EigenLayer

EigenLayer Sidecar Review

Version: 2.0

December, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the EigenLayer sidecar. 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 EigenLayer 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*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the EigenLayer components in scope.

Overview

The EigenLayer Sidecar is an open source, permissionless, verified indexer enabling anyone (AVS, operator, etc) to access EigenLayer's protocol rewards in real-time.

It aims to replace the centralized rewards data pipeline that EigenLabs currently runs and provide rewards data to consumers to generate roots, generate claims and attribute rewards to AVSs, operators, and strategies.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to files at commit 1de6ecb.

The fixes of the identified issues were assessed at commit ba61e84.

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

Approach

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`

Furthermore, the testing team leveraged fuzz testing techniques (i.e. fuzzing), which is a process allowing the identification of bugs by providing randomised and unexpected data inputs to software with the purpose of causing crashes and other unexpected behaviours (e.g. broken invariants, memory exhaustion, infinite / extended loops).

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 24 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.

- High: 4 issues.
- Medium: 6 issues.
- Low: 6 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the EigenLayer 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
ELSC-01	Indexing Beyond `safe` Blocks During Initial Sync	Critical	Resolved
ELSC-02	Race Condition In Concurrent RPC Fetching	Critical	Resolved
ELSC-03	Corrupted State Not Reprocessed After Deletion	Critical	Resolved
ELSC-04	Incorrect Root Index Parsing Due To Wrong Argument Reference	Critical	Resolved
ELSC-05	Timing Attack On Operator-AVS-Strategy Snapshots	High	Closed
ELSC-06	Missing Validation Of Restaked Strategies Leading To Incorrect Reward Calculations	High	Closed
ELSC-07	Silent Failure In `getOperatorRestakedStrategiesRetryable()`	High	Resolved
ELSC-08	Incorrect Error Handling In `RunForFetchedBlock()` Function	High	Resolved
ELSC-09	Incorrect Sorting Of Distribution Root Models	Medium	Resolved
ELSC-10	Lack Of Error Handling In `getAbi()`	Medium	Resolved
ELSC-11	Inefficient Leaf Encoding Using String Representations	Medium	Closed
ELSC-12	Incorrect `pctComplete` Calculation & Potential Division By Zero In `IndexFromCurrentToTip()`	Medium	Resolved
ELSC-13	Unhandled Errors When Decoding Logs & Calldata	Medium	Resolved
ELSC-14	Address Matching In `IsInterestingAddress()` Is Case Sensitive	Medium	Resolved
ELSC-15	Panic When Latest State Root Matches Last Indexed Block	Low	Resolved
ELSC-16	Lexicographic Sorting Of `SlotID`	Low	Resolved
ELSC-17	Missing JSON Tags In `genericRewardPaymentData` Struct	Low	Resolved
ELSC-18	Use Of Deprecated Package `xerrors`	Low	Resolved
ELSC-19	Unsafe Integer Comparison In Sort Functions	Low	Closed
ELSC-20	Merkle Tree Shrinking Attack In State Root Generation	Low	Resolved
ELSC-21	Restaked Strategies Indexing Interval Is Too Long	Informational	Closed
ELSC-22	Lack Of Context Cancellation	Informational	Resolved
ELSC-23	Missing Error Propagation In `GetLastIndexedBlock()`	Informational	Resolved
ELSC-24	Miscellaneous General Comments	Informational	Closed

ELSC-01	Indexing Beyond `safe` Blocks During Initial Sync		
Asset	`blockIndexer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `IndexFromCurrentToTip()` function indexes blocks up to the latest block number instead of the latest justified/safe block, which could lead to processing blocks that may get re-orged out of the chain. This creates inconsistent state and requires clean up.

The issue occurs because the function uses `eth_blockNumber` RPC call to get the target block number to sync up to:

blockIndexer.go::IndexFromCurrentToTip()

```
blockNumber, err := s.EthereumClient.GetBlockNumberUint64(ctx)
```

This returns the latest block number, not considering finality status. The function then proceeds to index all blocks up to this number:

blockIndexer.go::IndexFromCurrentToTip()

```
currentTip := atomic.Uint64{}
currentTip.Store(blockNumber)

// ...

for uint64(latestBlock) <= currentTip.Load() {
    // ...
}
```

This contrasts with `ProcessNewBlocks()` which correctly uses `GetLatestSafeBlock()` to only process justified blocks. If unsafe blocks were to be re-orged out of the chain, sidecar would keep track of incorrect state which would not be able to be cleaned up as there is no logic to handle re-orgs.

Recommendations

Modify `IndexFromCurrentToTip()` to use `GetLatestSafeBlock()` instead of `GetBlockNumberUint64()`.

Alternatively, query for the last finalised block instead of the last justified block by querying `eth_getBlockByNumber` with the string `"finalized"` instead of `"safe"`. Keep in mind to also change `ProcessNewBlocks()` to get the finalised block instead of the justified block.

Resolution

The EigenLayer team has implemented the first recommendation by using `GetLatestSafeBlock()`.

This issue has been resolved in commit 2effb51.

ELSC-02	Race Condition In Concurrent RPC Fetching		
Asset	`fetcher.go, client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Multiple goroutines are concurrently writing to shared variables without proper synchronisation, leading to race conditions that could cause data corruption or inconsistent results.

In `fetcher.go`, the `FetchBlocks()` function launches multiple goroutines that concurrently write to the shared `fetchedBlocks` slice and `foundErrors` boolean without any mutex protection:

fetcher.go::FetchBlocks()

```go
fetchedBlocks := make([]*FetchedBlock, 0)
foundErrors := false
wg := sync.WaitGroup{}
for _, block := range blocks {
  wg.Add(1)
  go func(b *ethereum.EthereumBlock) {
      defer wg.Done()
      receipts, err := f.FetchReceiptsForBlock(ctx, b)
      if err != nil {
          f.Logger.Sugar().Errorw("failed to fetch receipts for block",
              zap.Uint64("blockNumber", b.Number.Value()),
              zap.Error(err),
          )
          // @audit race condition in `foundErrors`
          foundErrors = true
          return
      }
      // @audit race condition in `fetchedBlocks`
      fetchedBlocks = append(fetchedBlocks, &FetchedBlock{
          Block:      b,
          TxReceipts: receipts,
      })
  }(block)
}
```

Similarly, in `client.go`, the `results` slice is being appended to concurrently without synchronisation:

client.go::ChunkedNativeBatchCall()

```go
results := []*RPCResponse{}
wg := sync.WaitGroup{}
for i, batch := range batches {
  wg.Add(1)

  go func(b []*RPCRequest) {
      defer wg.Done()

      c.Logger.Sugar().Debugw(fmt.Sprintf("[batch %d] Fetching batch with '%d' requests", i, len(b)))
      res, err := c.batchCall(ctx, b)
      if err != nil {
          c.Logger.Sugar().Errorw("failed to batch call", zap.Error(err))
          return
      }
      c.Logger.Sugar().Debugw(fmt.Sprintf("[batch %d] Received '%d' results", i, len(res)))
      // @audit race condition in `results`
      results = append(results, res...)
  }(batch)
}
```

These race conditions could lead to:

1. Lost updates to the slices when multiple goroutines append simultaneously
2. Inconsistent state of the `foundErrors` flag
3. Potential panic conditions if slice capacity needs to be increased concurrently

Recommendations

Use a mutex in both `fetcher.go` and `client.go` cases to protect access to shared variables.

Alternatively, consider using channels to collect results from goroutines, which would provide natural synchronisation without explicit locks.

Resolution

The EigenLayer team has implemented channels to collect results from goroutines in `fetcher.go` and `client.go` as suggested in the alternative recommendation.

This issue has been resolved in commit 45fb121.

ELSC-03	Corrupted State Not Reprocessed After Deletion		
Asset	`blockIndexer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When corrupted state is detected and deleted, the affected blocks are not reprocessed, leading to permanent gaps in the indexed data.

In the `IndexFromCurrentToTip()` function, when the latest state root is behind the latest block, the code correctly identifies this as corrupted state and deletes it. However, while the comment indicates that `latestBlock` should be set to `latestStateRoot.EthBlockNumber + 1` to reprocess the corrupted blocks, this assignment is never actually performed. Instead, the code continues with the original `latestBlock` value, effectively skipping over all the blocks that were just deleted.

blockIndexer.go::IndexFromCurrentToTip()

```go
// if the latest state root is behind the latest block, delete the corrupted state and set the
// latest block to the latest state root + 1
if latestStateRoot != nil && latestStateRoot.EthBlockNumber < uint64(latestBlock) {
    s.Logger.Sugar().Infow("Latest state root is behind latest block, deleting corrupted state",
        zap.Uint64("latestStateRoot", latestStateRoot.EthBlockNumber),
        zap.Int64("latestBlock", latestBlock),
    )
    if err := s.StateManager.DeleteCorruptedState(latestStateRoot.EthBlockNumber+1, uint64(latestBlock)); err != nil {
        s.Logger.Sugar().Errorw("Failed to delete corrupted state", zap.Error(err))
        return err
    }
    if err := s.Storage.DeleteCorruptedState(uint64(latestStateRoot.EthBlockNumber+1), uint64(latestBlock)); err != nil {
        s.Logger.Sugar().Errorw("Failed to delete corrupted state", zap.Error(err))
        return err
    }
    // @audit Missing `latestBlock = latestStateRoot.EthBlockNumber + 1`
}
```

Recommendations

Add the missing assignment after deleting the corrupted state.

blockIndexer.go::IndexFromCurrentToTip()

```go
if latestStateRoot != nil && latestStateRoot.EthBlockNumber < uint64(latestBlock) {
    // ...
    if err := s.Storage.DeleteCorruptedState(uint64(latestStateRoot.EthBlockNumber+1), uint64(latestBlock)); err != nil {
        s.Logger.Sugar().Errorw("Failed to delete corrupted state", zap.Error(err))
        return err
    }
    // Add this line to ensure deleted blocks are reprocessed
    latestBlock = int64(latestStateRoot.EthBlockNumber + 1)
}
```

Resolution

`latestBlock` is now set to `latestStateRoot.EthBlockNumber + 1` to reprocess the corrupted blocks as suggested in the recommendation above.

This issue has been resolved in commit 6d9d283.

ELSC-04	Incorrect Root Index Parsing Due To Wrong Argument Reference		
Asset	`submittedDistributionRoots.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `GetStateTransitions()` function incorrectly parses the `rootIndex` from the wrong event argument, leading to incorrect state transitions and corrupted data in the system.

The event to be parsed in `submittedDistributionRoots.go` is:

IRewardsCoordinator.sol

```
event DistributionRootSubmitted(
    uint32 indexed rootIndex, // arguments[0]
    bytes32 indexed root, // arguments[1]
    uint32 indexed rewardsCalculationEndTimestamp, // arguments[2]
    uint32 activatedAt
);
```

Instead of using `arguments[0]` (the correct argument for `rootIndex`), the code mistakenly uses `arguments[2]` which contains the `rewardsCalculationEndTimestamp` as shown below:

submittedDistributionRoots.go::GetStateTransitions()

```
var rootIndex uint64

t := reflect.TypeOf(arguments[0].Value)
switch t.Kind() {
case reflect.String:
    if arguments[0].Value.(string) == "0x0000000000000000000000000000000000000000000" {
        rootIndex = 0
        break
    }
    // @audit `rootIndex` is parsed from `arguments[2]`
    withoutPrefix := strings.TrimPrefix(arguments[2].Value.(string), "0x")
    rootIndex, err = strconv.ParseUint(withoutPrefix, 16, 32)
    if err != nil {
        return nil, xerrors.Errorf("Failed to decode rootIndex: %v", err)
    }
case reflect.Float64:
    rootIndex = uint64(arguments[0].Value.(float64))
default:
    return nil, xerrors.Errorf("Invalid type for rootIndex: %s", t.Kind())
}
```

This means the system is storing distribution roots with incorrect indices, ultimately leading to incorrect state transitions and reward calculations.

Recommendations

Update the code to use the correct argument index for parsing the `rootIndex`:

```
withoutPrefix := strings.TrimPrefix(arguments[0].Value.(string), "0x")
rootIndex, err = strconv.ParseUint(withoutPrefix, 16, 32)
```

Resolution

The EigenLayer team has implemented the recommendation above.

This issue has been resolved in commit a00cc4dc.

ELSC-05	Timing Attack On Operator-AVS-Strategy Snapshots		
Asset	`operatorAvsStrategySnapshots.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Operators can exploit the snapshot mechanism to maintain rewards while minimising slashing exposure by strategically registering and deregistering around snapshot times. This undermines the security model of the protocol by allowing operators to avoid intended slashing risks while still collecting rewards.

The current implementation in `operatorAvsStrategySnapshots.go` only considers gaps in registration when they span more than one day. This is evident in the `parsed_ranges` CTE:

operatorAvsStrategySnapshots.go

```
parsed_ranges AS (
    SELECT
        operator,
        avs,
        strategy,
        start_time,
        CASE
            WHEN next_start_time IS NULL OR next_start_time > start_time + INTERVAL '1' DAY THEN start_time
            ELSE next_start_time
        END AS end_time
    FROM grouped_records
),
```

The vulnerability is exacerbated by two key factors:

1. Strategies are only indexed every 3600 blocks (≥12 hours) as shown in `pipeline.go`:

   pipeline.go::RunForFetchedBlock()

   ```
   if block.Block.Number.Value()%3600 == 0 {
       p.Logger.Sugar().Infow("Indexing OperatorRestakedStrategies", zap.Uint64("blockNumber", block.Block.Number.Value()))
       if err := p.Indexer.ProcessRestakedStrategiesForBlock(ctx, block.Block.Number.Value()); err != nil {
           p.Logger.Sugar().Errorw("Failed to process restaked strategies", zap.Uint64("blockNumber",
               ↪ block.Block.Number.Value()), zap.Error(err))
           return err
       }
   }
   ```

2. The middleware places no meaningful restrictions on registering/deregistering from quorums by default, which determine which strategies an operator is exposed to.

An operator can exploit this by:

1. Registering to many AVS quorums shortly before the daily snapshot
2. Deregistering after the snapshot
3. Repeating this process every ~3600 blocks (snapshot interval)

Additionally, operators could register every 7200 blocks, and despite actual time gaps exceeding 24 hours due to potentially missed slots, the rounding up of block times to the next day means these gaps won't be detected.

Recommendations

Add restrictions or cooldown periods for quorum registration/deregistration to prevent rapid switching. The cooldown should be greater than a day to mitigate against timing attacks.

Resolution

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

> *"This [issue] will be officially fixed in the slashing release as will we no longer need to make contract calls to get operator restaked strategies."*

ELSC-06	Missing Validation Of Restaked Strategies Leading To Incorrect Reward Calculations		
Asset	`restakedStrategies.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In `restakedStrategies.go`, within the `getAndInsertRestakedStrategies()` function, there is no validation to ensure that the number of restaked strategies retrieved matches the number of AVS-operator pairs requested. If `GetAllOperatorRestakedStrategies()` returns fewer results than the number of pairs requested, the function proceeds without error. This can result in missing strategies for some AVS-operator pairs, leading to incorrect reward calculations.

The `getAndInsertRestakedStrategies()` function prepares a list of valid operator-AVS pairs then attempts to get the operator's restaked strategies.

restakedStrategies.go::getAndInsertRestakedStrategies()

```go
pairs := make([]*contractCaller.OperatorRestakedStrategy, 0)
for _, avsOperator := range avsOperators {
    if avsOperator == nil || avsOperator.Operator == "" || avsOperator.Avs == "" {
        return fmt.Errorf("Invalid AVS operator - %v", avsOperator)
    }
    pairs = append(pairs, &contractCaller.OperatorRestakedStrategy{
        Operator: avsOperator.Operator,
        Avs:      avsOperator.Avs,
    })
}

results, err := idx.ContractCaller.GetAllOperatorRestakedStrategies(ctx, pairs, blockNumber)
if err != nil {
    idx.Logger.Sugar().Errorw("Failed to get operator restaked strategies",
        zap.Error(err),
        zap.String("avsDirectoryAddress", avsDirectoryAddress),
        zap.Uint64("blockNumber", blockNumber),
    )
    return err
}
```

`contractCaller.go` is responsible for fetching restaked strategies. The key function here is `getOperatorRestakedStrategiesBatch()`, which processes multiple operator-AVS pairs concurrently using goroutines.

contractCaller/sequentialContractCaller/contractCaller.go::getOperatorRestakedStrategiesBatch()

```go
for _, operatorRestakedStrategy := range operatorRestakedStrategies {
        wg.Add(1)
        // make a local copy of the entire struct
        currentReq := *operatorRestakedStrategy
        go func() {
            defer wg.Done()
            results, err := getOperatorRestakedStrategiesRetryable(ctx, currentReq.Avs, currentReq.Operator, blockNumber,
                ↪  cc.EthereumClient, cc.Logger)
            if err != nil {
                cc.Logger.Sugar().Errorw("getOperatorRestakedStrategiesBatch - failed to get results",
                    zap.String("avs", currentReq.Avs),
                    zap.String("operator", currentReq.Operator),
                    zap.Uint64("blockNumber", blockNumber),
                    zap.Error(err),
                )
                return
            }
            currentReq.Results = results

            // send back a pointer to the copied struct
            responses <- &currentReq
        }()
    }
    wg.Wait()
    close(responses)

    allResponses := make([]*contractCaller.OperatorRestakedStrategy, 0)
    for response := range responses {
        allResponses = append(allResponses, response)
    }
```

Each goroutine calls `getOperatorRestakedStrategiesRetryable()`. If `getOperatorRestakedStrategiesRetryable()` returns an error, the goroutine logs the error and returns early without sending anything to the responses channel.

This means that the number of responses collected from the responses channel can be less than the number of requests (AVS-operator pairs), while receiving no indication of any errors that may have occurred.

This leads to potentially missing strategies for some operators, causing incorrect reward calculations.

Recommendations

Add a validation check after fetching the restaked strategies to ensure data completeness:

```go
if len(results) != len(pairs) {
    errMsg := fmt.Sprintf("Expected %d results but got %d", len(pairs), len(results))
    return fmt.Errorf(errMsg)
}
```

Resolution

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

> *"This behaviour is intended as it relies on calling a contract implemented and deployed by the AVS. If the contract does not conform to the expected interface, they will not be included in the rewards calculation process. This issue will be officially fixed in the slashing release as we will no longer need to make contract calls to get operator restaked strategies."*

ELSC-07	Silent Failure In `getOperatorRestakedStrategiesRetryable()`		
Asset	`contractCaller/sequentialContractCaller/contractCaller.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `getOperatorRestakedStrategiesRetryable()` function incorrectly returns `(nil, nil)` after exhausting all retry attempts, effectively hiding failures in retrieving operator restaking strategies. This creates ambiguity between reverted calls which also return `(nil, nil)` and failed calls due to an RPC error, resulting in less indexed restaking strategies than intended.

contractCaller/sequentialContractCaller/contractCaller.go::getOperatorRestakedStrategiesRetryable()

```go
func getOperatorRestakedStrategiesRetryable(...) ([]common.Address, error) {
    retries := []int{0, 2, 5, 10, 30}
    for i, backoff := range retries {
        results, err := getOperatorRestakedStrategies(...)
        if err != nil {
            // ... logging ...
            time.Sleep(time.Second * time.Duration(backoff))
        } else {
            return results, nil
        }
    }
    return nil, nil  // @audit silently suppressing errors
}
```

Due to this ambiguity, it is possible for operators to be indexed with less strategies than intended, resulting in incorrect reward calculations.

Recommendations

Consider returning an error after exhausting all retry attempts.

Resolution

The EigenLayer team has implemented the recommended fix by returning a new `ErrRetriesExceeded` error if all retries are exhausted.

This issue has been resolved in commit 16f291b.

ELSC-08	Incorrect Error Handling In `RunForFetchedBlock()` Function		
Asset	`pipeline.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `RunForFetchedBlock()` function incorrectly returns `err` instead of `ierr` when `ParseInterestingTransactionsAndLogs()` fails, causing the pipeline to silently continue execution when it should halt. This leads to blocks being silently skipped during processing, resulting in missing transactions, corrupted state, and incorrect rewards calculation.

The function returns `err` instead of `ierr` which contains the actual error from `ParseInterestingTransactionsAndLogs()`. This causes `nil` to be returned, as `err != nil` is checked beforehand:

pipeline.go::RunForFetchedBlock()

```
indexedBlock, found, err := p.Indexer.IndexFetchedBlock(block)
// @audit err is handled properly here
if err != nil {
    p.Logger.Sugar().Errorw("Failed to index block", zap.Uint64("blockNumber", blockNumber), zap.Error(err))
    return err
}
if found {
    p.Logger.Sugar().Infow("Block already indexed", zap.Uint64("blockNumber", blockNumber))
}
p.Logger.Sugar().Debugw("Indexed block",
    zap.Uint64("blockNumber", blockNumber),
    zap.Int64("indexTime", time.Since(blockFetchTime).Milliseconds()),
)

blockFetchTime = time.Now()

// Parse all transactions and logs for the block.
// - If a transaction is not calling to a contract, it is ignored
// - If a transaction has 0 interesting logs and itself is not interesting, it is ignored
parsedTransactions, ierr := p.Indexer.ParseInterestingTransactionsAndLogs(ctx, block)
if ierr != nil {
    p.Logger.Sugar().Errorw("Failed to parse transactions and logs",
        zap.Uint64("blockNumber", blockNumber),
        zap.String("transactionHash", ierr.TransactionHash),
        zap.Error(ierr.Err),
    )
    // @audit err will always be nil here as it's properly handled above
    return err
}
```

Recommendations

Update the error return statement to use the correct error variable.

Resolution

The EigenLayer team has implemented the recommendation by returning `ierr` instead of `err`.

This issue has been resolved in commit e04f272.

ELSC-09	Incorrect Sorting Of Distribution Root Models		
Asset	`disabledDistributionRoots.go, submittedDistributionRoots.go, baseEigenState.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `DisabledDistributionRootsModel` and `SubmittedDistributionRootsModel` fail to properly sort state entries by `SlotID` before merkleization, which will cause the merkle tree generation to fail with the error "*slotIDs are not in order*".

In `sortValuesForMerkleTree()`, the values are sorted by `RootIndex` before being converted to `MerkleTreeInput` entries with `SlotID` s as shown below in both the `DisabledDistributionRootsModel` and `SubmittedDistributionRootsModel` :

disabledDistributionRoots.go::sortValuesForMerkleTree()

```go
func (ddr *DisabledDistributionRootsModel) sortValuesForMerkleTree(inputs []*types.DisabledDistributionRoot)
     ↪  []*base.MerkleTreeInput {
    slices.SortFunc(inputs, func(i, j *types.DisabledDistributionRoot) int {
        return int(i.RootIndex - j.RootIndex)  // @audit Sorting by RootIndex instead of SlotID
    })
    // ...
}
```

submittedDistributionRoots.go::sortValuesForMerkleTree()

```go
func (sdr *SubmittedDistributionRootsModel) sortValuesForMerkleTree(inputs []types.SubmittedDistributionRoot)
     ↪  []*base.MerkleTreeInput {
    slices.SortFunc(inputs, func(i, j types.SubmittedDistributionRoot) int {
        return int(i.RootIndex - j.RootIndex)
    })

    values := make([]*base.MerkleTreeInput, 0)
    for _, input := range inputs {
        values = append(values, &base.MerkleTreeInput{
            SlotID: base.NewSlotID(input.TransactionHash, input.LogIndex),
            Value:  []byte(input.Root),
        })
    }
    return values
}
```

However, `MerkleizeState()` explicitly requires inputs to be sorted by `SlotID` in ascending order, enforcing this with a check that returns an error if they're not properly ordered.

baseEigenState.go::MerkleizeState()

```go
func (b *BaseEigenState) MerkleizeState(blockNumber uint64, inputs []*MerkleTreeInput) (*merkletree.MerkleTree, error) {
    om := orderedmap.New[types.SlotID, []byte]()

    for _, input := range inputs {
    _, found := om.Get(input.SlotID)
    if !found {
        om.Set(input.SlotID, input.Value)

        prev := om.GetPair(input.SlotID).Prev()
        // @audit inputs ordering is checked here
        if prev != nil && prev.Key > input.SlotID {
            om.Delete(input.SlotID)
            return nil, errors.New("slotIDs are not in order")
        }
    } else {
        return nil, fmt.Errorf("duplicate slotID %s", input.SlotID)
    }
    // ...
}
```

This mismatch in sorting criteria means that if multiple distribution roots are submitted or disabled in a block, it is extremely likely that the merkleisation will fail.

This issue has a low likelihood of occurring as distribution roots are currently submitted on a weekly basis, and disabling distribution roots is only done in extreme cases.

Recommendations

Consider modifying the `sortValuesForMerkleTree()` functions in both `DisabledDistributionRootsModel` and `SubmittedDistributionRootsModel` to sort by `SlotID` instead of `RootIndex`.

Resolution

The EigenLayer team has implemented the recommended fix by sorting by `SlotID` in both `DisabledDistributionRootsModel` and `SubmittedDistributionRootsModel`.

This issue has been resolved in commit 16f291b.

ELSC-10	Lack Of Error Handling In `getAbi()`		
Asset	`transactionLogs.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `getAbi()` function fails to properly handle errors during `ABI` unmarshaling, instead returning an invalid `ABI` object even when parsing fails. This can lead to incorrect or empty log decoding downstream since callers assume the `ABI` was parsed successfully.

The function `getAbi()` does not return any errors:

transactionLogs.go::getAbi()

```go
func (idx *Indexer) getAbi(json string) (*abi.ABI, error) {
    a := &abi.ABI{}

    err := a.UnmarshalJSON([]byte(json))

    if err != nil {
        foundMatch := false
        patterns := []*regexp.Regexp{
            regexp.MustCompile(`only single receive is allowed`),
            regexp.MustCompile(`only single fallback is allowed`),
        }

        for _, pattern := range patterns {
            if pattern.MatchString(err.Error()) {
                foundMatch = true
                break
            }
        }
        // Ignore really common compilation error
        if !foundMatch {
            idx.Logger.Sugar().Warnw("Error unmarshaling abi json", zap.Error(err))
        }
    }

    return a, nil
}
```

If `a.UnmarshalJSON()` fails, the function ignores the error (except for logging a warning) and returns `(a, nil)` where `a` may be in an invalid state. This means the caller will assume that the ABI is parsed successfully, potentially leading to incorrect or empty decoding later.

Consider the case where `a` is an empty `ABI`, in the `DecodeLog()` function `decodedLog` will have an empty `EventName`, `Arguments` and `OutputData`.

transactionLogs.go::DecodeLog()

```go
// @audit decodedLog is created with empty EventName, Arguments and OutputData
decodedLog := &parser.DecodedLog{
    Address:  logAddress.String(),
    LogIndex: lg.LogIndex.Value(),
}

if a == nil {
    idx.Logger.Sugar().Debugw(fmt.Sprintf("No ABI provided, using topic hash as event name '%s'", logAddress.String()))
    decodedLog.EventName = topicHash.String()
    return decodedLog, nil
}
// @audit this method will error out as `a` is an empty ABI
event, err := a.EventByID(topicHash)
if err != nil {
    // @audit the function will error out here and return with the incomplete decodedLog
    idx.Logger.Sugar().Debugw(fmt.Sprintf("Failed to find event by ID '%s'", topicHash))
    return decodedLog, err
}
```

Since the `decodedLog` does not have `EventName`, `Arguments` and `OutputData`, it will not be counted as an "interesting log" and will not be processed through any state models, potentially resulting in an incorrect state transition and reward calculation.

Recommendations

Consider returning the error from `getAbi()` instead of ignoring it.

Resolution

The EigenLayer team has implemented a fix by returning the error from `getAbi()`, if there is no regex match with the acceptable errors.

This issue has been resolved in commit 6d066b.

ELSC-11	Inefficient Leaf Encoding Using String Representations		
Asset	`eigenState/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

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

Multiple components are affected by this string encoding pattern:

1. Block number and block hash leaves in `GenerateStateRoot()` in `stateManager.go`:

stateManager.go::GenerateStateRoot()

```
roots := [][]byte{
    []byte(fmt.Sprintf("%d", blockNumber)), // @audit blocKNumber is a string in decimal representation
    []byte(blockHash), // @audit blockHash is a string with 0x prefix
}
```

2. Individual model state roots in `GenerateStateRoot()`, where the model root is returned as a string:

stateManager.go::GenerateStateRoot()

```
// @audit the model root is returned as a string with 0x prefix
return types.StateRoot(utils.ConvertBytesToString(fullTree.Root())), nil
```

3. Individual model leaves, for example in `DisabledDistributionRoots`:

disabledDistributionRoots.go::HandleStateChange()

```
values = append(values, &base.MerkleTreeInput{
    // @audit SlotID is a string in the format "txHash_logIndex"
    SlotID: base.NewSlotID(input.TransactionHash, input.LogIndex),
    // @audit Value is a
    Value:  []byte(fmt.Sprintf("%d", input.RootIndex)),
})
```

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. Use the `geth` library to represent Ethereum types instead of using `string`.

1. For block numbers, use big-endian encoding and `common.FromHex()`:

stateManager.go::GenerateStateRoot()

```go
roots := [][]byte{
    binary.BigEndian.AppendUint64([]byte{}, blockNumber),
    common.FromHex(blockHash),
}
```

2. For model roots, process them as raw bytes:

disabledDistributionRoots.go::GenerateStateRoot()

```go
func (ddr *DisabledDistributionRootsModel) GenerateStateRoot(blockNumber uint64) ([]bytes, error) {
    // ...
    return fullTree.Root(), nil
}
```

stateManager.go::encodeModelLeaf()

```go
func (e *EigenStateManager) encodeModelLeaf(model types.IEigenStateModel, blockNumber uint64) ([]byte, error) {
    // root is now in raw bytes
    root, err := model.GenerateStateRoot(blockNumber)
    // ....
    return append([]byte(model.GetModelName()), root...), nil
}
```

3. For model leaves, instead of encoding the `SlotID` as an underscore-separated string, the fields can be encapsulated in a struct. `Value` can be encoded directly as raw bytes instead of as a string. To streamline this, model delta structs should also use Ethereum types where appropriate such as using `common.Address` to represent Ethereum addresses instead of `string`.

Resolution

The EigenLayer team has implemented the recommended fixes for block number, block hash leaves and individual model state roots in commit e05059f.

The issue for model leaves described above has been acknowledged 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."*

ELSC-12	Incorrect `pctComplete` Calculation & Potential Division By Zero In `IndexFromCurrentToTip()`		
Asset	`blockIndexer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `IndexFromCurrentToTip()` function in `blockIndexer.go` contains issues related to the calculation of `pctComplete`. Specifically, the method incorrectly calculates the percentage of blocks processed and does not adequately handle scenarios where the number of remaining blocks is zero. This can lead to inaccurate progress reporting and runtime panics due to division by zero.

The percentage completion `pctComplete` and `runningAvg` is calculated as follows:

blockIndexer.go::IndexFromCurrentToTip()

```
pctComplete := (float64(blocksProcessed) / float64(blocksRemaining)) * 100
// ...
runningAvg = float64(totalDurationMs / blocksProcessed)
```

The calculation is incorrect as it divides `blocksProcessed` by `blocksRemaining` instead of the total number of blocks to process. This leads to inaccurate progress reporting.

There is also a potential division by zero vulnerability when `blocksRemaining` is zero, which occurs when `latestBlock` has caught up to `currentTip`. In this case, the calculation would result in a runtime panic due to division by zero.

Additionally, if no new blocks are processed during a loop iteration (for example, when `latestBlock` has caught up to `currentTip`), `blocksProcessed` remains zero. This creates another division by zero scenario that could cause the application to crash.

Recommendations

The calculation should be modified to divide `blocksProcessed` by the total number of blocks to process, rather than the remaining blocks. This will provide an accurate percentage completion value.

Additionally, checks should be implemented to ensure that `blocksProcessed` is not zero before performing any division operations. This will prevent potential runtime panics from division by zero scenarios when no new blocks have been processed in a loop iteration.

Resolution

The EigenLayer team has implemented the recommended fixes above.

This issue has been resolved in commits 9cf23d5 and c3ebec5.

ELSC-13	Unhandled Errors When Decoding Logs & Calldata		
Asset	`transactionLogs.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When decoding transaction input data and log data, errors are not properly handled, resulting in invalid or empty decoded data from being stored and processed.

This issue is present when decoding transaction input data:

transactionLogs.go::ParseTransactionLogs()

```go
if idx.IsInterestingAddress(contractAddress.Value()) {
    contract, err := idx.ContractManager.GetContractWithProxy(contractAddress.Value(), transaction.BlockNumber.Value())
    // ...

    if contract == nil {
        // @audit no error returned if no contract found for an interesting address
        idx.Logger.Sugar().Debugw("No contract found for address", zap.String("hash", transaction.Hash.Value()))
        return nil, nil
    }

    contractAbi := contract.CombineAbis()
    // ...

    if contractAbi == "" {
        // @audit no error returned if no ABI found for an interesting address
        idx.Logger.Sugar().Debugw("No ABI found for contract", zap.String("hash", transaction.Hash.Value()))
        return parsedTransaction, nil
    }
    // ...
}
```

transactionLogs.go::ParseTransactionLogs()

```go
if len(txInput) >= 10 {
    var method *abi.Method
    decodedSig, err := hex.DecodeString(txInput[2:10])
    if err != nil {
        // @audit error is logged but not propagated if cannot decode function sig
        idx.Logger.Sugar().Errorw("Failed to decode signature")
    }

    if len(decodedSig) > 0 {
        method, err = a.MethodById(decodedSig)
        if err != nil {
            // @audit error is logged but not propagated if cannot find method by ID
            idx.Logger.Sugar().Debugw(fmt.Sprintf("Failed to find method by ID '%s'", common.BytesToHash(decodedSig).String()))
            parsedTransaction.MethodName = "unknown"
        } else {
            parsedTransaction.MethodName = method.RawName
            decodedData, err := hex.DecodeString(txInput[10:])
            if err != nil {
                // @audit error is logged but not propagated if cannot decode input data
                idx.Logger.Sugar().Errorw("Failed to decode input data")
            } else {
                callMap := map[string]interface{}{}
                if err := method.Inputs.UnpackIntoMap(callMap, decodedData); err != nil {
                    // @audit error is logged but not propagated if cannot unpack data
                    idx.Logger.Sugar().Errorw("Failed to unpack data",
                        zap.Error(err),
                        zap.String("transactionHash", transaction.Hash.Value()),
                        zap.Uint64("blockNumber", transaction.BlockNumber.Value()),
                    )
                }
                callMapBytes, err := json.Marshal(callMap)
                if err != nil {
                    // @audit error is logged but not propagated if cannot marshal data
                    idx.Logger.Sugar().Errorw("Failed to marshal callMap data", zap.String("hash", transaction.Hash.Value()))
                }
                // @audit invalid/empty decoded data is stored
                parsedTransaction.DecodedData = string(callMapBytes)
            }
        }
    }
}
```

The same issue is present when decoding logs:

transactionLogs.go::ParseTransactionLogs()

```go
for i, lg := range receipt.Logs {
    if !idx.IsInterestingAddress(lg.Address.Value()) {
        continue
    }
    decodedLog, err := idx.DecodeLogWithAbi(a, receipt, lg)
    if err != nil {
        // @audit error is logged but not propagated if cannot decode log
        idx.Logger.Sugar().Debugw(fmt.Sprintf("Error decoding log - index: '%d' - '%s'", i, transaction.Hash.Value()),
            ↪  zap.Error(err))
    } else {
        idx.Logger.Sugar().Debugw(fmt.Sprintf("Decoded log - index: '%d' - '%s'", i, transaction.Hash.Value()),
            ↪  zap.Any("decodedLog", decodedLog))
    }

    // @audit invalid/empty decoded log is stored
    logs = append(logs, decodedLog)
}
```

transactionLogs.go::DecodeLog()

```go
if len(lg.Topics) > 1 {
    for i, param := range lg.Topics[1:] {
        d, err := ParseLogValueForType(event.Inputs[i], param.Value())
        if err != nil {
            // @audit error is logged but not propagated if cannot parse log value for type
            idx.Logger.Sugar().Errorw("Failed to parse log value for type", zap.Error(err))
        } else {
            decodedLog.Arguments[i].Value = d
        }
    }
}

if len(lg.Data) > 0 {
    // ...

    outputDataMap := make(map[string]interface{})
    err = a.UnpackIntoMap(outputDataMap, event.Name, byteData)
    if err != nil {
        // @audit error is logged but not propagated if cannot unpack log data
        idx.Logger.Sugar().Errorw("Failed to unpack data: ",
            zap.Error(err),
            zap.String("hash", lg.TransactionHash.Value()),
            zap.String("address", lg.Address.Value()),
            zap.String("eventName", event.Name),
            zap.String("transactionHash", lg.TransactionHash.Value()),
        )
    }
    // @audit invalid/empty decoded log data is stored
    decodedLog.OutputData = outputDataMap
}
```

Furthermore, when unable to get the `ABI` for the contract that emitted the log, the error is not propagated and `idx.DecodeLog()` is called without an `ABI`, resulting in an incomplete decoded log.

transactionLogs.go::DecodeLogWithAbi()

```go
// @audit There are 5 instances of the pattern below in the same function.
//        Only one is shown here for brevity.

// Find/create the log address and attempt to determine if it is a proxy address
foundOrCreatedContract, err := idx.ContractManager.GetContractWithProxy(logAddress.String(), txReceipt.BlockNumber.Value())
if err != nil {
    return idx.DecodeLog(nil, lg)
}
```

This issue has a high impact as it could lead to incorrect reward calculations if a log/transaction that is supposed to be decoded is incorrectly decoded or skipped. It's important for `sidecar` to halt execution whenever it encounters a log or transaction from or to an EigenLayer contract that cannot be decoded, as it is likely for these to be "interesting" and processed by state models.

Recommendations

Consider propagating the errors up the call stack so that `sidecar` halts execution whenever it encounters a log or transaction from or to an EigenLayer contract that cannot be decoded.

Furthermore, instead of attempting to decode the log without an `ABI` when the `ABI` cannot be fetched, also consider propagating the error up the call stack so that `sidecar` halts execution.

Resolution

The EigenLayer team has propagated any errors up the call stack as recommended.

This issue has been resolved in commit c7fa20d.

ELSC-14	Address Matching In `IsInterestingAddress()` Is Case Sensitive		
Asset	`transactionLogs.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `IsInterestingAddress()` function is case sensitive, such that if ERC-55 checksummed addresses are used, the function may not correctly identify the address as interesting.

The `IsInterestingAddress()` function is used to determine if a transaction target or log emitter is an EigenLayer contract. It uses the `slices.Contains()` function to check if the address is in the list of interesting addresses:

indexer.go::IsInterestingAddress()

```go
func (idx *Indexer) IsInterestingAddress(addr string) bool {
  if addr == "" {
    return false
  }
  return slices.Contains(idx.Config.GetInterestingAddressForConfigEnv(), addr)
}
```

`slices.Contains()` is case sensitive, so if `addr` is an ERC-55 checksummed address, the function will not correctly identify the address as interesting as addresses in the list of interesting addresses are not checksummed.

This issue has a low likelihood of being exploited as the JSON-RPC response from the Ethereum node usually does not contain checksummed addresses. However, according to the Ethereum JSON-RPC specification, addresses match the regex pattern `^0x[0-9a-fA-F]{40}$`, indicating that it is possible to have checksummed addresses in the response.

Recommendations

Consider making sure `IsInterestingAddress()` is case insensitive by converting `addr` to lowercase before checking if it is in the list of interesting addresses.

Resolution

The EigenLayer team has implemented the recommended fix above.

This issue has been resolved in commit c7fa20d.

ELSC-15	Panic When Latest State Root Matches Last Indexed Block		
Asset	`blockIndexer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When indexing from the current state to the tip, the system increments `latestBlock` after verifying and potentially removing any corrupted state. However, if the `latestStateRoot` is already up-to-date and corresponds exactly to the previously indexed block, the code increments `latestBlock` by one.

blockIndexer.go::IndexFromCurrentToTip()

```
} else {
    // otherwise, start from the latest block + 1
    latestBlock++
}
```

Therefore, if the current node tip (`blockNumber`) is exactly at this incremented block, the `latestBlock` variable will effectively become `currentTip + 1` . This leads to a scenario where the tip from the node (`blockNumber`) is less than `latestBlock` , causing the check to fail and return an error.

blockIndexer.go::IndexFromCurrentToTip()

```
if blockNumber < uint64(latestBlock) {
    return errors.New("Current tip is less than latest block. Please make sure your node is synced to tip.")
}
```

Recommendations

Consider trying to fetch the latest *safe* block number again after 7 minutes if the latest safe block number is less than the last indexed block.

Keep in mind that it is the latest *safe* block number, as mentioned in ELSC-01.

Resolution

The EigenLayer team has implemented a retry mechanism where `sidecar` will attempt to fetch the latest safe block number from the node every 7 minutes for 3 times.

This issue has been resolved in commits f7a7893 and 963876f.

ELSC-16	Lexicographic Sorting Of `SlotID`		
Asset	`eigenState/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`SlotID` is sorted lexicographically using `strings.Compare()` in `slices.SortFunc()`.

```
slices.SortFunc(inputs, func(i, j *base.MerkleTreeInput) int {
        return strings.Compare(string(i.SlotID), string(j.SlotID))
    })
```

However `SlotID` contains `logIndex` which is of type `uint64`. It is expressed in decimal in the formatted string:

rewardSubmissions.go::NewSlotID()

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

This can create ambiguity when sorting, as lexicographic sort will result in a different order.

Recommendations

Consider converting any numbers in the `SlotID` into fixed-length hex strings when creating the `slotID` in all `NewSlotID()` functions.

Resolution

The EigenLayer team has implemented a fix by replacing the `%d` format specifier with `%016x` in the `NewSlotID()` functions in the state models and `baseEigenState.go::NewSlotIDWithSuffix()` function.

This issue has been resolved in commit 8801a85.

ELSC-17	Missing JSON Tags In `genericRewardPaymentData` Struct		
Asset	`rewardSubmissions.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `genericRewardPaymentData` struct in the `rewardSubmissions` package is currently functioning without explicit JSON tags on most of its fields. While the code works because Go's `json.Decoder` is case-insensitive by default, this relies on implicit behaviour rather than explicit mapping.

rewardSubmissions.go

```go
type genericRewardPaymentData struct {
    Token                    string
    Amount                   json.Number
    StartTimestamp           uint64
    Duration                 uint64
    StrategiesAndMultipliers []struct {
        Strategy   string
        Multiplier json.Number
    } `json:"strategiesAndMultipliers"`
}
```

Recommendations

Add explicit JSON tags to all fields:

rewardSubmissions.go

```go
type genericRewardPaymentData struct {
    Token                    string      `json:"token"`
    Amount                   json.Number `json:"amount"`
    StartTimestamp           uint64      `json:"startTimestamp"`
    Duration                 uint64      `json:"duration"`
    StrategiesAndMultipliers []struct {
        Strategy   string      `json:"strategy"`
        Multiplier json.Number `json:"multiplier"`
    } `json:"strategiesAndMultipliers"`
}
```

Resolution

The EigenLayer team has implemented the recommended fixes above.

This issue has been resolved in commit 3d4a56a.

ELSC-18	Use Of Deprecated Package `xerrors`		
Asset	`/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The codebase uses 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 fixes above.

This issue has been resolved in commit edb46cd.

ELSC-19	Unsafe Integer Comparison In Sort Functions		
Asset	`pipeline.go, submittedDistributionRoots.go, disabledDistributionRoots.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A common but unsafe pattern for implementing sort comparisons is used across multiple files. The pattern involves subtracting two `uint64` values and casting the result to `int`, which can lead to integer overflow on 32-bit architectures.

This anti-pattern appears in several sorting implementations:

```go
// Common unsafe pattern:
slices.SortFunc(items, func(a, b SomeType) int {
    return int(a.SomeUint64Field - b.SomeUint64Field)
})
```

The pattern is used in three locations:

1. Block number comparisons in `pipeline.go` for sorting fetched blocks
2. Root index comparisons in `submittedDistributionRoots.go` for sorting state model leaves
3. Root index comparisons in `disabledDistributionRoots.go` for sorting state model leaves

This issue has a low impact and likelihood as `sidecar` only supports 64-bit systems. In 64-bit systems, `int` will be 64-bit and the casting will lead to the correct resulting value despite the overflow.

Recommendations

Consider replacing current implementation with safe `uint64` comparison logic that returns standard comparison values `(-1, 0, 1)` without performing subtraction.

Alternatively clearly document that `sidecar` does not support 32-bit systems.

Resolution

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

> *"We intend to only support 64-bit systems for `sidecar`."*

ELSC-20	Merkle Tree Shrinking Attack In State Root Generation		
Asset	`stateManager.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The state root merkle tree implementation does not employ any domain separation between the different types of leaves. This opens up the possibility of shrinking attacks in cases where the leaf is 64 bytes long.

The current implementation in `GenerateStateRoot()` creates a merkle tree with variable depth. This structure allows an attacker to potentially use an intermediate node as a leaf node in a proof, effectively "shrinking" the tree and creating valid, but fraudulent proofs as the proof length cannot be checked against a fixed constant.

This issue is classified as low impact and low likelihood as the state root is currently not used onchain and hence this issue is currently not exploitable since the leaves are hashed and either less than or greater than 64 bytes.

Recommendations

Since the state tree has variable depth, shrinking attacks cannot be mitigated by checking the proof length against a fixed constant.

Instead, consider implementing domain separation by adding a unique prefix to each leaf type to ensure they cannot be confused with intermediate nodes. An example of prefixes that can be used is shown below:

- `blockNumber` : `0x00`
- `blockHash` : `0x01`
- state model roots: `0x02`
- state model block number: `0x03`
- state model leaf: `0x04`

Resolution

The EigenLayer team has implemented the recommended fix by adding a unique prefix to each leaf type.

This issue has been resolved in commits e74479b and 48a71b0.

ELSC-21	Restaked Strategies Indexing Interval Is Too Long
Asset	`pipeline.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The current indexing of restaked strategies occurs every 3600 blocks. While this should theoretically happen twice daily (every 12 hours with a 12 second block time), missed/skipped slots could cause the actual indexing interval to be longer than intended, potentially resulting in the indexing occurring once in a day.

pipeline.go::RunForFetchedBlock()

```
if block.Block.Number.Value()%3600 == 0 {
    p.Logger.Sugar().Infow("Indexing OperatorRestakedStrategies", zap.Uint64("blockNumber", block.Block.Number.Value()))
    if err := p.Indexer.ProcessRestakedStrategiesForBlock(ctx, block.Block.Number.Value()); err != nil {
        p.Logger.Sugar().Errorw("Failed to process restaked strategies", zap.Uint64("blockNumber", block.Block.Number.Value()),
            ↪  zap.Error(err))
        return err
    }
}
```

Recommendations

To guarantee rewards gets indexed at least twice a day, consider decreasing the interval, such as for every 3500 blocks.

Resolution

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

> *"This [issue] will be officially fixed in the slashing release as we will no longer need to make contract calls to get operator restaked strategies."*

ELSC-22	Lack Of Context Cancellation
Asset	`blockIndexer.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The goroutine in `IndexFromCurrentToTip()` has no context awareness, meaning it will not respect context cancellation. While it checks the `shouldShutdown` flag, it does not properly handle the passed context that could signal cancellation from the caller. This could lead to a goroutine leak since the monitoring loop will continue running even if the parent context is cancelled.

```
// Every 10 seconds, check to see if the current tip has changed while the backfill/sync
// process is still running. If it has changed, update the value which will extend the loop
// to include the newly discovered blocks.
go func() {
```

Recommendations

The goroutine should be modified to properly handle context cancellation.

Resolution

The EigenLayer team has implemented a deferred `indexComplete` flag to signal the goroutine to shutdown.

This issue has been resolved in commit 8fd1999.

ELSC-23	Missing Error Propagation In `GetLastIndexedBlock()`
Asset	`blockIndexer.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `GetLastIndexedBlock()` function fails to propagate errors when retrieving the latest block, which leads to an ungraceful exit through a `nil` pointer dereference panic rather than proper error handling.

The function logs the error but does not return it, proceeding to access `block.Number` which will cause a panic when `block` is `nil`:

```go
func (s *Sidecar) GetLastIndexedBlock() (int64, error) {
    block, err := s.Storage.GetLatestBlock()
    if err != nil {
        s.Logger.Sugar().Errorw("Failed to get last indexed block", zap.Error(err))
    }
    return int64(block.Number), nil
}
```

This issue has an informational severity rating as this is primarily a code quality issue - the program would also exit if the error was correctly propagated. However, the current implementation does not exit gracefully and masks the original error with a panic stack trace.

Recommendations

Modify the function to properly propagate the error for cleaner error handling.

Resolution

The EigenLayer team has implemented the recommended fix above.

This issue has been resolved in commit ad0fcce.

ELSC-24	Miscellaneous General Comments
Asset	/*
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. **State Accumulator Existence Check**

   Related Asset(s): eigenState/*

   The `SetupStateForBlock()` function in state models does not verify if a state accumulator already exists for a given block number before creating a new one.

   submittedDistributionRoots.go::SetupStateForBlock()

   ```
   func (sdr *SubmittedDistributionRootsModel) SetupStateForBlock(blockNumber uint64) error {
       sdr.stateAccumulator[blockNumber] = make(map[types.SlotID]*types.SubmittedDistributionRoot)
       return nil
   }
   ```

   Consider returning an error if the `stateAccumulator` already exists for the block for all state models.

2. **Use Of Magic Numbers**

   Related Asset(s): pipeline.go, eigenState/*

   There are multiple instances of magic numbers in the codebase.

   (a) The magic number `3600` is used to represent: 12 hours (`3600 * 12 = 43200 = 12 hrs`)

   pipeline.go::RunForFetchedBlock()

   ```
   if block.Block.Number.Value()%3600 == 0
   ```

   Use a named constant for this value and explain that it is used to represent 12 hours.

   (b) All state model indexes are magic numbers in their respective files. They should be named constants and in one file so it's easy to read and access as shown below:

   baseEigenState.go

   ```
   package eigenState

   type ModelIndex int

   const (
       AvsOperatorsModel ModelIndex = iota          // 0
       OperatorSharesModel                          // 1
       StakerDelegationsModel                       // 2
       StakerSharesModel                            // 3
       SubmittedDistributionRootsModel              // 4
       RewardSubmissionsModel                       // 5
       DisabledDistributionRootsModel               // 6
   )
   ```

3. **Unnecessary Sleep**

   Related Asset(s): contractCaller/sequentialContractCaller/contractCaller.go

The retry loop in `getOperatorRestakedStrategiesRetryable()` sleeps after every failed request, meaning that it sleeps unnecessarily on the last request.

contractCaller/sequentialContractCaller/contractCaller.go::getOperatorRestakedStrategiesRetryable()

```go
for i, backoff := range retries {
        results, err := getOperatorRestakedStrategies(ctx, avs, operator, blockNumber, client, l)
        if err != nil {
            l.Sugar().Errorw("GetOperatorRestakedStrategiesRetryable - failed to get results",
                zap.Int("attempt", i+1),
                zap.String("avs", avs),
                zap.String("operator", operator),
                zap.Uint64("blockNumber", blockNumber),
                zap.Error(err),
            )
            // @audit It also sleeps here on the last retry attempt
            time.Sleep(time.Second * time.Duration(backoff))
        } else {
```

Consider sleeping before calling `getOperatorRestakedStrategies()`, or use the same code from `fetcher.go::FetchBlocksWithRetries()` where an early return is made when `err == nil`.

4. **Off By One Calculation**

   Related Asset(s): blockIndexer.go

   The calculation of `blocksProcessed` is off by one.

```go
blocksProcessed += (endBlock - latestBlock)
```

   It should be `blocksProcessed += (endBlock - latestBlock) + 1` as it is inclusive of the first block

5. **Database Atomicity**

   Related Asset(s): pipeline.go

   When inserting database entries in `RunForFetchedBlock`, there is a lack of database transaction wrapping to ensure atomicity.

   If one of the logs fail midway through the loop, there is no way to roll back the previous inserts. Successfully processed transactions/logs remain in the database.

   Consider implementing database transactions to wrap the entire block of operations in a single atomic unit, ensuring that either all operations succeed or none of them are committed.

6. **Unused Variable**

   Related Asset(s): pipeline.go

   In the `RunForFetchedBlock` function, `isBackfill` is an unused variable.

pipeline.go::RunForFetchedBlock()

```go
func (p *Pipeline) RunForFetchedBlock(ctx context.Context, block *fetcher.FetchedBlock, isBackfill bool) error {
```

   Consider removing the unused variable from the function signature.

7. **Defensive Programming**

   Related Asset(s): baseEigenState.go

   Using unvalidated table names in SQL queries via string formatting creates a potential SQL injection risk.

baseEigenState.go

```go
// tokenizing the table name apparently doesnt work, so we need to use Sprintf to include it.
query := fmt.Sprintf(`
    delete from %s
    where block_number >= @startBlockNumber
`, tableName)
```

   Consider validating `tableName` against a whitelist before inserting it into the query.

8. **Inefficient Operator Shares Calculation**

   Related Asset(s): operatorShares.go

   The calculation of operator shares is inefficient as it multiplies the shares by `-1` if it has decreased.

   operatorShares.go
   ```go
   if log.EventName == "OperatorSharesDecreased" {
       shares = shares.Mul(decimal.NewFromInt(-1))
   }
   ```

   Consider using `shares.Neg()` instead to reduce the overhead of the multiplication.

9. **Redundant Named Parameter In Query**

   Related Asset(s): pkg/eigenState/base/baseEigenState.go

   It is not necessary to pass `tableName` to the query as a named parameter because the code is using `Sprintf` to add the `tableName` to the query.

   baseEigenState.go
   ```go
   // tokenizing the table name apparently doesnt work, so we need to use Sprintf to include it.
   query := fmt.Sprintf(`
       delete from %s
       where block_number >= @startBlockNumber
   `, tableName)
   if endBlockNumber > 0 {
       query += " and block_number <= @endBlockNumber"
   }
   res := db.Exec(query,
       sql.Named("tableName", tableName),
   ```

   Remove the redundant line.

10. **Delayed Error Handling**

    Related Asset(s): pkg/pipeline/pipeline.go

    The error handling in `RunForFetchedBlock()` for writing state roots is delayed until the end of the function.

    pipeline.go::RunForFetchedBlock()
    ```go
    sr, err := p.stateManager.WriteStateRoot(blockNumber, block.Block.Hash.Value(), stateRoot)
        if err != nil {
            p.Logger.Sugar().Errorw("Failed to write state root", zap.Uint64("blockNumber", blockNumber), zap.Error(err))
        } else {
            p.Logger.Sugar().Debugw("Wrote state root", zap.Uint64("blockNumber", blockNumber), zap.Any("stateRoot", sr))
        }
    // ...

    return err
    ```

    Handle the error immediately after writing the state root fails.

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 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.

Impact				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

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