Layr Labs

Rewards v2

EigenLayer Sidecar Review

Version: 2.0

January, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Layr Labs components. 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 components 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 components in scope.

Overview

Rewards v2 is an upgrade to EigenLayer's rewards system, enabling performance-based rewards distribution from AVSs to operators and implementing variable operator fees. The upgrade modifies the core protocol, middleware, and sidecar components, while maintaining on-chain verifiability and security of all rewards calculations.

The new features introduced by Rewards v2 include:

1. Performance-based and directed rewards from AVSs to operators,
2. Variable operator fee on AVS rewards,
3. Operator split for Programmatic Incentives specifically,
4. Batch rewards claiming for stakers and operators.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the following repositories:

1. Layr-Labs/eigenlayer-contracts
2. Layr-Labs/eigenlayer-middleware
3. Layr-Labs/sidecar

The scope of this time-boxed review was strictly limited to files at the following commits:

1. eigenlayer-contracts@b8567e4.
   - `src/contracts/core/RewardsCoordinator.sol`
   - `src/contracts/core/RewardsCoordinatorStorage.sol`
   - `src/contracts/interfaces/IRewardsCoordinator.sol`
2. eigenlayer-middleware@91400d9.
   - `src/ServiceManagerBase.sol`
   - `src/ServiceManagerBaseStorage.sol`
   - `src/interfaces/IServiceManager.sol`
   - `src/unaudited/ECDSAServiceManagerBase.sol`
   - `lib/eigenlayer-contracts` (Respective imports)
3. sidecar@d9bb2ab.
   - Changes in PR #106
   - Changes in PR #154

The fixes of the identified issues were assessed at the following commits:

1. eigenlayer-contracts@47f1232
2. eigenlayer-middleware@027226b
3. sidecar@2941257

*Note: third party libraries and dependencies and the **SQL components of** `sidecar` 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`

For the Golang libraries and modules, the review focused on internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime. The manual review explored known Golang antipatterns such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks and a multitude of panics including but not limited to `nil` pointer deferences, index out of bounds and calls to `panic()`.

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

- golangci-lint: `https://golangci-lint.run/`
- vet: `https://pkg.go.dev/cmd/vet`
- errcheck: `https://github.com/kisielk/errcheck`

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 13 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.

- High: 1 issue.
- Medium: 1 issue.
- Low: 5 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Layr Labs components 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
ELSC2-01	Incorrect Operator Split Filtering Logic	Critical	Resolved
ELSC2-02	Incorrect Split Window Filtering Due To End Time Calculation	High	Resolved
ELSC2-03	Inefficient Leaf Encoding Using String Representations	Medium	Closed
ELSC2-04	Inconsistent Number Formatting In Merkle Tree Values	Low	Closed
ELSC2-05	Missing `SafeERC20` Usage For Token Interactions	Low	Resolved
ELSC2-06	Griefing Attack On Reward Claims Through Front-Running `setClaimer()`	Low	Closed
ELSC2-07	Operators Can Lock In Current Default Split By Front-Running Changes	Low	Resolved
ELSC2-08	Same Day Split Selection Relies On Implicit Ordering	Low	Resolved
ELSC2-09	Fee-On-Transfer And Rebasing Tokens Can Cause Issues In Rewards Accounting	Informational	Closed
ELSC2-10	Use Of Deprecated Package `xerrors`	Informational	Resolved
ELSC2-11	Split Activation Delay Shares Same Delay As Distribution Roots	Informational	Closed
ELSC2-12	Miscellaneous General Comments	Informational	Closed
ELSC2-13	Hardcoded Default Operator Split Values Used In Reward Calculations	Informational	Resolved

ELSC2-01	Incorrect Operator Split Filtering Logic		
Asset	`operatorAvsSplitSnapshots.go`, `operatorPISplitSnapshots.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The current implementation incorrectly discards valid split events due to a flawed join condition in the SQL queries, potentially leading to incorrect reward distributions.

The issue exists in both operator AVS and PI split snapshot calculations. The problematic join conditions in the `decorated_operator_avs_splits` and `decorated_operator_splits` CTEs use `rops.rn > rops2.rn`, which incorrectly filters out newer split events when they have later activation times than older events.

operatorAvsSplitSnapshots.go

```
ranked_operator_avs_split_records as (
    SELECT
        *,
        -- round activated up to the nearest day
        date_trunc('day', activated_at) + INTERVAL '1' day AS rounded_activated_at,
        -- @audit older split events are are ranked higher (lower rn)
        ROW_NUMBER() OVER (PARTITION BY operator, avs ORDER BY block_time asc, log_index asc) AS rn
    FROM operator_avs_splits_with_block_info
),
decorated_operator_avs_splits as (
    select
        rops.*,
        -- @audit incorrect condition: rops.rn > rops2.rn will discard newer splits
        case when rops2.block_time is not null then false else true end as active
    from ranked_operator_avs_split_records as rops
    left join ranked_operator_avs_split_records as rops2 on (
        rops.operator = rops2.operator
        and rops.avs = rops2.avs
        -- @audit this should be rops.rn < rops2.rn to filter out older splits
        and rops.rn > rops2.rn
        and rops2.rounded_activated_at <= rops.rounded_activated_at
    )
)
```

Example scenario:

1. Split A: Set at Day 1, activated at Day 3
2. Split B: Set at Day 2, activated at Day 4

The current logic would incorrectly discard Split B because:

- `rops` would be Split B (higher `rn`)
- `rops2` would be Split A (lower `rn`)
- The condition `rops.rn > rops2.rn` is true

- The condition `rops2.rounded_activated_at <= rops.rounded_activated_at` is true (Split A is activated before Split B)
- Split B gets marked as inactive and is filtered out

As overlapping splits are not allowed onchain due to a restriction in the `RewardsCoordinator` contract, the end result is that all subsequent splits after the first meet the conditions and are filtered out.

This issue is classified as high impact because it directly affects reward distributions by using incorrect split ratios, potentially leading to incorrectly distributed rewards for operators or stakers. A malicious operator can set a short 100% initial split and then replace it with a 0% split to deceive stakers into believing they are receiving 100% of the rewards when they are actually receiving 0%.

Recommendations

The join condition should be changed to `rops.rn < rops2.rn` in both files. This will correctly filter out older split events that have later activation times, ensuring that newer split configurations take precedence.

Resolution

The EigenLayer team has fixed this issue by selecting only the latest activated split for each operator, AVS, and activation date.

This issue has been resolved in commit 7de44a2.

ELSC2-02	Incorrect Split Window Filtering Due To End Time Calculation		
Asset	`operatorAvsSplitSnapshots.go, operatorPISplitSnapshots.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The current implementation incorrectly filters out valid operator splits that are activated one day before the next split, leading to missing reward calculations and incorrect split history.

The issue occurs in the `operator_avs_split_windows` CTE where the `end_time` calculation subtracts one day from the next split's activation time. This subtraction, combined with the subsequent filtering in the `cleaned_records` CTE where `start_time < end_time`, causes the system to discard valid splits that should be included in the rewards calculation.

operatorAvsSplitSnapshots.go

```
-- Get the range for each operator, avs pairing
operator_avs_split_windows as (
  SELECT
    operator, avs, split, snapshot_time as start_time,
    CASE
      -- If the range does not have the end, use the current timestamp truncated to 0 UTC
      WHEN LEAD(snapshot_time) OVER (PARTITION BY operator, avs ORDER BY snapshot_time) is null THEN date_trunc('day', TIMESTAMP
        ↪  '{{.cutoffDate}}')

      -- need to subtract 1 day from the end time since generate_series will be inclusive below.
      ELSE LEAD(snapshot_time) OVER (PARTITION BY operator ORDER BY snapshot_time) - interval '1 day'
      END AS end_time
  FROM active_operator_splits
),

-- @audit This filtering removes valid splits that activate one day before the next split
cleaned_records as (
  SELECT * FROM operator_avs_split_windows
  WHERE start_time < end_time
)
```

The comment specifies that the 1-day subtraction is needed as `generate_series()` is inclusive of the `end_time`. However, this is mistakenly done twice, as the 1-day subtraction also occurs in the `final_results` CTE:

operatorAvsSplitSnapshots.go

```
final_results as (
  -- ...
  FROM
    cleaned_records
      CROSS JOIN
    -- @audit 1 day is also subtracted here
    generate_series(DATE(start_time), DATE(end_time) - interval '1' day, interval '1' day) AS d
)
```

Recommendations

Remove the day subtraction from the `end_time` calculation in the `operator_avs_split_windows` CTE as the end time should be exactly when the next split starts.

Do the same for the `operator_pi_split_windows` CTE in `operatorPISplitSnapshots.go`.

Resolution

The EigenLayer team has implemented the fix recommended above.

This issue has been resolved in commit 44dc5ad.

ELSC2-03	Inefficient Leaf Encoding Using String Representations		
Asset	`eigenState/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There exists values in the state tree which are inefficiently encoded as string representations instead of raw bytes, leading to increased gas costs and complexity if these proofs are verified onchain.

1. `SlotID`s of `OperatorDirectedRewardSubmissionsModel` and every other state model through `baseEigenState::NewSlowIDWithSuffix()`:

operatorDirectedRewardSubmissions.go::NewSlotID()

```
func NewSlotID(transactionHash string, logIndex uint64, rewardHash string, strategyIndex uint64, operatorIndex uint64)
    ↪ types.SlotID {
    return base.NewSlotIDWithSuffix(transactionHash, logIndex, fmt.Sprintf("%s_%d_%d", rewardHash, strategyIndex,
        ↪ operatorIndex))
}
```

baseEigenState.go::NewSlotIDWithSuffix()

```
func NewSlotIDWithSuffix(txHash string, logIndex uint64, suffix string) types.SlotID {
    baseSlotId := fmt.Sprintf("%s_%016x", txHash, logIndex)
    if suffix != "" {
        baseSlotId = fmt.Sprintf("%s_%s", baseSlotId, suffix)
    }
    return types.SlotID(baseSlotId)
}
```

2. Leaf values of each state model through the use of `fmt.Sprintf()`:

operatorDirectedRewardSubmissions.go::sortValuesForMerkleTree()

```
for _, split := range splits {
    // @audit slotID and value are both both utf-8 strings
    slotID := base.NewSlotID(split.TransactionHash, split.LogIndex)
    value := fmt.Sprintf("%s_%s_%d_%d_%d", split.Operator, split.Avs, split.ActivatedAt.Unix(),
        ↪ split.OldOperatorAVSSplitBips, split.NewOperatorAVSSplitBips)
    inputs = append(inputs, &base.MerkleTreeInput{
        SlotID: slotID,
        Value:  []byte(value),
    })
}
```

The impact is classified as low since state proofs are currently not implemented onchain. If they were to be implemented onchain, the current state tree implemention would significantly increase gas costs and complexity as values need to be converted from their `UTF-8` string representations.

Recommendations

Modify the encoding to use raw bytes instead of string representations. This can be achieved via byte concatenation by using `append()` or `copy()`, since the length of the `SlotID` and `Value` should be fixed for each model.

Resolution

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

> *"We have chosen to not refactor the model leaves as it would require a lot of changes to the codebase. Since we're not writing state roots onchain currently, this is not a priority to fix right now."*

ELSC2-04	Inconsistent Number Formatting In Merkle Tree Values		
Asset	`eigenState/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current implementation uses inconsistent numeric formatting in Merkle tree leaf nodes, leading to variable leaf lengths and potential sorting issues since the values are compared lexicographically. This manifests in two ways:

1. Storage of numeric values as decimal strings in structs (e.g., `Amount` and `Multiplier` in `OperatorDirectedRewardSubmission`)

   OperatorDirectedRewardSubmission.go

   ```
   type OperatorDirectedRewardSubmission struct {
       // ...
       Amount          string
       Multiplier      string
       // ...
   }
   ```

   OperatorPISplitModel.go

   ```
   func (ops *OperatorPISplitModel) sortValuesForMerkleTree(splits []*OperatorPISplit) []*base.MerkleTreeInput {
       // ...
       value := fmt.Sprintf("%s_%d_%d_%d", split.Operator, split.ActivatedAt.Unix(), split.OldOperatorPISplitBips,
           ↪  split.NewOperatorPISplitBips)
       // ...
   }
   ```

2. Direct use of decimal formatting (`%d`) in string formatting

Since these values are sorted lexicographically using `strings.Compare()`, numbers like `"2"` and `"10"` would be sorted as `"10" < "2"`, which is incorrect for numerical ordering. Additionally, storing numeric values as decimal strings introduces unnecessary string conversion overhead and potential parsing issues.

Recommendations

Consider implementing the following changes:

1. Change string representations of numeric values to appropriate `big.Int` types in structs:

   OperatorDirectedRewardSubmission.go

   ```
   type OperatorDirectedRewardSubmission struct {
       // ...
       Amount          *big.Int    // Changed from string
       Multiplier      *big.Int    // Changed from string
       // ...
   }
   ```

2. Replace decimal formatting (`%d`) with fixed-length hexadecimal formatting (`%016x` for `uint64` , `%024x` for `uint96` , `%064x` for `uint256`) for all numeric values in the Merkle tree leaf construction:

OperatorPISplitModel.go

```
func (ops *OperatorPISplitModel) sortValuesForMerkleTree(splits []*OperatorPISplit) []*base.MerkleTreeInput {
    // ...
    value := fmt.Sprintf("%s_0x%016x_0x%016x_0x%016x", split.Operator, split.ActivatedAt.Unix(),
        ↪  split.OldOperatorPISplitBips, split.NewOperatorPISplitBips)
    // ...
}
```

OperatorDirectedRewardSubmissionsModel.go

```
func (odrs *OperatorDirectedRewardSubmissionsModel) sortValuesForMerkleTree(submissions []*OperatorDirectedRewardSubmission)
    ↪  []*base.MerkleTreeInput {
    // ...
    // submission.Multiplier and submission.Amount need to be parsed from decimal strings
    // submission.Multiplier is a uint96, which is 12 bytes
    value := fmt.Sprintf("%s_%s_0x%024x_%s_0x%064x", submission.RewardHash, submission.Strategy, submission.Multiplier,
        ↪  submission.Operator, submission.Amount)
    // ...
}
```

Keep in mind that the hexadecimal formatting needs to be prefixed with `0x` to keep the formatting consistent.

Resolution

The EigenLayer team has replaced the decimal formatting in the leaf value strings with hexadecimal formatting in commits 770c596 and fc36524.

String representations of fields in structs have been kept as-is as they are only used during the transformation phase and the strings are easier to deal with.

ELSC2-05	Missing `SafeERC20` Usage For Token Interactions		
Asset	`ServiceManagerBase.sol, ECDSAServiceManagerBase.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The contract directly interacts with `ERC-20` tokens without using OpenZeppelin's `SafeERC20` library, which could lead to failed transfers and approvals going undetected or transactions reverting unexpectedly.

The contract makes direct `ERC-20` calls in both `createAVSRewardsSubmission()` and `createOperatorDirectedAVSRewardsSubmission()` functions.

ServiceManagerBase.sol::createAVSRewardsSubmission()

```
for (uint256 i = 0; i < rewardsSubmissions.length; ++i) {
    // Step 1: Transfer tokens to the contract
    rewardsSubmissions[i].token.transferFrom(
        msg.sender,
        address(this),
        rewardsSubmissions[i].amount
    );

    // Step 2: Approve the RewardsCoordinator to transfer the tokens
    uint256 allowance = rewardsSubmissions[i].token.allowance(
        address(this),
        address(_rewardsCoordinator)
    );
    rewardsSubmissions[i].token.approve(
        address(_rewardsCoordinator),
        rewardsSubmissions[i].amount + allowance
    );
}

// Step 3: Call the RewardsCoordinator to create the rewards submission
_rewardsCoordinator.createAVSRewardsSubmission(rewardsSubmissions);
```

There are two types of non-standard `ERC-20` tokens where this could lead to issues:

1. Tokens that return `false` instead of reverting on failed transfers (e.g. `EURS`)
2. Tokens that require setting allowance to 0 first before being able to set a non-zero allowance (e.g. `USDT`)

In the former case, the first transfer in Step 1 may fail silently, but the function would still continue to execute and call `RewardsCoordinator` to approve the transfer in Step 2. `RewardsCoordinator` will then try to transfer the tokens in from the contract in Step 3 and fail, as the balance is insufficient.

In the latter case, the approval in Step 2 will fail, as the allowance may be non-zero from a previous approval. This occurs if there are multiple reward submissions for the same token in the same call.

This issue has a low impact and low likelihood of occurring, as the non-standard `ERC-20` tokens are uncommon. Furthermore, in the latter case, it is possible to work around the issue by separating the reward submissions into multiple calls.

Recommendations

Use OpenZeppelin's `SafeERC20` library for all token interactions.

Resolution

The EigenLayer team has implemented the fix recommended above.

This issue has been resolved in commit 027226b.

ELSC2-06	Griefing Attack On Reward Claims Through Front-Running `setClaimer()`		
Asset	`RewardsCoordinator.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

An earner can grief a claimer by front-running their `processClaims()` transaction with `setClaimer()`, causing the claimer's transaction to revert and waste gas.

When processing multiple claims via `processClaims()`, all claims must succeed for the transaction to complete. The function validates that `msg.sender` matches the claimer for each earner. However, an earner can front-run this transaction by calling `setClaimer()` to change their designated claimer, causing the validation to fail and the entire transaction to revert:

RewardsCoordinator.sol::processClaims()

```
function processClaims(
    RewardsMerkleClaim[] calldata claims,
    address recipient
) external onlyWhenNotPaused(PAUSED_PROCESS_CLAIM) nonReentrant {
    for (uint256 i = 0; i < claims.length; i++) {
        _processClaim(claims[i], recipient);
    }
}
```

RewardsCoordinator.sol::_processClaim()

```
function _processClaim(RewardsMerkleClaim calldata claim, address recipient) internal {
    // ...
    require(msg.sender == claimer, "RewardsCoordinator.processClaim: caller is not valid claimer");
    // ...
}
```

This issue is classified as low impact since it only results in wasted gas for the claimer, but medium likelihood since it requires minimal coordination and can be executed either maliciously or accidentally.

Recommendations

Consider implementing one of the following:

1. Allow partial claim processing by skipping failed claims rather than reverting the entire transaction
2. Add a time delay before claimer changes take effect, similar to how operator splits are handled

Resolution

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

"The claimer is a trusted address (and mostly owned by the earner). There is a trust assumption that the earner will not grief the claimer.

ELSC2-07	Operators Can Lock In Current Default Split By Front-Running Changes		
Asset	`RewardsCoordinator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Operators can front-run changes to `defaultOperatorSplitBips` by setting a custom split, allowing them to retain the old default split value rather than accepting the new one.

When an operator sets their split for the first time via `setOperatorAVSSplit()` or `setOperatorPISplit()`, the `_setOperatorSplit()` function initialises `oldSplitBips` to the current `defaultOperatorSplitBips` value. This means that if an operator detects a pending governance transaction to change the default split (e.g., from 1000 to 500 bips), they can front-run it by setting a custom split, which will lock in the current default value of 1000 bips as their `oldSplitBips`.

RewardsCoordinator.sol::_setOperatorSplit()

```
function _setOperatorSplit(OperatorSplit storage operatorSplit, uint16 split, uint32 activatedAt) internal {
    require(
        block.timestamp > operatorSplit.activatedAt,
        "RewardsCoordinator._setOperatorSplit: earlier split not activated yet"
    );
    if (operatorSplit.activatedAt == 0) {
        // If the operator split has not been initialized yet, set the old split to the default split.
        operatorSplit.oldSplitBips = defaultOperatorSplitBips;
    } else {
        operatorSplit.oldSplitBips = operatorSplit.newSplitBips;
    }
    operatorSplit.newSplitBips = split;
    operatorSplit.activatedAt = activatedAt;
}
```

Recommendations

Consider keeping track of the `activatedAt` timestamp of the old split in the `OperatorSplit` struct. If `oldSplitActivatedAt = 0` and the pending split hasn't been activated yet, then also return `defaultOperatorSplitBips` from `_getOperatorSplit()`:

RewardsCoordinator.sol::_getOperatorSplit()

```
function _getOperatorSplit(OperatorSplit memory operatorSplit) internal view returns (uint16) {
    if (
        (operatorSplit.activatedAt == 0) ||
        (operatorSplit.activatedAt > block.timestamp && operatorSplit.oldSplitActivatedAt == 0)
    ) {
        return defaultOperatorSplitBips;
    }
    // ...
}
```

Resolution

The EigenLayer team has fixed this issue by using `oldSplitBips = type(uint16).max` as a flag to indicate that the operator split has not been initialised yet.

The `_getOperatorSplit()` function now checks for the flag and returns `defaultOperatorSplitBips` instead of `oldSplitBips`.

This issue has been resolved in commit 47f1232.

ELSC2-08	Same Day Split Selection Relies On Implicit Ordering		
Asset	`operatorAvsSplitSnapshots.go`, `operatorPISplitSnapshots.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When the `activationDelay` in `RewardsCoordinator` is less than one day, multiple operator splits can have activation times that fall within the same day. Due to rounding up to the next day boundary, these splits end up having the same effective activation time, relying on implicit ordering from previous `ROW_NUMBER()` operations to determine which split is selected.

The issue exists in the SQL query where multiple splits with the same rounded activation time are processed. While there is existing logic to filter overlapping splits, this logic is currently incorrect (see issue ELSC2-01). If this logic were to be removed, the selection between same-day splits would depend on implicit ordering from previous `ROW_NUMBER()` operations:

operatorAvsSplitSnapshots.go

```
ranked_operator_avs_split_records as (
    SELECT
        *,
        -- round activated up to the nearest day
        date_trunc('day', activated_at) + INTERVAL '1' day AS rounded_activated_at,
        -- @audit initial ordering is done here by block_time and log_index
        ROW_NUMBER() OVER (PARTITION BY operator, avs ORDER BY block_time asc, log_index asc) AS rn
    FROM operator_avs_splits_with_block_info
),

-- removed filtering overlapping splits CTEs

operator_avs_split_windows as (
  SELECT
    operator, avs, split, snapshot_time as start_time,
    CASE
      WHEN LEAD(snapshot_time) OVER (PARTITION BY operator, avs ORDER BY snapshot_time) is null
        THEN date_trunc('day', TIMESTAMP '{{.cutoffDate}}')
      ELSE LEAD(snapshot_time) OVER (PARTITION BY operator ORDER BY snapshot_time) - interval '1 day'
    END AS end_time
  FROM active_operator_splits
),
cleaned_records as (
  SELECT * FROM operator_avs_split_windows
  WHERE start_time < end_time
)
```

If multiple rows in the same window have identical ordering values, then by default the SQL engine can (and often does) return those rows in a non-deterministic order when using analytic functions like `LEAD()`.

The `LEAD()` function will arbitrarily order splits with the same activation time based on their previous ordering, causing all but one split to be filtered out when `start_time = end_time`.

Fortunately, due to the `ROW_NUMBER()` operation in the `ranked_operator_avs_split_records` CTE, the split will be ordered correctly by `block_time` and `log_index`.

This issue is classified as low severity instead of informational as even though the implicit ordering currently leads to the intended behaviour, small changes to the surrounding SQL code could lead to incorrect ordering. Hence, it is recommended to address this issue.

Recommendations

Consider any/both of the following recommendations:

1. Add explicit handling for splits with the same rounded activation time.
2. Enforce `RewardsCoordinator.activationDelay` to be greater than one day.

Resolution

As part of the fix for ELSC2-01, the SQL query now deterministically selects one split per operator and AVS on the same activation day.

This issue has been resolved in commit 7de44a2.

ELSC2-09	Fee-On-Transfer And Rebasing Tokens Can Cause Issues In Rewards Accounting
Asset	`RewardsCoordinator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The reward submission functions do not verify that the actual amount of tokens received matches the expected amount when transferring rewards tokens. This can lead to accounting inconsistencies if fee-on-transfer or rebasing tokens are used as reward tokens.

When processing rewards, the submission functions emit the reward submission `struct`, which contains the amount of tokens to distribute.

RewardsCoordinator.sol::createOperatorDirectedAVSRewardsSubmission()

```
emit OperatorDirectedAVSRewardsSubmissionCreated(
    msg.sender,
    avs,
    operatorDirectedRewardsSubmissionHash,
    nonce,
    operatorDirectedRewardsSubmission // @audit this contains the amount of tokens to distribute
);
```

However, for fee-on-transfer tokens, the actual amount received by the contract will be less than the amount emitted in the event. This creates a discrepancy between the emitted rewards amounts and the actual tokens available for distribution. When operators later try to claim their rewards, there won't be enough tokens to fulfill all claims.

A similar issue exists for rebasing tokens which use shares-based mechanisms to calculate token balances as these tokens tend to incur small rounding errors during transfers that result in less tokens being transferred than expected. An example of a token that exhibits this behavior is Lido's `stETH`.

This issue can lead to insolvency as the `RewardsCoordinator` may not have enough tokens to fulfil all claims. However, it is very unlikely that this will happen in practice. Firstly, fee-on-transfer tokens are relatively uncommon in DeFi. Secondly, the rounding errors in rebasing tokens are small enough (1-2 wei) that they are likely to be accounted for in the rewards pipeline, which rounds down calculated rewards, and the balance of rebasing tokens are likely to increase over time such that the rounding error is negated at time of claiming.

Recommendations

Consider blocking fee-on-transfer and rebasing tokens from being used as reward tokens.

To check for these types of tokens, you can check the balance of the token before and after the transfer. If the difference in balance is not equal to the expected amount, then the token is a fee-on-transfer token and the transaction should revert.

RewardsCoordinator.sol::createOperatorDirectedAVSRewardsSubmission()

```
uint256 balanceBefore = operatorDirectedRewardsSubmission.token.balanceOf(address(this));
operatorDirectedRewardsSubmission.token.safeTransferFrom(msg.sender, address(this), totalAmount);
uint256 balanceAfter = operatorDirectedRewardsSubmission.token.balanceOf(address(this));

require(
    balanceAfter - balanceBefore == totalAmount,
    "RewardsCoordinator: incorrect amount received"
);
```

However this check may not always reject rebasing tokens, as rounding errors may not always occur during transfers. It is important to communicate through documentation that AVSs should not use rebasing or fee-on-transfer tokens as reward tokens.

Resolution

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

> *"We warn AVSs not to use non-standard ERC20s in the documentation and will not be blocking its use if they do intend to go ahead."*

ELSC2-10	Use Of Deprecated Package `xerrors`
Asset	`eigenState/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The three new state models (`operatorAVSSplits` , `operatorPISplits` , and `operatorDirectedRewards`) use the `xerrors` package which has been deprecated since Go `1.13` . Using deprecated packages can lead to several issues including lack of security updates, incompatibility with newer Go versions, and potential removal in future Go releases.

The `xerrors` package was an experimental package that was merged into the standard `errors` package. Continuing to use it means the codebase is not following current Go best practices and may face maintenance issues in the future.

Recommendations

Replace all usage of `xerrors` with the standard `errors` package. The standard package provides all the same functionality including:

- Error wrapping with `fmt.Errorf("... %w", err)`
- Error unwrapping with `errors.Unwrap()`
- Error inspection with `errors.Is()` and `errors.As()`

Resolution

The EigenLayer team has implemented the recommended changes.

This issue has been resolved in commit d9bc465.

ELSC2-11	Split Activation Delay Shares Same Delay As Distribution Roots
Asset	`RewardsCoordinator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `activationDelay` variable is used for both distribution root activation and operator split changes, which could lead to suboptimal configurations since these two features may require different delay periods for security and operational purposes.

In both `setOperatorAVSSplit()` and `setOperatorPISplit()`, the activation delay for new splits is set using:

RewardsCoordinator.sol
```
uint32 activatedAt = uint32(block.timestamp) + activationDelay;
```

This same `activationDelay` variable is also used when submitting new distribution roots in `submitRoot()`. While this currently works, it couples two distinct features that may need different delay periods based on their respective security requirements and operational needs.

Recommendations

Consider creating a separate delay variable specifically for operator splits:

RewardsCoordinator.sol
```
uint32 public operatorSplitActivationDelay;
```

Add a setter function for this new delay:

RewardsCoordinator.sol
```
function setOperatorSplitActivationDelay(uint32 _delay) external onlyOwner {
    emit OperatorSplitActivationDelaySet(operatorSplitActivationDelay, _delay);
    operatorSplitActivationDelay = _delay;
}
```

Update the split setting functions to use this new delay variable instead of the shared `activationDelay`.

Resolution

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

> *"This is intended behavior as we want the delay variable to have global usage. We don't intend to change this in the future."*

ELSC2-12	Miscellaneous General Comments
Asset	All files
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. **Timestamp Calculations**

   Related Asset(s): operatorDirectedRewardSubmissions.go

   Consider performing the following timestamp calculations outside of the `for` loop, as it will be the same for all strategies.

   operatorDirectedRewardSubmissions.go::handleOperatorDirectedRewardSubmissionCreatedEvent()

   ```
   startTimestamp := time.Unix(int64(outputRewardData.StartTimestamp), 0)
   endTimestamp := startTimestamp.Add(time.Duration(outputRewardData.Duration) * time.Second)
   ```

2. **Unused Column**

   Related Asset(s): 8_goldOperatorODRewardAmounts.go

   The calculated column `split_pct` defined in the SQL statement using the following.

   8_goldOperatorODRewardAmounts.go

   ```
   COALESCE(oas.split, 1000) / CAST(10000 AS DECIMAL) as split_pct
   ```

   However, it is unused, consider removing if not needed.

3. **Unused Field**

   Related Asset(s): operatorDirectedRewardSubmissions.go

   The `Description` field in the `operatorDirectedRewardData` struct is unused.

   Consider removing the field if not needed.

4. **`address(0)` Check**

   Related Asset(s): RewardsCoordinator.sol

   Consider validating `avs != address(0)` in `setOperatorAVSSplit()`

5. **Enforce Day Boundaries Onchain**

   Related Asset(s): RewardsCoordinator.sol

   The offchain reward calculation pipeline takes daily snapshots on each day boundary (00:00 UTC). However, there is no input validation on `submitRoot()` to ensure that the `rewardsCalculationEndTimestamp` is a multiple of 86400 (the number of seconds in a day).

   Consider adding a check in `submitRoot()` to ensure that the `rewardsCalculationEndTimestamp` is a multiple of 86400.

6. **Zero Multiplier Validation**

   Related Asset(s): RewardsCoordinator.sol

   In `_validateCommonRewardsSubmission()`, consider validating `multiplier != 0` to ensure that there are no redundant strategies included in the rewards submission.

7. **Redundant Event Parameter**

   Related Asset(s): RewardsCoordinator.sol

   The `OperatorDirectedAVSRewardsSubmissionCreated` event unnecessarily includes both the `caller` and `avs` parameters, when they are the same due to the `require()` statement below:

   RewardsCoordinator.sol::createOperatorDirectedAVSRewardsSubmission()

   ```
   require(
       msg.sender == avs,
       "RewardsCoordinator.createOperatorDirectedAVSRewardsSubmission: caller is not the AVS"
   );
   ```

   Consider emitting and indexing `avs` only.

Recommendations

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

Resolution

The EigenLayer team has acknowledged the issues above.

ELSC2-13	Hardcoded Default Operator Split Values Used In Reward Calculations
Asset	`rewards/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The reward calculation system contains hardcoded default operator split values that could lead to incorrect reward distributions between operators and stakers.

Multiple SQL queries across the rewards calculation pipeline use a hardcoded default split value of `1000` (10%) when calculating reward distributions between operators and stakers. This value is used when the operator has not explicitly set a split value onchain, resulting in the `operator_avs_split_snapshots` table being empty for the snapshot. However, if the default split values are modified on-chain, these hardcoded values will not reflect the changes, resulting in incorrect reward calculations.

This issue has an informational severity rating as it has been discovered/known by the EigenLayer team during retesting.

Recommendations

Consider creating a new state model that keeps track of changes to the default operator split values. These default operator split values can be incorporated into the rewards calculation pipeline using a similar snapshotting mechanism as the `operator_avs_split_snapshots` table.

Resolution

The EigenLayer team has implemented a a new state model that keeps track of changes to the default operator split values.

The default operator split values are converted into daily snapshots and stored in the `default_operator_split_snapshots` table. These snapshots are then used when calculating the reward distributions between operators and stakers in the rewards calculation pipeline.

This issue has been resolved in commit 2941257.

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 16 tests for src/test/unit/RewardsCoordinatorUnitSigP.t.sol:RewardsCoordinatorUnitSigP
[PASS] testFuzz_MultipleCreateRewardsForAllEarners_BeforeAndAfter((address,uint256,uint256,uint256),uint256) (runs: 256, μ:
      ↪  2707061, ~: 2527391)
[PASS] testFuzz_MultipleCreateRewardsForAllSubmission_BeforeAndAfter((address,uint256,uint256,uint256),uint256) (runs: 256, μ:
      ↪  2633440, ~: 2519729)
[PASS] testFuzz_Revert_WhenAmountIsZero(uint256,uint256) (runs: 256, μ: 757022, ~: 757285)
[PASS] testFuzz_Revert_WhenAmountzero(uint256) (runs: 256, μ: 81354, ~: 81374)
[PASS] testFuzz_Revert_WhenBlocktimestampGreaterThanActivatedAt(bytes32,uint32) (runs: 256, μ: 106983, ~: 106983)
[PASS] testFuzz_Revert_WhenIndexGreaterThanLength(uint32) (runs: 256, μ: 29436, ~: 29436)
[PASS] testFuzz_Revert_WhenRewardsCalculationEndTimestampGreaterThanBlocktimestamp(bytes32,uint32) (runs: 256, μ: 28067, ~: 28067)
[PASS] testFuzz_Revert_rewardTooFarOut(uint256,uint256,uint256) (runs: 256, μ: 778070, ~: 778138)
[PASS] testFuzz_Revert_whenExceedingMaxReward(uint256,uint256,uint256) (runs: 256, μ: 778224, ~: 778136)
[PASS] testFuzz_Revert_whenRewardsCalculationEndTimestampGreaterThanRewardsCalculationEndTimestamp(bytes32,uint32) (runs: 256, μ:
      ↪  28116, ~: 28116)
[PASS] testFuzz_createAVSRewardsSubmission_BeforeAndAfter(address,uint256,uint256,uint256) (runs: 256, μ: 856952, ~: 857119)
[PASS] testFuzz_createRewardsForAllEarners_BeforeAndAfter(uint256,uint256,uint256) (runs: 256, μ: 858993, ~: 859125)
[PASS] testFuzz_createRewardsForAllSubmission_BeforeAndAfter(uint256,uint256,uint256) (runs: 256, μ: 854259, ~: 854388)
[PASS] testFuzz_revert_WhenDisableRoot(bytes32,uint32) (runs: 256, μ: 111104, ~: 111104)
[PASS] testFuzz_revert_WhenEmptyStrategy(uint256,uint256,uint256) (runs: 256, μ: 754352, ~: 754411)
[PASS] testFuzz_setClaimerFor(address) (runs: 256, μ: 51850, ~: 51850)
Suite result: ok. 16 passed; 0 failed; 0 skipped; finished in 431.03ms (1.75s CPU time)

Ran 2 tests for src/test/unit/RewardsCoordinatorUnitSigP.t.sol:testCreateOperatorDirectedAVSRewardsSubmission
[PASS] testFuzz_CreateOperatorDirectedAVSRewardsSubmission_BeforeAndAfter(address,uint256,uint256) (runs: 256, μ: 884993, ~:
      ↪  885251)
[PASS] testFuzz_MultipleCreateOperatorDirectedAVSRewardsSubmission_BeforeAndAfter((address,uint256,uint256),uint256) (runs: 256, μ:
      ↪  2736603, ~: 2600861)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 537.31ms (2.32s CPU time)
```

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.

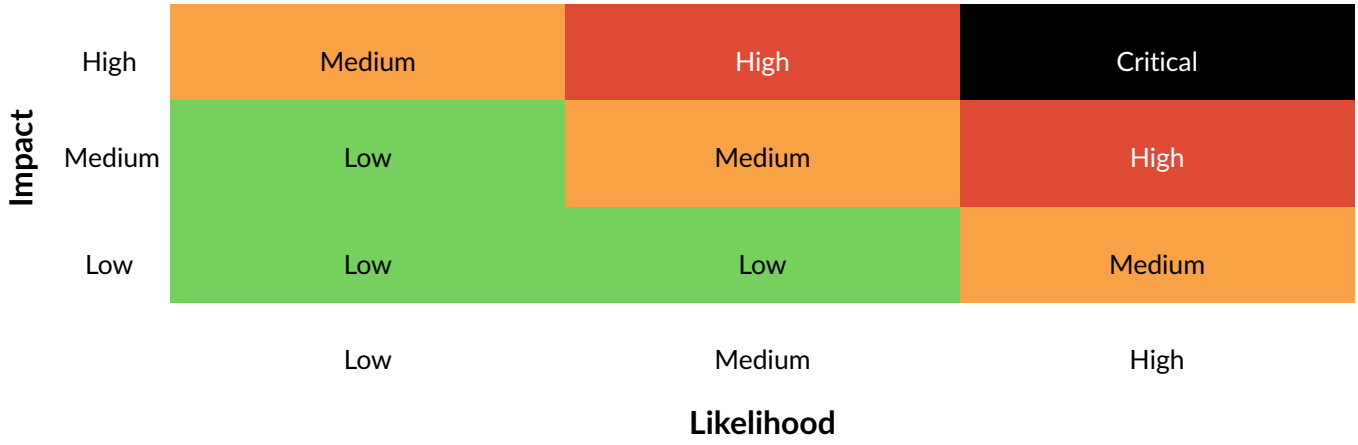

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].