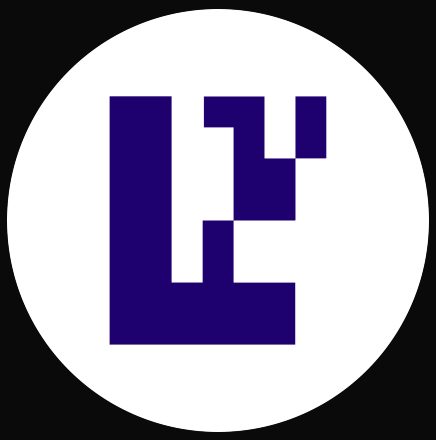
certora

Security Assessment Report



EigenLayer Hourglass pt2

August 2025

Prepared for EigenLayer team

certora

Table of Contents

certora

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
EigenLayer Hourglass pt2	https://github.com/Layr-Labs/hourglass-monorepo/	072f516	Ethereum

The following files are included in the audit scope:

ponos/pkg/chainPoller/EVMChainPoller/evmChainPoller.go
ponos/pkg/aggregator/avsExecutionManager/avsExecutionManager.go
ponos/pkg/operatorManager/operatorManager.go
ponos/pkg/peering/peering.go
ponos/pkg/aggregator/aggregator.go
ponos/pkg/signing/aggregation/ecdsa.go
ponos/pkg/signing/aggregation/bn254.go
ponos/pkg/signer/web3Signer/web3Signer.go
ponos/pkg/contractCaller/contractCaller.go
ponos/pkg/chainPoller/chainPoller.go
ponos/pkg/transactionLogParser/transactionLogParser.go
ponos/pkg/types/task.go
ponos/pkg/taskSession/taskSession.go

Project Overview

This document describes the findings of the manual review of **EigenLayer Hourglass pt2**. The work was undertaken from **July 31, 2025** to **August 11, 2025.**

The team manually audited the respective files using code analysis and inspection tools to search for sensitive patterns and assess code structure, with particular attention to issues that can lead to denial of service on the Hourglass components, and/or incorrect behavior. During the audit, the Certora team discovered issues in the code, as listed on the following page.

Security Considerations

The Hourglass repository contains off-chain components used in the EigenLayer protocol. It contains the Aggregator, Executor, and Performer, which were developed in Golang.

These off-chain components are designed to operate while communicating with each other, and with a limited, pre-configured set of other systems, and thus expose little external attack surface. Apart from the limited external attack surface, Operators control some of the components (the Executors, specifically), and being considered a valid part of the threat model, a malicious Operator / Executor poses a more significant attack surface.

The Hourglass is a relatively large and complex codebase, implementing the Aggregator, Executor and Performer, communication between them, and communication with Onchain systems.

Considering all this, our audit focused on the highest risk areas as we see them. In particular, we considered external attackers (i.e internet exposed endpoints), malicious Operators, friction points between different components, and scenarios that could lead to significant risks, such as cheating (gaining unearned rewards, possibly at the expense of honest peers) and denial of service on important components, especially the Aggregator.

Audit Goals

1. Enumerate the attack surfaces related to the Hourglass repo, the Executor, Aggregator and Performer in particular.
2. Find specific attack vectors in which attackers could realistically lead to incorrect behavior and/or denial of service of said components.
3. Suggest limited and accurate fixes for such attack vectors.

Key Focus Areas

1. Task Processing Vulnerabilities
2. Aggregation System Security
3. Concurrency and Race Conditions
4. Data Parsing and Validation

Coverage and Conclusions

1. The audited components have limited direct external attack surface, as they mostly communicate with pre-defined, trusted components. However, these endpoints expose significant security risks, and the EigenLayer team mentioned that they intend to close most of these endpoints, which is possibly the most important takeaway.
2. The most significant attack vector that was considered is a malicious Operator, which controls an Executor and thus has substantial attack surfaces.
3. The audit focused on:
    a. A malicious Operator targeting other Operators or the Aggregator
    b. External attackers (open RPCs)
    c. Attacks that DOS the aggregator
    d. Attacks that allow cheating (gaining rewards unfairly)
    e. Integrations between the Executors, Aggregator, and smart contracts.
4. Some areas that were not fully covered:
    a. A malicious AVS
    b. Code directly related to performers
    c. The infrastructure itself
    d. Some of the interactions between components
5. In particular, the audit's scope did **not** include:
    a. MITM attack scenarios
    b. Supply chain attacks (for example, a compromised or vulnerable Go package used by the Transporter)
    c. Security of third-party code
    d. Attackers with access to the environment in which the audited components are executed

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	-	-
High	4	4	4
Medium	5	5	4
Low	4	4	3
Informational	3	3	0
Total	16	16	11

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
H-01	Bips vs Percentage Confusion in NewBN254TaskResultAggregator() Leads to 1% Threshold for Task Acceptance	High	Fix confirmed
H-02	Bips vs Percentage Confusion in NewECDSATaskResultAggregator() Leads to 1% Threshold for Task Acceptance	High	Fix confirmed
H-03	Aggregator TOCTOU Issues Regarding Stake Weights and Operator Set	High	Fix confirmed
H-04	Very Large TaskResult Could DOS the Aggregator	High	Fix confirmed
M-01	Executor Contains Open RPC Endpoints Which Allow Code Execution	Medium	Fix confirmed
M-02	Malicious Operator Can Unfairly Gain Rewards By Providing an Honest Operator's Executor and Public Key	Medium	Acknowledged
M-03	Task Result uses lastReceivedResponse	Medium	Fix confirmed
M-04	Non-Deterministic Signature Ordering of ESCDA Certificates	Medium	Fix confirmed

M-05	Task ID Confusion	Medium	Fix confirmed
L-01	Resource Exhaustion via Task Deadline Abuse	Low	Acknowledged
L-02	Task Deadline Integer Overflow	Low	Fix confirmed
L-03	Signing Threshold Never Met Due to Task Submission Failures	Low	Fix confirmed.
L-04	TaskSession.Broadcast(): Task Processing Race Condition	Low	Fix confirmed
I-01	Processing Tasks for AVS without Executor should Fail Fast	Informational	Acknowledged
I-02	Remote Web3 Signer: TLS not enforced	Informational	Acknowledged
I-03	Remote Web3 Signer: Authentication not enforced	Informational	Acknowledged

High Severity Issues

H-01: Bips vs Percentage Confusion in NewBN254TaskResultAggregator() Leads to 1% Threshold for Task Acceptance

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: ponos/pkg/taskSession/taskSession.go ponos/pkg/signing/aggregation/bn254.go	Status: Fix confirmed	

Description:

The definition `NewBN254TaskResultAggregator()` contains a threshold parameter. In the past this was `thresholdPercentage` argument, but currently (in master) it was changed to `thresholdBips`. The call from `taskSession.go::59` passes `100` as an argument (likely this was forgotten when moving from percentage to Bips).

Currently - 1% of signers is enough for task verification.

Exploit Scenario:

It is enough for 1% of the Operators to collude and pass the threshold required, so malicious Operators can cheaply accept any task.

Recommendations

1. Use `10000` instead of `100` when calling `NewBN254TaskResultAggregator()`
2. Consider using a define/constant value instead of hard-coded values

Customer's response: Fixed in PR-133.

Fix Review: Fix confirmed

H-02: Bips vs Percentage Confusion in NewECDSATaskResultAggregator() Leads to 1% Threshold for Task Acceptance

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: ponos/pkg/taskSession/taskSession.go ponos/pkg/signing/aggregation/ecdsa.go	Status: Fix confirmed	

Description:

The definition `NewECDSATaskResultAggregator()` contains a threshold parameter. In the past this was `thresholdPercentage` argument, but currently (in master) it was changed to `thresholdBips`. The call from `taskSession.go::111` passes `100` as an argument (likely this was forgotten when moving from percentage to Bips).

Currently - 1% of signers is enough for task verification.

Exploit Scenario:

It is enough for 1% of the Operators to collude and pass the threshold required, so malicious Operators can cheaply accept any task.

Recommendations

1. Use `10000` instead of `100` when calling `NewECDSATaskResultAggregator()`
2. Consider using a define/constant value instead of hard-coded values

Customer's response: Fixed in PR-133.

Fix Review: Fix confirmed

H-03: Aggregator TOCTOU Issues Regarding Stake Weights and Operator Set

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description:

At `operatorManager/operatorManager.go::GetExecutorPeersAndWeightsForBlock()`, the Aggregator fetches the stakes from the blockchain before handling a task. It uses the fetched value until the task handling is complete. However, the stakes and/or `OperatorSet` could have changed after the call to `GetExecutorPeersAndWeightsForBlock()` and before the task handling was completed.

Exploit Scenarios:

Scenario 1 - Stake Weights:

1. Operator has 25% stake weight when the task starts at `operatorManager.go:187 (GetOperatorTableDataForOperatorSet)`
2. Aggregator fetches this 25% weight and stores it in `operatorPeersWeight` at `avsExecutionManager.go:373-378`
3. During task processing (which can take time), operator unstakes to reduce their actual stake to 0%
4. Task completion uses the stale 25% weight for reward calculations at `avsExecutionManager.go:488 (SubmitBN254TaskResultRetryable)` and `avsExecutionManager.go:580 (SubmitECDSATaskResultRetryable)`

Scenario 2 - Operator Set:

1. Operator registers in the operator set just before task creation
2. Aggregator fetches operator set at `operatorManager.go:211-214 (ListExecutorOperators)`
3. During task processing, operator deregisters from the set
4. Operator still receives rewards as part of the "valid" operator set despite no longer being registered

Recommendations: Avoid this scenario by enforcing task SLA and withdraw delay to be within safe limits.

Customer's response: FIxed in PR-1604

Fix Review: Fix confirmed

H-04: Very Large TaskResult Could DOS the Aggregator

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description:

A malicious Operator that handles a task can return a very large `TaskResult`, which will cause an out-of-memory error in the Aggregator and potentially crash it.

Exploit Scenario:

1. Aggregator distributes a legitimate task to the malicious operator via `SubmitTask` (at `taskSession.go:225`)
2. The operator crafts a `TaskResult` with an enormous Output field, for example 2GB.
3. gRPC Transmission: The oversized result is sent back to the aggregator through the gRPC channel

Recommendations: The Aggregator should check the TaskResult's size, and discard it if it passes a size threshold.

Customer's response: Fixed in PR-135

Fix Review: Fix confirmed
Note: Task Result size check was implemented correctly. The aggregator checks result size **after** completion of result reception. Large results can still be submitted and potentially cause aggregator OOM. Consider implementing socket level size check and stop processing once the result size exceeds the threshold.

Medium Severity Issues

M-01 Executor Contains Open RPC Endpoints Which Allow Code Execution

Severity: **Medium**	Impact: **High**	Likelihood: **High**
Files: ponos/pkg/executor/handlers.go	Status: Fix confirmed	

Description:

The executor contains an open gRPC endpoint - `DeployArtifact()` - which allows anyone to execute code (within a performer) with no verification via `ponos/pkg/executor/handlers.go::DeployArtifact()`.

Notice that this is not the only risky endpoint - we didn't analyze each since the client said they are aware of this issue and will be protecting these endpoints with authentication.

Exploit Scenario:

An external attacker finds the IP of the open endpoint and calls `DeployArtifact()` to directly execute arbitrary code within a performer.

Recommendations: Authenticate external users who access any open endpoint.

Customer's response: Fixed in PR-136

Fix Review: Fix confirmed

Note: Consider Adding Rate Limits GetChallengeToken. Calling this endpoint with valid operator addresses generates a token. Attackers can call this repeatedly with public operator addresses. Each call generates new token in memory, which eventually can cause memory exhaustion and DOS the aggregator.

M-02 Malicious Operator Can Unfairly Gain Rewards By Providing an Honest Operator's Executor and Public Key

Severity: **Medium**	Impact: **High**	Likelihood: **Medium**
Files: ponos/pkg/taskSession/taskSession.go	Status: Acknowledged	

Description:

A malicious Operator copies an honest Operator's executor endpoint and public key, and has the Aggregator assign tasks to that honest Operator, but the malicious Operator will be netting the rewards

Recommendations: The registrar should not allow multiple Operators with the same public key.

Customer's response: Addressed by enforcement of unique keys on-chain.

M-03 Task Result uses lastReceivedResponse

Severity: **Medium**	Impact: **High**	Likelihood: **High**
Files: ponos/pkg/signing/aggregation/bn254.go ponos/pkg/signing/aggregation/ecdsa.go	Status: Fix confirmed	

Description:

The aggregator uses only the `lastReceivedResponse` for the final certificate, ignoring all previous task results from other operators. This creates a condition where the last operator to respond determines the final task output, regardless of whether their result is correct or malicious. A single malicious operator can control the final task outcome.

Note: This finding was previously identified by the EigenLayr internal team and is documented in the codebase with a TODO comment acknowledging the issue. The internal team has already recognized the vulnerability where the last operator to respond determines the final task output, potentially allowing malicious operators to control the task outcome.

ponos/pkg/signing/aggregation/ecdsa.go:230
ponos/pkg/signing/aggregation/bn254.go:224

```Go
tra.aggregatedOperators.lastReceivedResponse = rr  // ← LAST WRITER WINS
```

Exploit Scenario:

3 operators submit different results

- Operator A: result = "correct_answer"
- Operator B: result = "correct_answer"
- Operator C: result = "malicious_answer" ← Last to respond

Final certificate uses only Operator C's result:
TaskResponse: "malicious_answer" ← WRONG RESULT

Recommendations: Use the results that have been received most times. Check the hash for each result, count the hashes, and use the result with the highest hash count.

Customer's response: Fixed in PR-137.

Fix Review: Fix confirmed

M-04 Non-Deterministic Signature Ordering of ESCDA Certificates

Severity: **Medium**	Impact: **High**	Likelihood: **High**
Files: ponos/pkg/signing/aggregation/ecdsa.go	Status: Fix confirmed	

Description:

The aggregator violates the smart contract's requirement for ECDSA signatures to be ordered by signer address in ascending order. The Go code uses non-deterministic map iteration, causing signature order to vary between submissions, leading to contract verification failures.

hourglass-monorepo/ponos/pkg/signing/aggregation/ecdsa.go:36

```go
    SignersSignatures map[common.Address][]byte
```

hourglass-monorepo/ponos/pkg/signing/aggregation/ecdsa.go:44-56

```go
func (cert *AggregatedECDSACertificate) GetFinalSignature() ([]byte, error)
{
    var finalSignature []byte
    for _, sig := range cert.SignersSignatures {  // ← NON-DETERMINISTIC MAP
ITERATION
        finalSignature = append(finalSignature, sig...)
    }
    return finalSignature, nil
}
```

eigenlayer-contracts/src/contracts/multichain/ECDSACertificateVerifier.sol:146

```
Go
require(i == 0 || recovered > signers[i - 1], SignersNotOrdered());
```

Recommendations:

Convert the non-deterministic map iteration into a deterministic slice iteration.

hourglass-monorepo/ponos/pkg/signing/aggregation/ecdsa.go:44-56

```
Go
func (cert *AggregatedECDSACertificate) GetFinalSignature() ([]byte, error)
{
    // Extract and sort addresses
    addresses := make([]common.Address, 0, len(cert.SignersSignatures))
    for addr := range cert.SignersSignatures {
        addresses = append(addresses, addr)
    }
    sort.Slice(addresses, func(i, j int) bool {  // SORT INTO SLICE
        return addresses[i].Hex() < addresses[j].Hex()
    })

    // Concatenate in sorted order
    var finalSignature []byte
    for _, addr := range addresses {
        finalSignature = append(finalSignature,
cert.SignersSignatures[addr]...)
    }
    return finalSignature, nil
}
```

Customer's response: Fixed in PR-138

Fix Review: Fix confirmed

M-05 Task ID Confusion

Severity: **Medium**	Impact: **High**	Likelihood: **High**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description:

The aggregator lacks validation to ensure that the task ID in the submitted task result matches the expected task ID for the current task session. This allows malicious operators to submit results with different task IDs, potentially causing signature verification failure or, task result corruption and gas waste due to failed transactions.

ponos/pkg/taskSession/taskSession.go:256

```go
if err := ts.taskAggregator.ProcessNewSignature(ts.context,
taskResult.TaskId, taskResult); err != nil {
```

Missing Validation in bn254.go:
ponos/pkg/signing/aggregation/bn254.go:145

```go
func (tra *BN254TaskResultAggregator) ProcessNewSignature(
	ctx context.Context,
	taskId string,
	taskResponse *types.TaskResult,
) error {
	// ... validation of operator address and signature ...
```

```
    // BUT NO VALIDATION that taskId == taskResponse.TaskId

    rr := &ReceivedBN254ResponseWithDigest{
        TaskId:     taskId,  // Uses the passed taskId parameter
        TaskResult: taskResponse,
        Signature:  sig,
        Digest:     taskResponse.OutputDigest,
    }
```

Missing Validation in ecdsa.go:
ponos/pkg/signing/aggregation/ecdsa.go:152

```go
func (tra *ECDSATaskResultAggregator) ProcessNewSignature(
    ctx context.Context,
    taskId string,
    taskResponse *types.TaskResult,
) error {
    // ... validation of operator address and signature ...
    // BUT NO VALIDATION that taskId == taskResponse.TaskId

    rr := &ReceivedECDSAResponseWithDigest{
        TaskId:     taskId,  // Uses the passed taskId parameter
        TaskResult: taskResponse,
        Signature:  sig,
        Digest:     digest,
    }
```

Recommendations: Ensure operators can only contribute results for the specific task they were assigned, maintaining proper task isolation.

hourglass-monorepo/ponos/pkg/taskSession/taskSession.go:255

```go
// Change from:
if err := ts.taskAggregator.ProcessNewSignature(ts.context,
taskResult.TaskId, taskResult); err != nil {

// To:
if err := ts.taskAggregator.ProcessNewSignature(ts.context, ts.Task.TaskId,
taskResult); err != nil {
```

In aggregation logic (bn254.go/ecdsa.go):

```go
// Add validation at the start of ProcessNewSignature:
if taskId != taskResponse.TaskId {
    return fmt.Errorf("task ID mismatch: expected %s, got %s", taskId,
taskResponse.TaskId)
}
```

Customer's response: Fixed in PR-139 & PR-142.

Fix Review: Fix confirmed

Low Severity Issues

L-01: Resource Exhaustion via Task Deadline Abuse		
Severity: **Low**	Impact: **low**	Likelihood: **Low**
Files: ponos/pkg/taskSession/taskSession.go	Status: Acknowledged	

Description:

Non-responsive operators can tie up the AVS aggregator's resources for extended periods, as it must wait for the full task deadline before releasing resources. While this is a known design decision, it presents an asymmetric cost attack vector.

Exploit Scenario:

1. A malicious Operator registers to multiple Operator Sets with the goal of delaying as many tasks as possible on a target Aggregator.
2. The malicious executor accepts Tasks, but doesn't provide any TaskResult at all.
3. Each affected TaskSession will remain active until either a consensus is met, or until the Task Deadline has passed. This can lead to delay of tasks handling, and resource exhaustion on the Aggregator.

Recommendations: Consider adding logic to keep tabs on mis-behaving Executors. If an executor is consistently and significantly slower than its peers, or alternatively - consistently doesn't provide a TaskResult - take action.

Customer's response: Acknowledged.

Fix Review: It is the Operator's responsibility to evaluate the risk of running an AVS with very long task deadlines.

L-02: Task Deadline Integer Overflow

Severity: **Low**	Impact: **low**	Likelihood: **Low**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description:

`parsedTaskDeadline.Int64()` can overflow, resulting in a `taskDeadlineTime` value in the past. AVS can set a very large `taskSLA`, resulting in a negative `int64` value. Tasks with deadlines set in the past are cancelled immediately without processing.

ponos/pkg/types/task.go:100

```go
var od *outputDataType
if err := json.Unmarshal(outputBytes, &od); err != nil {
    return nil, fmt.Errorf("failed to unmarshal output data: %w", err)
}
parsedTaskDeadline := new(big.Int).SetUint64(od.TaskDeadline)
taskDeadlineTime := time.Now().Add(time.Duration(parsedTaskDeadline.Int64())
* time.Second)
```

Solidity checks only non-zero values for taskSLA, no max value:
contracts/src/core/TaskMailbox.sol:104

```
require(
```

```
            taskConfig.curveType != IKeyRegistrarTypes.CurveType.NONE &&
address(taskConfig.taskHook) != address(0)
                && taskConfig.taskSLA > 0 &&
taskConfig.consensus.consensusType != ConsensusType.NONE,
            ExecutorOperatorSetTaskConfigNotSet()
        );
```

contracts/src/interfaces/core/ITaskMailbox.sol:49

```
None
if parsedTaskDeadline.Int64() < 0 {
        return nil, fmt.Errorf("value %s exceeds int64 bounds", b.String())
}
```

Customer's response: Fixed in PR-141.

Fix Review: Fix confirmed

L-03: Signing Threshold Never Met Due to Task Submission Failures

Severity: **Low**	Impact: **low**	Likelihood: **Low**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description:

The aggregator's signing threshold is never reached when task submission failures occur silently, causing tasks to fail unnecessarily. The threshold calculation assumes all operators will respond, but silent failures from network issues, timeouts, or operator unavailability prevent threshold achievement, leading to task failures even when sufficient operators are available.

hourglass-monorepo/ponos/pkg/taskSession/taskSession.go:199-242

```go
go func(peer *peering.OperatorPeerInfo) {
    socket, err := peer.GetSocketForOperatorSet(ts.Task.OperatorSetId)
    if err != nil {
        ts.logger.Sugar().Errorw("Failed to get socket for operator set",
...)
        return  // ← SILENT FAILURE, NO RETRY
    }

    c, err := executorClient.NewExecutorClient(socket, true)
    if err != nil {
        ts.logger.Sugar().Errorw("Failed to create executor client", ...)
        return  // ← SILENT FAILURE, NO RETRY
    }
```

```
    res, err := c.SubmitTask(ts.context, taskSubmission)
    if err != nil {
        ts.logger.Sugar().Errorw("Failed to submit task to executor", ...)
        return  // ← SILENT FAILURE, NO RETRY
    }

    // Only successful submissions reach here
    resultsChan <- types.TaskResultFromTaskResultProto(res)
}(peer)
```

Threshold Calculation Ignores Failures:
hourglass-monorepo/ponos/pkg/signing/aggregation/ecdsa.go:240-243

```Go
func (tra *ECDSATaskResultAggregator) SigningThresholdMet() bool {
    required := int((float64(tra.ThresholdBips) / 10_000.0) *
float64(len(tra.Operators)))
    // Assumes ALL operators will respond, ignores submission failures
    return tra.aggregatedOperators.totalSigners >= required
}
```

Exploit Scenario

1. Selective Network Issues: Operators can selectively fail network connections
2. Timeout Manipulation: Cause timeouts to prevent task submission
3. Service Disruption: Force task failures even with sufficient willing operators

Recommendations:

- Track failing task submissions
- Adjust Threshold for failed submissions
- Add submission retry logic

Customer's response: Fixed in PR-143.

Fix Review: Fix confirmed

L-04: TaskSession.Broadcast(): Task Processing Race Condition

Severity: **Low**	Impact: **low**	Likelihood: **Low**
Files: ponos/pkg/taskSession/taskSession.go	Status: Fix confirmed	

Description

If the task signing threshold is met while task submission is still ongoing, the result channel may be closed while processing pending tasks. Attempting to insert a `taskResult` into the result channel after the `resultChan` was closed will panic the aggregator.

Malicious operators can time their Task Submissions responses to hit the critical windows and crash the aggregator.

ponos/pkg/taskSession/taskSession.go:200

```go
// PRODUCER
go func(peer *peering.OperatorPeerInfo) {
    // ... network call ...
    res, err := c.SubmitTask(ts.context, taskSubmission)
    // ... processing ...
    resultsChan <- types.TaskResultFromTaskResultProto(res)  // PANIC HERE
}(peer)
```

ponos/pkg/taskSession/taskSession.go:276

```Go
// CONSUMER
for taskResult := range resultsChan {
    // ... process result ...
    ts.thresholdMet.Store(true)

    // threshold met, close the results channel to stop further processing
    close(resultsChan)
}
```

Recommendations

Use a separate channel to Signal task Producer threads upon Signing Threshold Hit.

Consumer:

```Go
      if ts.taskAggregator.SigningThresholdMet() {
           ts.thresholdMet.Store(true)
           close(doneChan)  // SIGNAL PRODUCERS TO STOP
           close(resultsChan)

           cert, err := ts.taskAggregator.GenerateFinalCertificate()
           return cert, nil
       }
```

Producer Thread:

```go
          // CHECK IF DONE BEFORE SENDING
           select {
                case resultsChan <-
types.TaskResultFromTaskResultProto(res):
                case <-doneChan:
                  return  // STOP SENDING IF DONE
           }
```

Customer's response: Fixed in PR-145.

Fix Review: Fix confirmed

Informational Issues

I-01. Processing Tasks for AVS without Executor should Fail Fast

Description:

Check for a non-empty operator list that is missing. The missing check does not introduce a bug, but this is detected down, later in the code flow, after further processing is done.
Future code changes would be more prone to errors, hence it would be better to terminate task handling right upon receiving an empty operator list.

ponos/pkg/operatorManager/operatorManager.go:211

```go
operators, err := om.peeringDataFetcher.ListExecutorOperators(ctx,
om.config.AvsAddress)   // Non-empty list check is missing
```

The code flow continues normally until eventually errors on `NewBN254TaskResultAggregator` or `NewECDSATaskResultAggregator`:

ponos/pkg/signing/aggregation/ecdsa.go:128
ponos/pkg/signing/aggregation/bn254.go:74

```go
      if len(operators) == 0 {
            return nil, ErrNoOperatorAddresses
      }
```

Recommendations: Check that the returned operator list has values. Fail fast in case the list is empty.

Customer's response: Acknowledged.

I-02. Remote Web3 Signer: TLS not enforced

Description:

`remoteSignerConfig` can be passed without the `https://` prefix. As a result, `NewConfigWithTLS` will fall back to `HTTP`. `https://` can be omitted from configuration deliberately (signer is knowingly configured without https or by assuming https is the default, omitting it from the url completely.

```
Go
# Available but not used in production
operatorPrivateKey:
  remoteSigner: true
  remoteSignerConfig:
    url: "web3signer.example.com"  # ← No http:// or https://
```

ponos/pkg/clients/web3signer/client.go:120

```
Go
// Only configure TLS if we have HTTPS and at least one TLS field
 if strings.HasPrefix(baseURL, "https://") && (caCert != "" || clientCert !=
"" || clientKey != "") {
        tlsConfig := &TLSConfig{
            CACert:     caCert,
            ClientCert: clientCert,
            ClientKey:  clientKey,
        }
        config.TLS = tlsConfig
    }
```

Recommendations: Enforce TLS for remote signers.

Customer's response: Acknowledged.

I-03. Remote Web3 Signer: Authentication not enforced

Description:

It is possible to create a remote web3 signer without authentication in place. Although client certificates can be used for authentication, it is not mandatory and not enforced.

Recommendations: Enforce authentication using TLS Client certificate or another method (API Key, Token, etc).

Customer's response: Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.