedAllocation(operator, operatorSet.key(), strategy);
            // @audit modification no longer counts as pending even though effectBlock has not passed
            require(allocation.pendingDiff == 0, ModificationAlreadyPending());
            // ...
        }
    }
}
```

`allocation.pendingDiff == 0` incorrectly assumes that the allocation has no pending modifications, even though the `allocation.effectBlock` has not passed. Once this allocation is modified, the `allocation.effectBlock` will be set again, resulting in the `deallocationQueue` being unsorted.

An unsorted `deallocationQueue` will result in the `_clearDeallocationQeue()` function not clearing all of the deallocations in the queue. `getAllocatableMagnitude()` will also incorrectly calculate the return value as it discounts deallocations.

Recommendations

Consider changing the function `modifyAllocations()` to check for pending modifications based on `allocation.effectBlock` instead of `allocation.pendingDiff`.

Resolution

The EigenLayer team has implemented the recommended fix above.

This issue has been resolved in PR #1052.

EGSL-06	Incorrect Event Emission When Completing Queued Withdrawal As Shares		
Asset	`DelegationManager.sol, StrategyManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `StrategyManager.addShares()` function does not validate whether the `token` input parameter corresponds to the strategy's underlying token. This can result in an incorrect event emmision.

When completing a queued withdrawal with `receiveAsTokens = false`, `StrategyManager.addShares()` is called for non-EigenPod strategies and the user-inputted `token` is passed to `_addShares()` without verifying that it matches the strategy's underlying token.

StrategyManager.sol::addShares()

```
function addShares(
  address staker,
  IStrategy strategy,
  IERC20 token,
  uint256 shares
) external onlyDelegationManager returns (uint256, uint256) {
  // @audit no input validation for token
  return _addShares(staker, token, strategy, shares);
}
```

This allows an attacker to spoof the `token` parameter in the `Deposit()` event emitted by `StrategyManager._addShares()`.

This issue is classified as low impact because it does not have a direct onchain impact. Front-ends or indexing services that rely on emitted events may display or parse incorrect data.

Recommendations

Consider adding input validation to `token` in `StrategyManager.addShares()` to ensure that each token is correctly aligned with its corresponding strategy.

Resolution

The `token` parameter has been removed from the `Deposit()` event and the `addShares()` and `_addShares()` functions.

This issue has been resolved in PR #1013.

EGSL-07	`getMinimumSlashableStake()` Does not Account For Removed Strategies		
Asset	`AllocationManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `getMinimumSlashableStake()` function in the `AllocationManager.sol` does not validate whether a given strategy is currently part of the operator set, potentially returning inaccurate values if strategies are removed from the set after initial inclusion.

Operator sets are dynamic entities that can have strategies added or removed over time by `AVSs`. The `removeStrategiesFromOperatorSet()` function allows for the removal of strategies from an operator set. However, the `getMinimumSlashableStake()` function does not account for these potential changes when calculating the minimum slashable stake for an operator.

`getMinimumSlashableStake()` iterates through all strategies provided as input, regardless of whether they are still part of the active operator set. This approach can lead to the inclusion of strategies which are no longer part of the active operator set in the calculation, resulting in an inflated or otherwise incorrect minimum slashable stake value.

AllocationManager.sol::getMinimumSlashableStake()

```
for (uint256 j = 0; j < strategies.length; j++) {
    // @audit no check for strategy being part of operator set
    IStrategy strategy = strategies[j];

    // Fetch the max magnitude and allocation for the operator/strategy.
    // Prevent division by 0 if needed. This mirrors the "FullySlashed" checks
    // in the DelegationManager
    uint64 maxMagnitude = _maxMagnitudeHistory[operator][strategy].latest();
    if (maxMagnitude == 0) {
        continue;
    }

    Allocation memory alloc = getAllocation(operator, operatorSet, strategy);
    // ...
    slashableStake[i][j] = delegatedStake[i][j].mulWad(slashableProportion);
}
```

Once a strategy is removed from an operator set, the operator is no longer slashable for that strategy. Therefore it is misleading to include the slashable stake as part of `getMinimumSlashableStake()`

Recommendations

The `getMinimumSlashableStake()` function should be modified to only consider strategies that are currently part of the active operator set. This can be achieved by validating the strategy is part of operator set, similar to the validation performed in `slashOperator()`, and reverting with the error `StrategyNotInOperatorSet()`.

Resolution

The EigenLayer team has acknowledged this issue with the following comment:

> *"`AVSs` are the primary user of this method. They are expected to know which strategies are configured to be a part of the operator set."*

EGSL-08	Global Deallocation Delay May Not Be Appropriate For Different Operator Sets		
Asset	`AllocationManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current implementation of a global `DEALLOCATION_DELAY` in the `AllocationManager` may not adequately address the diverse security requirements of different operator sets.

The EigenLayer slashing upgrade introduces the concept of operator sets, which are logical groupings of operators created by `AVSs` for various tasks and security requirements. Each operator set can have unique slashing conditions and security needs. For instance, an operator set with high-risk, computationally intensive tasks might require a longer deallocation delay compared to a set with simpler, low-risk tasks.

However, the current implementation uses a single, global `DEALLOCATION_DELAY` immutable constant for all operator sets in `AllocationManager`. This may not be appropriate given the varying risk profiles and operational requirements of different sets.

AllocationManagerStorage.sol

```
/// @notice Delay before deallocations are clearable and can be added back into freeMagnitude
/// In this window, deallocations still remain slashable by the operatorSet they were allocated to.
uint32 public immutable DEALLOCATION_DELAY;
```

Recommendations

Consider implementing a unique `DEALLOCATION_DELAY` for each operator set to allow for more granular control over security parameters. This can be achieved by extending the `OperatorSet` struct to include a custom `deallocationDelay` field.

Resolution

The EigenLayer team has acknowledged this issue with the following comment:

> *"We agree that a global deallocation delay won't work for all use-cases. This is a known limitation we'd like to address in a future release."*

EGSL-09	Denial Of Service Due To Unbounded Allocation Delay		
Asset	`AllocationManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `setAllocationDelay()` function does not enforce a maximum delay value, allowing operators to set an extremely long delay that prevent any allocations from being set.

AllocationManager.sol::_setAllocationDelay()

```
function _setAllocationDelay(address operator, uint32 delay) internal {
    AllocationDelayInfo memory info = _allocationDelayInfo[operator];

    // If there is a pending delay that can be applied now, set it
    if (info.effectBlock != 0 && block.number >= info.effectBlock) {
        info.delay = info.pendingDelay;
        info.isSet = true;
    }

    info.pendingDelay = delay;
    info.effectBlock = uint32(block.number) + ALLOCATION_CONFIGURATION_DELAY + 1;

    _allocationDelayInfo[operator] = info;
    emit AllocationDelaySet(operator, delay, info.effectBlock);
}
```

An operator could set an extremely large delay (up to `MAX_UINT32` blocks, or 1634 years), which would cause the `allocation.effectBlock` calculation in `modifyAllocations()` to overflow:

AllocationManager.sol::modifyAllocations()

```
else if (allocation.pendingDiff > 0) {
    // ...

    // @audit allocation.effectBlock is uint32, so this can revert from overflow if the delay is too large
    allocation.effectBlock = uint32(block.number) + operatorAllocationDelay;
}
```

This would cause all allocation increases to the operator to revert from an overflow.

Recommendations

Add an upper bound check in the `setAllocationDelay()` function to prevent unreasonably large delays.

Resolution

The EigenLayer team has acknowledged this issue with the following comment:

> *"We've deemed adding a reasonable upper bound to be more than it's worth. We expect operators to be responsible about the way they modify their allocations."*

EGSL-10	Operators Under Immediate Slashing Risk Upon Reactivation Of Previous Strategy		
Asset	`AllocationManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current implementation of strategy management in operator sets may expose operators to immediate slashing risk when strategies are removed and subsequently re-added.

In the `AllocationManager`, strategies can be added to or removed from operator sets dynamically. Operators allocate their stake to these strategies, and these allocations are subject to slashing conditions. The protocol implements a delay for new allocations to become effective:

AllocationManager.sol::modifyAllocations()

```
allocation.effectBlock = uint32(block.number) + operatorAllocationDelay;
```

However, when a strategy is removed from an operator set and later re-added, the existing allocations associated with that strategy become immediately slashable upon reactivation. This is because operators maintain historical allocation records even after strategy removal.

The standard allocation delay only applies to new allocations, so this behavior upon reactivation contrasts with the delay applied to new allocations and may not align with operators' expectations, potentially resulting in unfair penalties for operators who may not have had sufficient time to adjust their allocations after a strategy's reactivation.

Recommendations

Consider a delay mechanism for strategy activation when adding or re-adding strategies to operator sets.

Resolution

The EigenLayer team has acknowledged this issue with the following comment:

> *"We're aware of this, but feel adding yet another delay mechanism is more trouble than it's worth. This is documented as expected behavior in our contract docs."*

EGSL-11	Incorrect `_addShares()` Can Lead to Over-Delegation After Slashing Upgrade
Asset	`EigenPodManager.sol, DelegationManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_addShares()` function in `EigenPodManager` does not take into account negative initial share balances when returning the number of `shares` added. This results in the operator being credited with more shares than they are entitled to and the `depositScalingFactor` being updated incorrectly.

Before the slashing upgrade, `podOwnerDepositShares` can become negative after a checkpoint if there is a queued withdrawal. After the upgrade, the negative shares are reconciled when completing the queued withdrawal by adding the withdrawable shares of the withdrawal back to `podOwnerDepositShares` to make it non-negative. However, when the queued withdrawal is completed with `receiveAsTokens = false`, `EigenPodManager._addShares()` returns `shares` without accounting for any initial negative shares:

EigenPodManager._addShares()

```
function _addShares(address staker, uint256 shares) internal returns (uint256, uint256) {
    require(staker != address(0), InputAddressZero());
    require(int256(shares) >= 0, SharesNegative());

    int256 sharesToAdd = int256(shares);
    int256 prevDepositShares = podOwnerDepositShares[staker];
    int256 updatedDepositShares = prevDepositShares + sharesToAdd;
    podOwnerDepositShares[staker] = updatedDepositShares;

    emit PodSharesUpdated(staker, sharesToAdd);
    emit NewTotalShares(staker, updatedDepositShares);

    if (updatedDepositShares <= 0) {
        return (0, 0);
    }

    // @audit `shares` is returned even if `prevDepositShares < 0'
    //        resulting in an incorrect number of added shares
    return (prevDepositShares < 0 ? 0 : uint256(prevDepositShares), shares);
}
```

The returned `shares` value is used to increase the operator's shares and the staker's `depositScalingFactor` for the strategy:

DelegationManager._completeQueuedWithdrawal()

```
// Award shares back in StrategyManager/EigenPodManager.
(uint256 prevDepositShares, uint256 addedShares) = shareManager.addShares({
    staker: withdrawal.staker,
    strategy: withdrawal.strategies[i],
    token: tokens[i],
    shares: sharesToWithdraw
});

// Update the staker's deposit scaling factor and delegate shares to their operator
_increaseDelegation({
    operator: newOperator,
    staker: withdrawal.staker,
    strategy: withdrawal.strategies[i],
    prevDepositShares: prevDepositShares,
    addedShares: addedShares,
    slashingFactor: newSlashingFactors[i]
});
```

DelegationManager._increaseDelegation()

```
  function _increaseDelegation(
    address operator,
    address staker,
    IStrategy strategy,
    uint256 prevDepositShares,
    uint256 addedShares,
    uint256 slashingFactor
) internal {
    // Ensure that the operator has not been fully slashed for a strategy
    // and that the staker has not been fully slashed if it is the beaconChainStrategy
    // This is to prevent a divWad by 0 when updating the depositScalingFactor
    require(slashingFactor != 0, FullySlashed());

    // Update the staker's depositScalingFactor. This only results in an update
    // if the slashing factor has changed for this strategy.
    DepositScalingFactor storage dsf = _depositScalingFactor[staker][strategy];
    // @audit the depositScalingFactor is updated with the incorrect number of shares
    dsf.update(prevDepositShares, addedShares, slashingFactor);
    emit DepositScalingFactorUpdated(staker, strategy, dsf.scalingFactor());

    // If the staker is delegated to an operator, update the operator's shares
    if (isDelegated(staker)) {
        // @audit the operator's shares are increased by the incorrect number of shares
        operatorShares[operator][strategy] += addedShares;
        emit OperatorSharesIncreased(operator, staker, strategy, addedShares);
    }
}
```

The actual issue has a high impact as it results in the operator being credited with more shares than they are entitled to. In the case of a `depositScalingFactor` update, which would occur if the `slashingFactor` has changed, then the operator would end up with more withdrawable shares than they are entitled to.

This issue has an informational severity rating in the report as it was discovered by the EigenLayer team during the engagement.

Recommendations

Consider returning `updatedDepositShares` instead of `shares` from `EigenPodManager._addShares()` if `prevDepositShares < 0`.

Resolution

`EigenPodManager._addShares()` has been updated to return `updatedDepositShares` instead of `shares` if `prevDepositShares < 0` as recommended above.

This issue has been resolved in PR #1033.

EGSL-12	`beaconChainSlashingFactor` Is Negated After Delegation
Asset	`DelegationManager.sol, EigenPodManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_increaseDelegation()` function incorrectly assumes that the `addedShares` parameter corresponds to the number of added withdrawable shares. This leads to an incorrect deposit scaling factor (`dsf`) and operator shares update when the staker has been slashed on the beacon chain before delegating.

When a staker delegates to an operator, their `depositedShares` are used as the `addedShares` parameter in the `_increaseDelegation()` function:

DelegationManager.sol::_delegate()

```
function _delegate(address staker, address operator) internal onlyWhenNotPaused(PAUSED_NEW_DELEGATION) {
    // ...

    // read staker's deposited shares and strategies to add to operator's shares
    // and also update the staker depositScalingFactor for each strategy
    (IStrategy[] memory strategies, uint256[] memory depositedShares) = getDepositedShares(staker);
    uint256[] memory slashingFactors = _getSlashingFactors(staker, operator, strategies);

    for (uint256 i = 0; i < strategies.length; ++i) {
        // forgefmt: disable-next-item
        _increaseDelegation({
            operator: operator,
            staker: staker,
            strategy: strategies[i],
            prevDepositShares: uint256(0),
            // @audit incorrectly assumes depositedShares = withdrawableShares
            addedShares: depositedShares[i],
            slashingFactor: slashingFactors[i]
        });
    }
}
```

The `_increaseDelegation()` function assumes that `addedShares` corresponds to withdrawable shares, which is the case for token strategies as no slashing has occurred. However, for `beaconChainETHStrategy` , it is possible for beacon chain slashing to occur before the staker is delegated, such that the assumption no longer holds true. This results in the operator being credited with more shares than they are entitled to.

DelegationManager.sol::_increaseDelegation()

```
function _increaseDelegation(
    address operator,
    address staker,
    IStrategy strategy,
    uint256 prevDepositShares,
    uint256 addedShares, // @audit `addedShares` corresponds to an increase in withdrawable shares
    uint256 slashingFactor
) internal {
    // Ensure that the operator has not been fully slashed for a strategy
    // and that the staker has not been fully slashed if it is the beaconChainStrategy
    // This is to prevent a divWad by 0 when updating the depositScalingFactor
    require(slashingFactor != 0, FullySlashed());

    // Update the staker's depositScalingFactor. This only results in an update
    // if the slashing factor has changed for this strategy.
    DepositScalingFactor storage dsf = _depositScalingFactor[staker][strategy];
    // @audit dsf is incorrectly updated with `addedShares`
    dsf.update(prevDepositShares, addedShares, slashingFactor);
    emit DepositScalingFactorUpdated(staker, strategy, dsf.scalingFactor());

    // If the staker is delegated to an operator, update the operator's shares
    if (isDelegated(staker)) {
        // @audit operator shares is incorrectly increased by `addedShares`
        operatorShares[operator][strategy] += addedShares;
        emit OperatorSharesIncreased(operator, staker, strategy, addedShares);
    }
}
```

Furthermore, a staker that has been slashed on the beacon chain before delegation will be able to negate any slashing events prior to delegation. This is because the `dsf` is reset to `1 / slashingFactor` on the initial deposit/delegation:

SlashingLib.sol::update()

```
function update(
    DepositScalingFactor storage dsf,
    uint256 prevDepositShares,
    uint256 addedShares,
    uint256 slashingFactor
) internal {
    // If this is the staker's first deposit, set the scaling factor to
    // the inverse of slashingFactor
    if (prevDepositShares == 0) {
        // @audit this cancels out the slashingFactor, so any slashing events prior to delegation are negated
        dsf._scalingFactor = uint256(WAD).divWad(slashingFactor);
        return;
    }
    // ...
}
```

The actual issue has a high impact as it allows a staker to negate changes to their `beaconChainSlashingFactor` before delegating, and allows them to have more shares delegated to the operator than they are entitled to.

This issue has an informational severity rating in the report as it was discovered by the EigenLayer team during the engagement.

Recommendations

Consider implmenting the following:

1. Scale `addedShares` by the `beaconChainSlashingFactor` for the `beaconChainETHStrategy` in the `_delegate()` function before calling `_increaseDelegation()`.

2. For the initial deposit/delegation, assign the `dsf` to `1 / maxMagnitude` instead of `1 / slashingFactor`.

Resolution

The EigenLayer team has implemented the following:

1. Use `withdrawableShares` instead of `depositedShares` as the `addedShares` argument in `_increaseDelegation()`.
2. For the initial deposit/delegation, scale the current `dsf` by `1 / operatorSlashingFactor` to selectively negate AVS slashing but maintain beacon chain slashing.

This issue has been resolved in PR #1045.

EGSL-13	Minimum Slashing Amount Can Result In No Token Burnt Due To Rounding
Asset	`AllocationManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When the `slashOperator()` function is called with a small `wadToSlash` value, such as `1`, rounding can result in no tokens being added to `burnableShares[strategy]`, even though the slashing still effectively takes place through reductions in the operator's maximum and encumbered magnitude.

The occurs because the calculation of slashed tokens uses round-down division when converting from magnitude to shares, which can result in zero tokens being marked for burning in cases with small slashing amounts.

While this does not negatively impact other stakers or operators, it means the token burning mechanism can be circumvented in certain cases, leaving tokens permanently locked in the contract rather than being burned as intended.

Recommendations

Consider adding a note in the documentation that small slashing amounts may not result in actual token burns due to rounding, which will result in small amounts of tokens locked in the contract rather than fully burnable through the burn mechanism.

Resolution

The Eigenlayer team has added a Natspec comment for the `slashOperator()` function to inform users of this particular edge case.

This issue has been resolved in PR #1088.

EGSL-14	Lack of Whitelist Validation in `addStrategiesToOperatorSet()`
Asset	`AllocationManager.sol StrategyFactory.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `addStrategiesToOperatorSet()` function does not validate whether the `strategies` being added to an operator set are whitelisted on `StrategyManager`. This allows `AVSs` to add arbitrary strategies to an operator set.

AllocationManager.sol::addStrategiesToOperatorSet()

```
function addStrategiesToOperatorSet(
    address avs,
    uint32 operatorSetId,
    IStrategy[] calldata strategies
) external checkCanCall(avs) {
    OperatorSet memory operatorSet = OperatorSet(avs, operatorSetId);
    bytes32 operatorSetKey = operatorSet.key();

    require(_operatorSets[avs].contains(operatorSet.id), InvalidOperatorSet());

    for (uint256 i = 0; i < strategies.length; i++) {
        // @audit no validation of whether the strategy is whitelisted
        require(_operatorSetStrategies[operatorSetKey].add(address(strategies[i])), StrategyAlreadyInOperatorSet());
        emit StrategyAddedToOperatorSet(operatorSet, strategies[i]);
    }
}
```

Recommendations

Consider validating that each strategy has been approved by `StrategyManager` to prevent arbitrary `strategies` from being added to an operator set.

Resolution

The Eigenlayer team has acknowledged the issue with the following comment:

> *"Whitelisting is primarily concerned with deposits on the `StrategyManager` side. While this finding is valid, we feel it's more trouble than it's worth to add this check to the `AllocationManager`."*

EGSL-15	Function Selector Based Permissions May Break During Contract Upgrades
Asset	`PermissionController.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`PermissionController` uses function selectors to determine if a caller has permission to execute a specific function. This functionality may not be compatible with contract upgrades, as the function selector will change when the function parameters are modified. This will cause existing permissions to break, as the stored selector in `_permissions[account].appointeePermissions[caller]` will no longer match the new function selector.

PermissionController.sol::canCall()

```
function canCall(address account, address caller, address target, bytes4 selector) external view returns (bool) {
    return isAdmin(account, caller)
        || _permissions[account].appointeePermissions[caller].contains(_encodeTargetSelector(target, selector));
}
```

Recommendations

Instead of using function selectors, consider using the hash of just the function name for permission checks. This would make permissions resilient to parameter changes in contract upgrades.

Resolution

The EigenLayer team has opted to not implement the fix above and instead added documentation to inform users of this issue.

The relevant documentation has been added in PR #1096.

EGSL-16	Rounding Of `slashingFactor` Can Result In Complete Loss Of Withdrawable Shares
Asset	`DelegationManager.sol EigenPodManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Due to rounding, it is possible for the `slashingFactor` to be rounded down to zero, even when both `operatorMaxMagnitude` and `beaconChainSlashingFactor` are non-zero. This can result in incorrect `withdrawableShares` being calculated for the `beaconChainETHStrategy`.

The `slashingFactor` for the `beaconChainETHStrategy` is calculated as:

DelegationManager.sol::_getSlashingFactor()

```
if (strategy == beaconChainETHStrategy) {
    uint64 beaconChainSlashingFactor = eigenPodManager.beaconChainSlashingFactor(staker);
    return operatorMaxMagnitude.mulWad(beaconChainSlashingFactor);
}
```

Due to the rounding of the `mulWad()` function, it is possible for the `slashingFactor` to be rounded down to 0, even when both `operatorMaxMagnitude` and `beaconChainSlashingFactor` are non-zero.

An operator can exploit this rounding to burn their stakers' withdrawable shares by doing the following:

1. The operator creates a malicious `AVS` and registers and allocates to it.
2. The operator slashes themself through their `AVS` by a large amount such that their `maxMagnitude` is reduced to a very small value (e.g. 1).
3. A staker verifies a validator on their `EigenPod` and delegates to the operator.
4. The staker gets penalised on the beacon chain by a very small amount (e.g. 1 gwei)

Once the staker's `beaconChainSlashingFactor` is reduced below `WAD` in step 4, their `slashingFactor` becomes 0 and they lose all their withdrawable shares.

Though the impact of this issue can be extremely severe, this issue has an informational severity rating as it is very unlikely that an operator would grief their stakers without any potential upside. Furthermore, it is assumed that stakers will perform due diligence on the operators they delegate to.

Recommendations

Consider informing users on this particular edge case in the documentation. Furthermore, provide plenty of warning to stakers on the front-end when operators may be considered malicious and have an extremely low `maxMagnitude`.

Resolution

The EigenLayer team has added a Natspec comment to the `_getSlashingFactor()` function to inform users of this particular edge case.

This issue has been resolved in PR #1089.

EGSL-17	Completing Queued Withdrawal Can Result In Revert When `sharesToWithdraw` Is Zero
Asset	`DelegationManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_completeQueuedWithdrawal()` function does not handle scenarios where `sharesToWithdraw = 0`, which can cause issues with `ERC-20` tokens that do not support zero-value transfers.

When `receiveAsTokens = true`, the `StrategyManager.withdrawSharesAsTokens()` function is called to withdraw tokens from the strategy. This poses a problem for `ERC-20` implementations that explicitly prohibit zero-value transfers, as
`sharesToWithdraw = 0` in cases where the operator is fully slashed during the withdrawal delay period.

DelegationManager.sol::_completeQueuedWithdrawal()

```
// @audit this is zero when prevSlashingFactors[i] is zero
uint256 sharesToWithdraw = SlashingLib.scaleForCompleteWithdrawal({
    scaledShares: withdrawal.scaledShares[i],
    slashingFactor: prevSlashingFactors[i]
});
if (receiveAsTokens) {
    // @audit this will revert for tokens that do not support zero-value transfers
    //        if sharesToWithdraw = 0
    shareManager.withdrawSharesAsTokens({
        staker: withdrawal.staker,
        strategy: withdrawal.strategies[i],
        token: tokens[i],
        shares: sharesToWithdraw
    });
}
```

When `receiveAsTokens = false`, the `StrategyManager.addShares()` function is called to add shares back to the strategy. However, the `StrategyManager._addShares()` function does not allow for zero-value `shares` to be added, and hence will also revert.

The actual impact is rated high, as it would cause the staker's tokens in other strategies associated with the queued withdrawal to be permanently stuck. However, the likelihood of this issue is rated low, as it requires an `ERC-20` token that does not support zero-value transfers and the `operator` to become fully slashed during the cooldown period.

This issue is rated as informational severity in the report, as it was discovered by the EigenLayer team during the engagement.

Recommendations

Consider adding a check to ensure `sharesToWithdraw != 0` before performing the withdraw. If `sharesToWithdraw = 0`, the withdrawal should be completed without withdrawing any tokens.

Resolution

The EigenLayer team has added a check to skip calling `SharesManager` if `sharesToWithdraw = 0` as recommended above.

This issue has been resolved in PR #1019.

EGSL-18	Unsafe Casting In `OperatorSetLib.decode()`
Asset	`OperatorSetLib.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `decode()` will take a 32-byte `_key` as input and decode it to a 20-byte `avs` address and 3-byte `id`. The remaining bytes are unused and may be arbitrarily set.

OperatorSetLib.sol::decode()

```
function decode(
    bytes32 _key
) internal pure returns (OperatorSet memory) {
    /// forgefmt: disable-next-item
    return OperatorSet({
        avs: address(uint160(uint256(_key) >> 96)),
        id: uint32(uint256(_key) & type(uint96).max) // @audit lossy casting here if the key has bits 136-192 set
    });
}
```

This issue is rated as informational severity as each occurrence of `decode()` occurs on keys which have been encoded by `OperatorSetLib`. Therefore, it is not possible to call `decode()` with a `_key` that has bits 136 to 192 set.

Recommendations

Consider performing safe casting and revert if `uint96(uint256(_key)) >= (1 << 32)`.

Furthermore, the `& type(uint96).max` condition is not necessary when casting a `uint256` to `uint32` as any values larger than 2^{32} will be truncated. Consider removing the unnecessary code.

Resolution

The EigenLayer team has acknowledged this issue and has chosen to not implement a fix as there is no known attack vector.

EGSL-19	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Redundant Event Emission**

   Related Asset(s): AllocationManager.sol

   The `createOperatorSets()` function allows duplicate strategies within `params[i].strategies` when adding to operator sets. While the `EnumerableSet` library prevents actual duplicate entries in storage, there is potential to trigger redundant `StrategyAddedToOperatorSet` events despite no state change.

   AllocationManager.sol::createOperatorSets()

   ```
   bytes32 operatorSetKey = operatorSet.key();
   for (uint256 j = 0; j < params[i].strategies.length; j++) {
       _operatorSetStrategies[operatorSetKey].add(address(params[i].strategies[j]));
       emit StrategyAddedToOperatorSet(operatorSet, params[i].strategies[j]);
   }
   ```

   Wrap the `add()` line in `require()` statement to ensure there are no duplicate strategies, or check that the `params[i].strategies` array is sorted and that consecutive elements are not identical.

2. **Gas Optimisation**

   Related Asset(s): DelegationManager.sol

   In the `_undelegate()` function, consider declaring the relevant arrays outside of the `for` loop to save gas

   DelegationManager.sol::_undelegate()

   ```
   for (uint256 i = 0; i < strategies.length; i++) {
       IStrategy[] memory singleStrategy = new IStrategy[](1);
       uint256[] memory singleDepositShares = new uint256[](1);
       uint256[] memory singleSlashingFactor = new uint256[](1);
       // ...
   }
   ```

3. **Missing Return Value Check In `_removeSharesAndQueueWithdrawal()`**

   Related Asset(s): DelegationManager.sol

   `_removeSharesAndQueueWithdrawal()` does not verify that the withdrawal root is not already in the set. While `withdrawalRoot` overlaps should not occur due to the uniqueness of the nonce field, explicitly checking the return value improves code clarity and maintainability.

DelegationManager.sol::_removeSharesAndQueueWithdrawal()

```
uint256 nonce = cumulativeWithdrawalsQueued[staker];
cumulativeWithdrawalsQueued[staker]++;

Withdrawal memory withdrawal = Withdrawal({
    staker: staker,
    delegatedTo: operator,
    withdrawer: staker,
    nonce: nonce,
    startBlock: uint32(block.number),
    strategies: strategies,
    scaledShares: scaledShares
});

bytes32 withdrawalRoot = calculateWithdrawalRoot(withdrawal);

pendingWithdrawals[withdrawalRoot] = true;
queuedWithdrawals[withdrawalRoot] = withdrawal;
// @audit wrap this in a require statement to ensure the withdrawal root
//        is not already in the set
_stakerQueuedWithdrawalRoots[staker].add(withdrawalRoot);

emit SlashingWithdrawalQueued(withdrawalRoot, withdrawal, withdrawableShares);
return withdrawalRoot;
```

Consider checking the returned value is true.

4. **Inconsistent NatSpec Documentation On Whole Gwei Requirements**

   Related Asset(s): EigenPodManager.sol EigenPod.sol

   The NatSpec for `EigenPodManager.withdrawSharesAsTokens()` states that `shares` must be a whole Gwei amount to avoid reverting. Similarly, `EigenPod.withdrawRestakedBeaconChainETH()` specifies that `amountWei` must also be a whole Gwei amount to prevent reverts.

   This requirement is no longer enforced after the slashing upgrade.

   Consider updating the NatSpec comments to remove the requirement for whole Gwei amounts and clarify that any sub-Gwei values are rounded off.

5. **`modifyAllocations()` Allows Empty `params` Array**

   Related Asset(s): AllocationManager.sol

   The function `modifyAllocations()` allows the function parameter `params` to be of length zero. When this is the case no allocations are modified.

   Consider adding a check to ensure `params.length > 0`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The EigenLayer team has acknowledged the issues above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/tests-local/OperatorSetLib.t.sol:OperatorSetLibTest
[PASS] testFuzz_KeyDecode(address,uint32) (runs: 1008, μ: 4421, ~: 4421)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 30.17ms (27.68ms CPU time)

Ran 2 tests for test/tests-local/AllocationManager.t.sol:AllocationManagerTest
[PASS] testDiffFuzz_addInt128(uint64,int128) (runs: 1008, μ: 17544, ~: 17371)
[PASS] testFuzz_addInt128(uint64,int128) (runs: 1008, μ: 15525, ~: 14924)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 66.33ms (120.61ms CPU time)

Ran 1 test for test/tests-fork/upgrade/EigenPodManager.upgrade.fork.t.sol:EigenPodManagerUpgradeForkTest
[PASS] test_addShares_NegativeInitialShareBalanceIncorrectOperatorShares_Vuln() (gas: 26540117)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.25s (22.33ms CPU time)

Ran 1 test for test/tests-fork/upgrade/DelegationManager.preupgrade.fork.t.sol:DelegationManagerUpgradeForkTest
[PASS] test_completeQueuedWithdrawals_OverSlashingAfterUpgrade_Vuln() (gas: 27376317)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.40s (171.94ms CPU time)

Ran 14 tests for test/tests-fork/DelegationManager.fork.t.sol:DelegationManagerForkTest
[PASS] test_DelegationViaQueuedWithdrawal() (gas: 1074067)
[PASS] test_RevertWhen_UsingOldApproverSignature() (gas: 209816)
[PASS] test_WithdrawableShares_SlashAfterQueuedWithdrawal() (gas: 2369717)
[PASS] test_WithdrawableShares_SlashBeforeQueuedWithdrawal() (gas: 2369688)
[PASS] test_ZeroDepositSharesWithdrawal() (gas: 187394)
[PASS] test_delegate_UpdateDSF_AfterBeaconChainSlashingWithSecondValidator() (gas: 4006650)
[PASS] test_delegate_WithdrawablePodSharesAfterSlashingAndDelegation_Vuln() (gas: 1957218)
[SKIP: Known accepted issue]
      ↪  test_increaseDelegation_BeaconChainSlashingFactorIsZero_CantCheckpointUnverifiedValidatorWithdrawnEth_Vuln() (gas: 0)
[SKIP: Known accepted issue] test_increaseDelegation_BeaconChainSlashingFactorIsZero_NewValidatorCantVerify_Vuln() (gas: 0)
[PASS] test_increaseDelegation_MaxMagnitudeIsZero_CantCheckpointUnverifiedValidatorWithdrawnEth() (gas: 2475919)
[PASS] test_increaseDelegation_MaxMagnitudeIsZero_NewValidatorCantVerify() (gas: 1956916)
[PASS] test_removeSharesAndQueueWithdrawal_OperatorSharesDelta() (gas: 2398122)
[PASS] test_slashOperator_BurnableShares() (gas: 2521780)
[PASS] test_slashOperator_NoBeaconChainSlashableSharesInQueue_Vuln() (gas: 2282891)
Suite result: ok. 12 passed; 0 failed; 2 skipped; finished in 1.41s (321.46ms CPU time)

Ran 21 tests for test/tests-fork/EigenPodManager.fork.t.sol:EigenPodManagerForkTest
[PASS] test_BeaconChainSlashingFactorIsNotZeroWithPendingWithdrawal() (gas: 3448820)
[PASS] test_BeaconChainSlashingFactorShouldBeZeroAfterFullySlashed() (gas: 2684739)
[PASS] test_CannotWithdrawWhenStillStakedOnBeaconChain() (gas: 1969589)
[PASS] test_FullySlashedValidatorShouldWithdrawZero() (gas: 3133816)
[PASS] test_MultipleValidatorOnPod() (gas: 2465588)
[PASS] test_MultipleValidatorWithDifferentApproach() (gas: 2415802)
[PASS] test_MultipleValidators() (gas: 3304484)
[PASS] test_MustCompleteCheckPoint() (gas: 1445403)
[PASS] test_PreventReducingBeaconChainSlashingFactorByDirectlySendingEth() (gas: 1741941)
[PASS] test_TestWithdrawalSubGweiAmountRounding() (gas: 2351691)
[PASS] test_ValidatorExitDoesNotReduceSlashingFactor() (gas: 2287011)
[PASS] test_WithdrawableShares_SlashBeaconChainThenCheckpointThenSlashBeaconChain_StakeSecondValidatorBeforeCheckpoint() (gas:
      ↪  2729497)
[PASS] test_WithdrawableShares_SlashBeaconChainThenCheckpointThenSlashOperator_StakeSecondValidatorAfterSlashOperator() (gas:
      ↪  2874079)
[PASS] test_WithdrawableShares_SlashBeaconChainThenCheckpointThenSlashOperator_StakeSecondValidatorBeforeSlashOperator() (gas:
      ↪  2872649)
[PASS] test_WithdrawableShares_SlashOperatorBeforeDelegate_StakeSecondValidatorBeforeCheckpoint() (gas: 2970427)
[SKIP: EGSL-02: Issue acknowledged]
      ↪  test_WithdrawableShares_SlashOperatorThenSlashBeaconChainThenCheckpoint_StakeSecondValidatorAfterFinalizeCheckpoint_Vuln()
      ↪  (gas: 0)
[SKIP: EGSL-02: Issue acknowledged]
      ↪  test_WithdrawableShares_SlashOperatorThenSlashBeaconChainThenCheckpoint_StakeSecondValidatorBeforeFinalizeCheckpoint_Vuln()
      ↪  (gas: 0)
```

```
[SKIP: EGSL-02: Issue acknowledged]
      ↪  test_WithdrawableShares_SlashOperatorThenSlashBeaconChainThenCheckpoint_StakeSecondValidatorBeforeStartCheckpoint_Vuln()
      ↪  (gas: 0)
[SKIP: EGSL-01: Issue acknowledged] test_WithdrawableShares_SlashOperatorThenSlashBeaconChain_Vuln() (gas: 0)
[PASS] test_WithdrawableShares_VerifyWithdrawalCredentialsAfterStartCheckpoint() (gas: 2269991)
[PASS] test_onlyOwnerCanStartCheckPoint() (gas: 1381064)
Suite result: ok. 17 passed; 0 failed; 4 skipped; finished in 1.44s (845.70ms CPU time)

Ran 31 tests for test/tests-fork/AllocationManager.fork.t.sol:AllocationManagerForkTest
[PASS] testFuzz_slashOperator_MagnitudeInvariants(uint256,int256,uint256) (runs: 1008, μ: 501038, ~: 490463)
[PASS] test_DeallocationQueueUnsortedAfterPartialSlashAndModify_Vuln() (gas: 1196717)
[PASS] test_addStrategiesToOperatorSet() (gas: 179545)
[PASS] test_clearDeallocationQueue() (gas: 813827)
[PASS] test_clearDeallocationQueue_withUncompletableDeallocation() (gas: 869404)
[PASS] test_createOperatorSets() (gas: 97059)
[PASS] test_deregisterFromOperatorSets() (gas: 350901)
[PASS] test_getAllocatableMagnitude() (gas: 968728)
[PASS] test_getAllocatedSets() (gas: 901521)
[PASS] test_getAllocatedStrategies() (gas: 900742)
[PASS] test_getAllocationDelay_withPendingDelay() (gas: 934708)
[PASS] test_getAllocations() (gas: 919395)
[PASS] test_getMaxMagnitudes() (gas: 1005128)
[PASS] test_getMaxMagnitudesAtBlock() (gas: 911419)
[PASS] test_getMembers_and_getMemberCount() (gas: 1146517)
[PASS] test_getMinimumSlashableStake() (gas: 915887)
[SKIP: EGSL-07: Issue acknowledged] test_getMinimumSlashableStake_IncorrectValue_Vuln() (gas: 0)
[PASS] test_getMinimumSlashableStake_pendingDeallocation() (gas: 1015052)
[PASS] test_getOperatorSetCount() (gas: 398747)
[PASS] test_getStrategyAllocations() (gas: 982646)
[PASS] test_modifyAllocations_immediateDeallocation() (gas: 793485)
[PASS] test_registerForOperatorSets() (gas: 413543)
[PASS] test_removeStrategiesFromOperatorSet() (gas: 195828)
[PASS] test_setAVSRegistrar() (gas: 245904)
[PASS] test_slashOperator() (gas: 1091209)
[PASS] test_slashOperator_rounding() (gas: 1113392)
[PASS] test_slashOperator_roundingDownSmallShares() (gas: 1095335)
[PASS] test_slashOperator_withPendingDeallocation() (gas: 1165938)
[PASS] test_slashing_burns_correct_share_amount_during_queueWithdraw() (gas: 1339992)
[PASS] test_slashing_only_affects_existing_stakers() (gas: 1572476)
[PASS] test_updateAVSMetadataURI() (gas: 38720)
Suite result: ok. 30 passed; 0 failed; 1 skipped; finished in 2.30s (1.16s CPU time)

Ran 7 tests for test/tests-local/Snapshots.t.sol:SnapshotsTest
[PASS] testFuzz_binarySearchEdges(uint32[],uint64) (runs: 1003, μ: 487456, ~: 494229)
[PASS] testFuzz_boundaryKeys(uint64) (runs: 1008, μ: 143843, ~: 144870)
[PASS] testFuzz_revertOnInvalidSnapshotOrdering(uint32,uint32,uint64) (runs: 1003, μ: 49070, ~: 49070)
[PASS] testFuzz_sequentialPushes(uint32,uint32,uint8) (runs: 1008, μ: 294363, ~: 273602)
[PASS] testFuzz_updateExistingSnapshot(uint32,uint64,uint64) (runs: 1008, μ: 100476, ~: 100501)
[PASS] testFuzz_upperLookup(uint32,uint32) (runs: 1008, μ: 2151090, ~: 2033850)
[PASS] test_emptySnapshots() (gas: 12124)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 2.61s (3.78s CPU time)

Ran 8 test suites in 2.62s (10.51s CPU time): 71 tests passed, 0 failed, 7 skipped (78 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

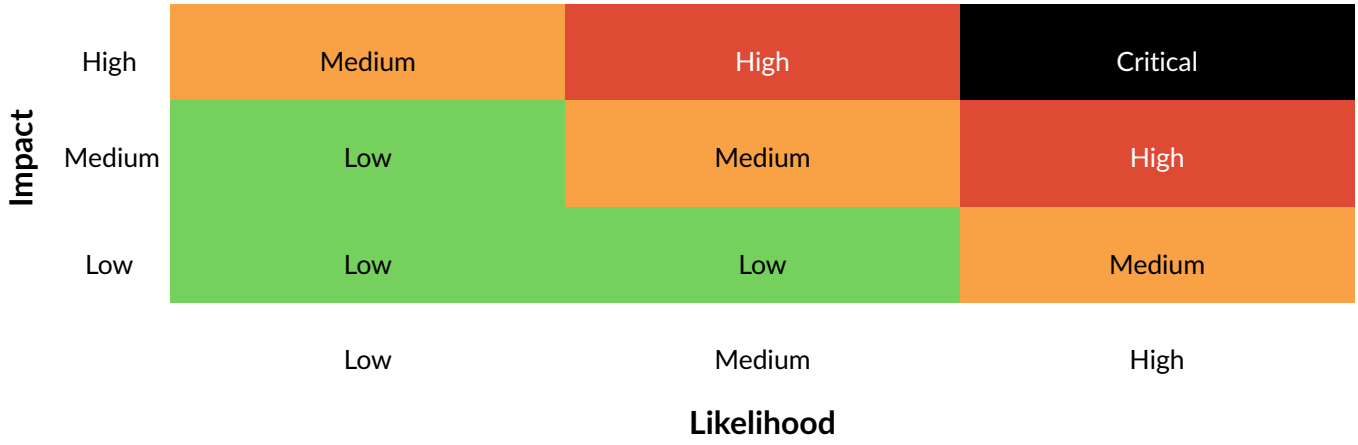

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].