MANTLE

Mantle EigenDA Integration

Security Assessment Report

Version: 2.1

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Mantle EigenDA integration. The review focused solely on the security aspects of the implementation, 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 component 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 Mantle 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 Mantle components in scope.

Overview

The Mantle EigenDA integration reroutes of blob data operations from direct calls to EigenDA Disperser to a new intermediary, EigenDA Proxy, which handles blob commitment validation and implements robust fallback mechanisms to prevent L2 safe block derivation from stalling.

By leveraging fallback processes such as S3 backup for writes and fallback reads when EigenDA Disperser fails, this upgrade not only mitigates the risk of data fetch issues but also simplifies the adoption of new EigenDA features.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the mantlenetworkio/mantle-v2 and mantlenetworkio/op-geth repositories.

The scope of this time-boxed review was strictly limited to the following pull requests:

- PR-163: `https://github.com/mantlenetworkio/mantle-v2/pull/163`
- PR-184: `https://github.com/mantlenetworkio/mantle-v2/pull/184`
- PR-185: `https://github.com/mantlenetworkio/mantle-v2/pull/185`
- PR-84: `https://github.com/mantlenetworkio/op-geth/pull/84`
- PR-95 (MetaTx disable): `https://github.com/mantlenetworkio/op-geth/pull/95`
- PR-93 (RIP-7212 implementation): `https://github.com/mantlenetworkio/op-geth/pull/93`

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

Approach

The manual review focused on identifying issues associated with the business logic implementation of the libraries and modules. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime.

Additionally, the manual review process focused on identifying vulnerabilities related to known Golang anti-patterns and attack vectors, such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks, and various panic scenarios including nil pointer dereferences, index out of bounds, and explicit panic calls.

To support the Golang components of the review, the testing team also utilised the following automated testing tools:

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

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 23 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 3 issues.
- Low: 13 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Mantle 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
MEDA-01	Lack Of Cryptographic Verification Of The Offchain Data	High	Closed
MEDA-02	Lack Of `BlobInfo` Validation	Medium	Closed
MEDA-03	Denial-of-Service (DoS) Via Oversized `txData` Frame In `txAggregator`	Medium	Resolved
MEDA-04	Unlimited Retries In `loopRollupDa()` Due To `isRetry()` Always Returning `true`	Medium	Closed
MEDA-05	`nil` Pointer Dereference In `EigenDADataStore` When `daCfg` Is `nil`	Low	Open
MEDA-06	Potential Data Inconsistency Due To Partial Decoding Of Blob Data	Low	Closed
MEDA-07	Unbounded Blob Data Merging In Calldata Source May Lead To Resource Exhaustion	Low	Closed
MEDA-08	Lack Of Retry Backoff And Limits When Indexer Is Syncing May Cause Denial-Of-Service (DoS)	Low	Closed
MEDA-09	Missing Enforcement Of HTTPS/TLS	Low	Closed
MEDA-10	Insufficient SRS File Length Validation Leading To Potential Out-of-Bounds Reads	Low	Closed
MEDA-11	Error Handling Uses Explicit `panic()`	Low	Closed
MEDA-12	Missing Error Handling in JSON Unmarshaling	Low	Resolved
MEDA-13	Potential Denial-of-Service (DoS) Via Excessive Memory Allocation For SRS Field Elements	Low	Closed
MEDA-14	Empty Blob Candidate Generated When First Frame Exceeds Size Limit	Low	Resolved
MEDA-15	HTTP Response Body Not Closed on Error Paths	Low	Open
MEDA-16	Required Casting For `sysCfg.Scalar` Parameter In `EcotoneScalars`	Low	Closed
MEDA-17	Lack Of `nil` Check In `NodeVersion()` May Lead To Panic	Low	Open
MEDA-18	`EigenDaClient.DisperseBlob()` Does Not Handle HTTP 503 Error	Informational	Open
MEDA-19	`SRSOrder` Comment Mismatch	Informational	Closed
MEDA-20	Inconsistent Length Check for `MetaTxPrefix`	Informational	Closed
MEDA-21	Hash Input Considerations	Informational	Closed
MEDA-22	Improper Error Handling in Beacon API Initialization Could Result In Silent Misconfiguration	Informational	Open
MEDA-23	Miscellaneous General Comments	Informational	Closed

MEDA-01	Lack Of Cryptographic Verification Of The Offchain Data		
Asset	`op-node/../calldata_source.go, op-node/../datastore.go, op-batcher/../driver_da.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In `op-node/rollup/derive/calldata_source.go`, when retrieving data from EigenDA via `dataFromEigenDa()`, the code does not appear to verify any cryptographic commitment or Merkle root.

Similarly, in `op-batcher/batcher/driver_da.go` the function `txAggregatorForEigenDa()` does not perform any validation on retrieved data either.

In `op-node/rollup/derive/calldata_source.go::dataFromEigenDa()`, the code calls `RetrieveBlob()` and then trusts the returned bytes. No hash check or commitment verification is performed to ensure the returned data matches an onchain or previously known commitment.

If the offchain source is compromised or returns tampered data, the function simply merges it into the returned data.

As advised by the development team, the verification of blobs is handled within EigenDA Proxy, which is deployed together with op-node. Whilst it partially addresses this finding, it does not fully eliminate it:

- If the node trusts data from the proxy without revalidating cryptographic commitments, then the compromise of the proxy could result in tampered data being passed straight to the node.
- Having a single point (the proxy) that performs certificate or proof validation does reduce the code duplication for verifying KZG or other cryptographic proofs. However, it also creates a single point of failure if that verification or environment is compromised, the node's trust model is compromised as well.
- In common rollup designs, each node (or at least each "verifier" node) does or can do a minimal re-check that the data matches an onchain or known commitment. This ensures any tampering between the data source and the node is detectable.

Note, if the proxy is local and strongly isolated, that risk may be reduced.

Recommendations

If the node is expected to be running in more distributed or less controlled settings, implement additional checks within the node to re-verify data, or re-check commitment hashes, ensuring a compromised proxy cannot feed bad data.

Resolution

The finding has been closed with the following comment from the development team:

"The eigenda-proxy is integrated as part of the Mantle operator node. The operator node has expanded from its original two components (op-node and op-geth) to three components: op-node, op-geth, and eigenda-proxy. Notably, unlike op-node and op-geth, the eigenda-proxy does not require exposing ports to external networks. This design significantly reduces its vulnerability to be compromised compared to op-node and op-geth.

When the proxy is compromised, it means that the node is likely also compromised. Moreover, when a proxy is compromised, it only affects its corresponding node and does not impact other nodes. We believe that the proxy merely places the verification logic in a separate process, and this does not affect the security of the node."

MEDA-02	Lack Of `BlobInfo` Validation		
Asset	`op-service/eigenda/da_proxy.go, op-batcher/batcher/driver_da.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In `op-service/eigenda/da_proxy.go` the function `DecodeCommitment()` does not perform any validation on the `blobInfo` after calling `rlp.DecodeBytes(data, blobInfo)`.

There could be a situation where `blobInfo.BlobVerificationProof` is `nil`, which could lead to `nil` dereference as seen in the following code segment:

```
unc (l *BatchSubmitter) disperseEigenDaData(data [][]byte) ([]byte, error) {
    encodeData, err := rlp.EncodeToBytes(data)
    if err != nil {
        l.log.Error("op-batcher unable to encode txn", "err", err)
        return nil, err
    }

    blobInfo, err := l.eigenDA.DisperseBlob(l.shutdownCtx, encodeData)
    if err != nil {
        l.log.Error("Unable to publish batch frameset to EigenDA", "err", err)
        return nil, err

    }
    commitment, err := eigenda.EncodeCommitment(blobInfo)
    if err != nil {
        return nil, err
    }

    quorumIDs := make([]uint32, len(blobInfo.BlobHeader.BlobQuorumParams))
    for i := range quorumIDs {
        quorumIDs[i] = blobInfo.BlobHeader.BlobQuorumParams[i].QuorumNumber
    }
    calldataFrame := &op_service.CalldataFrame{
        Value: &op_service.CalldataFrame_FrameRef{
            FrameRef: &op_service.FrameRef{
                ReferenceBlockNumber: blobInfo.BlobVerificationProof.BatchMetadata.BatchHeader.ReferenceBlockNumber,    // @audit
  ↪  this could panic with nil pointer dereference
                QuorumIds:           quorumIDs,
                BlobLength:          uint32(len(encodeData)),
                Commitment:          commitment,
            },
        },
    }
...
}
```

Recommendations

Verify that `blobInfo.BlobHeader`, `BlobInfo.BlobVerificationProof` and their members are not `nil`.

Resolution

The finding has been closed with the following comment from the development team:

> *"The commitment is returned by eigenda-proxy. In the currently deployed eigenda-proxy (v1.6.3), the following code ensures the blobInfo.BlobVerificationProof after DecodeCommitment never be nil: https://github.com/Layr-Labs/eigenda-proxy/blob/5530ac4e93a50806bb45a18d15761f2bddb92c7b/verify/certificate.go#L35."*

MEDA-03	Denial-of-Service (DoS) Via Oversized `txData` Frame In `txAggregator`		
Asset	`op-batcher/batcher/driver_da.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `txAggregator()` function in `driver_da.go` is vulnerable to a potential DoS scenario if a singular `txData` frame exceeds the configured `RollupMaxSize`. In such a case, the loop in `txAggregator()` immediately breaks, resulting in an empty transaction payload, while leaving the oversized frame in the pending queue, causing subsequent aggregator calls to repeatedly fail.

On each iteration of the `txAggregator()` function, it encodes the current list of frames and checks the total size. If the size meets or exceeds the limit, the function logs the event and breaks out of the loop without processing further frames:

```
if uint64(len(txnBufBytes)) >= l.RollupMaxSize {
    l.log.Info("op-batcher transactionByte size is more than RollupMaxSize", "rollupMaxSize", l.RollupMaxSize, "txnBufBytes",
      ↪  len(txnBufBytes), "transactionByte", len(transactionByte))
    l.metr.RecordTxOverMaxLimit()
    break
}
```

In the case where the very first `txData` is already oversized, the loop exits immediately. As a result, the aggregated `transactionByte` remains empty, and the function ultimately returns an error *"txsData is empty"*. Critically, the oversized frame is not removed from `daPendingTxData`.

While frames are initially obtained via `s.pendingChannel.NextFrame()`, which is limited by a configured `MaxFrameSize`, there is no enforcement ensuring that `MaxFrameSize` is always less than or equal to `RollupMaxSize`. Misconfiguration could allow an oversized frame to occur, triggering this DoS condition.

Recommendations

Enforce an invariant where the configuration parameter `MaxFrameSize` is always less than or equal to `RollupMaxSize`. This validation should be performed at startup to avoid misconfigurations that could trigger this issue.

Resolution

The finding has been resolved in PR-194.

MEDA-04	Unlimited Retries In `loopRollupDa()` Due To `isRetry()` Always Returning `true`		
Asset	`op-batcher/batcher/driver_da.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `isRetry()` function within the batch submission process always returns `true`, even after exceeding the maximum number of allowed attempts.

This causes the error handling branch in `loopRollupDa()` to become unreachable, leading to an unlimited number of retries.

```
func (l *BatchSubmitter) isRetry(retry *int32) bool {
    *retry = *retry + 1
    l.metr.RecordRollupRetry(*retry)
    if *retry > DaLoopRetryNum {
        l.log.Error("rollup failed by 10 attempts, need to re-store data to mantle da")
        *retry = 0
        l.state.params = nil
        l.state.initStoreDataReceipt = nil
        l.metr.RecordDaRetry(1)
        return true
    }
    time.Sleep(5 * time.Second)
    return true
}
```

In the caller function `loopRollupDa()`, the code checks the return value of `isRetry()`:

```
if l.isRetry(&retry) {
    continue
}
return false, err
```

Since `isRetry()` always returns `true`, the condition never allows the function to exit the loop by returning `false, err`, even when the maximum number of retry attempts (currently set to 10) is reached.

This could result in a DoS condition in `loopRollupDa()` when persistent errors occur.

Recommendations

Modify the `isRetry()` function to return `false` when the retry count exceeds `DaLoopRetryNum`.

This will allow the caller function `loopRollupDa()` to break out of the loop and properly propagate the error.

Resolution

The finding has been closed with the following comment from the development team:

> *"After the successful upgrade, the LoopRollupDa() method will no longer be invoked."*

MEDA-05	`nil` Pointer Dereference In `EigenDADataStore` When `daCfg` Is `nil`		
Asset	`op-node/rollup/da/datastore.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When initializing an `EigenDADataStore` via `NewEigenDADataStore()`, the `daClient` is conditionally instantiated only if `daCfg` is not `nil`.

If `daCfg` is `nil`, `daClient` remains uninitialized, which will later lead to a `nil` pointer dereference panic when any method, such as `RetrieveBlob()` or `RetrieveBlobWithCommitment()`, attempts to call a method on `daClient`:

```
func NewEigenDADataStore(ctx context.Context, log log.Logger, daCfg *eigenda.Config, cfg *MantleDataStoreConfig, m eigenda.Metrics)
    ↪  *EigenDADataStore {
    var daClient eigenda.IEigenDA
    if daCfg != nil {
        daClient = eigenda.NewEigenDAClient(*daCfg, log, m)
    }
    return &EigenDADataStore{
        daClient: daClient,
        Cfg:      cfg,
        Ctx:      ctx,
    }
}
```

Recommendations

Modify `NewEigenDADataStore()` to explicitly check if `daCfg` is `nil`.

If `daCfg` is `nil`, the function should either return an error (changing the function signature to include error propagation) or initialize a safe fallback that prevents subsequent method calls from causing a `nil` pointer dereference.

Resolution

The finding remains open, with the following comment from the development team:

> *"dacfgIs will not be nil when the op-batcher started up with correct configuration. We will fix this issue in the next version."*

MEDA-06	Potential Data Inconsistency Due To Partial Decoding Of Blob Data		
Asset	`op-node/rollup/derive/calldata_source.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The current implementation in `op-node/rollup/derive/calldata_source.go` logs an error and continues when a blob fails to parse, potentially leading to partial data merges.

This behavior risks an incomplete state if the blobs being skipped are critical transactions, which may ultimately cause inconsistencies or unexpected reorgs in the chain.

The intended functionality of this module is to construct a complete and correct set of calldata from multiple sources, including L1 transactions, receipts, EigenDA data, and blobs.

```
blobData, err := blob.ToData()
if err != nil {
    log.Error("ignoring blob due to parse failure", "err", err)
    continue
}
```

When a blob fails to decode, this error-handling silently drops the blob data without halting processing or signaling a failure at a higher level. If the skipped blob contains critical transaction data, the resulting state could be incomplete or inconsistent.

If this behavior is not intentional and partial merges are not acceptable for the integrity of the state, then continuing after a critical blob parsing failure could result in invalid block processing, leading to reorgs or other unintended protocol behaviors.

Recommendations

Modify the implementation such that, if an invalid frame is critical, fail the entire batch or mark the entire block for reprocessing instead of skipping.

Resolution

The finding has been closed with the following comment from the development team:

> *"The decision to handle corrupted blobs by dropping them is because, when the op-node derives the safe block, it must be continuous. Therefore, whether or not the blob is dropped, the growth of the safe block will stop.*
>
> *The reason for dropping it is that once we detect that the safe block has stopped growing, we can restart the op-batcher and resubmit the transactions from the following block of the latest safe block. If we halted processing*

on this blob, there would be no way to resume the derivation of the safe block if a permanently corrupted blob occurs.

Therefore, we believe that continuing is a better option than halting processing or signaling a failure at a higher level."

MEDA-07	Unbounded Blob Data Merging In Calldata Source May Lead To Resource Exhaustion		
Asset	`op-node/rollup/derive/calldata_source.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The current implementation in `op-node/rollup/derive/calldata_source.go` merges blob data from various sources without imposing limits on the number or total size of blobs.

This unbounded merging, followed by RLP decoding of the concatenated blob data, could lead to memory exhaustion or excessive CPU usage when processing abnormally large or numerous blobs.

In the `NewDataSource()` function, when blob transactions are processed, the code preallocates a buffer for merging blob data using:

```
wholeBlobData := make([]byte, 0, len(blobs)*seth.MaxBlobDataSize)
```

This allocation is directly proportional to the number of blobs (`len(blobs)`) and a constant maximum blob data size (`seth.MaxBlobDataSize`). However, there are no safeguards to limit `len(blobs)` or to chunk and bound the resulting `frameData` after merging.

The subsequent call to `rlp.DecodeBytes` then processes this potentially massive byte slice, incurring high CPU overhead and memory usage.

Recommendations

Introduce configurable limits for the maximum number of blobs allowed per L1 block as well as a maximum total size for blob data.

Before merging, validate the size of each blob. If any blob or the aggregated size exceeds a safe threshold, log an appropriate error and either skip processing that block or trigger a fallback mechanism.

Resolution

The finding has been closed with the following comment from the development team:

> *"In op-node/rollup/derive/params.go, the constant MaxRLPBytesPerChannel is defined as 10_000_000. This means that frameData will definitely be smaller than this value. Therefore, it will not lead to high CPU overhead and memory usage."*

MEDA-08	Lack Of Retry Backoff And Limits When Indexer Is Syncing May Cause Denial-Of-Service (DoS)		
Asset	`op-node/rollup/da/datastore.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When the DA indexer is syncing and thus returns `nil` for `reply.Data` on line [**261**] in `datastore.go`, the current implementation logs an error and immediately returns.

This behavior may cause the caller to repeatedly retry the operation in a tight loop, potentially consuming excessive resources and leading to a denial-of-service (DoS) scenario if the indexer remains slow or stuck.

```
if reply.GetData() == nil {
    lastDataStoreId, err := client.RetrieveLastConfirmDataStoreId(mda.Ctx, &mdar.LastDataStoreIdRequest{}, opt)
    if err != nil {
        log.Error("Retrieve last confirmed data store id fail", "err", err)
        return nil, err
    }
    if dataStoreId < lastDataStoreId.GetDataStoreId() {
        log.Warn("this batch is not confirmed in mantle da,but new batch is confirmed,data corruption exists,need to skip this
      ↪  dataStoreId ", "dataStore id", dataStoreId, "latest dataStore id", lastDataStoreId.GetDataStoreId())
        return nil, nil
    }
    log.Error("frames is nil ,waiting da indexer syncing")
    return nil, fmt.Errorf("frames is nil,maybe da indexer is syncing,need to try again,dataStore id %d", dataStoreId)
}
```

The intended behavior when the indexer is not ready should involve a controlled retry mechanism that employs backoff and/or a maximum retry limit. This would prevent the system from being overwhelmed by constant retry attempts while the underlying indexer issue persists.

Recommendations

Implement the following:

- **Exponential Backoff:** Modify the code to include an exponential backoff mechanism before each retry attempt.
- **Maximum Retry Limit or Timeout:** Introduce a counter for the number of retry attempts or use a context with a deadline/timeout.

Resolution

The finding has been closed with the following comment by the development team:

"The error returned by 'RetrievalFramesFromDaIndexer' is wrapped into NewTemporaryError at op-node/rollup/derive/calldata_source.go L342. This error is then propagated outward eventLoop() at op-node/rollup/driver/state.go L303. The error is identified as an ErrTemporary at L322, triggering a retry with reqStep. Since stepAttempts != 0 at this point, a reattempt with delay will be triggered.".

MEDA-09	Missing Enforcement Of HTTPS/TLS		
Asset	`/op-node/node/client.go, /op-node/rollup/da/datastore.go, /op-service/eigenda/da_proxy.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The current implementation does not enforce HTTPS/TLS on critical network endpoints.

Specifically, the `BeaconAddr` in `op-node/node/client.go` is not checked to ensure it uses HTTPS, and similar omissions are found in the Mantle DA Indexer Socket connection in `op-node/rollup/da/datastore.go` on line [**241**] as well as the connection to `c.proxyUrl` in `op-service/eigenda/da_proxy.go`.

This can result in operators inadvertently running nodes over unsecured channels, opening the system up to Man-in-the-Middle (MitM) attacks, data tampering, and potential service disruptions.

Recommendations

Enforce the following:

- HTTPS for `BeaconAddr`
- HTTPS for DA Proxy Connections
- TLS for DA Indexer Socket

Resolution

The finding has been closed with the following comment from the development team:

> *"BeaconAddr supports both HTTP and HTTPS because Mantle does not provide an L1 Beacon RPC. To allow Mantle Operators running a node without an HTTPS Beacon RPC, BeaconAddr permits HTTP configuration. HTTPS is forced in Mantle's Operators. We can't force 3rd parties to use HTTPS."*

MEDA-10	Insufficient SRS File Length Validation Leading To Potential Out-of-Bounds Reads		
Asset	`op-service/eigenda/config.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The configuration sets `SRSNumberToLoad` to `numBytes / 32` without verifying that the `SRS` file (provided via `G1Path` and `G2PowerOfTauPath`) contains at least that many points.

This could result in attempts to read beyond the available data in the `SRS` file, leading to runtime errors or incomplete data loading during the verification process.

```
kzgCfg := &kzg.KzgConfig{
    G1Path:          c.G1Path,
    G2PowerOf2Path:  c.G2PowerOfTauPath,
    CacheDir:        c.CacheDir,
    SRSOrder:        268435456,     // 2 ^ 32
    SRSNumberToLoad: numBytes / 32, // # of fp.Elements
    NumWorker:       uint64(runtime.GOMAXPROCS(0)),
}
```

Recommendations

Add a check to confirm that the SRS has enough points for the requested load.

Resolution

The finding has been closed as false-positive, the relevant code has already been removed in PR-185.

MEDA-11	Error Handling Uses Explicit `panic()`		
Asset	`op-service/eigenda/config.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The current implementation of the `VerificationCfg()` function directly panics when `GetMaxBlobLength()` returns an error.

This explicit panic can abruptly crash the application if the configuration check fails:

```
func (c *Config) VerificationCfg() *verify.Config {
    numBytes, err := c.GetMaxBlobLength()
    if err != nil {
        panic(fmt.Errorf("Check() was not called on config object, err is not nil: %w", err))
    }
```

A panic can crash the application if not recovered, which can be undesirable for availability.

Recommendations

Replace the panic with error propagation, allowing the caller to handle the error gracefully.

Resolution

The finding has been closed as false-positive, the relevant code has already been removed in PR-185.

MEDA-12	Missing Error Handling in JSON Unmarshaling		
Asset	`code/mantle-v2/op-service/eigenda/da_proxy.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `GetBlobExtraInfo()` function fails to handle errors during JSON unmarshalling, which can result in silent failures and misinterpretation of the extra blob data.

This may lead to unexpected behavior in downstream components that depend on correctly decoded blob information.

```
data, err := io.ReadAll(resp.Body)
 if err != nil {
     return nil, err
 }

 output := make(map[string]interface{})
 json.Unmarshal(data, &output)
```

Recommendations

Modify the `GetBlobExtraInfo()` function to properly handle errors returned by `json.Unmarshal()`:

```
if err := json.Unmarshal(data, &output); err != nil {
    return nil, fmt.Errorf("failed to decode extra info JSON: %w", err)
}
```

Resolution

The finding has been resolved in PR-199.

MEDA-13	Potential Denial-of-Service (DoS) Via Excessive Memory Allocation For SRS Field Elements		
Asset	`op-service/eigenda/config.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When the configured blob size `MaxBlobLength` approaches the upper limit of 1 GB, the computed `SRSNumberToLoad` value becomes extremely large (approximately 32 million field elements).

This can lead to significant memory consumption and processing delays, potentially resulting in a denial-of-service (DoS) condition on nodes that are not equipped to handle such large in-memory datasets:

```
kzgCfg := &kzg.KzgConfig{
    // ... other config fields ...
    SRSNumberToLoad: numBytes / 32, // # of fp.Elements
    // ... remaining config ...
}
```

This design assumes that the environment can handle such an enormous in-memory data structure, which is often not the case for typical nodes.

Without measures to stream or chunk the processing of these field elements, the system risks consuming excessive memory and processing time.

Recommendations

Confirm that the environment can handle loading up to 32 million elements. Otherwise, implement stricter upper blob size limits.

Resolution

The finding has been closed as false-positive, the relevant code has already been removed in PR-185.

MEDA-14	Empty Blob Candidate Generated When First Frame Exceeds Size Limit		
Asset	`op-batcher/batcher/driver_da.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When the very first frame of data exceeds the allowed blob size, i.e. when on the initial iteration `len(nextEncodeData) > se.MaxBlobDataSize * MaxblobNum`, the function does not update the `encodeData` variable.

As a result, the subsequent for-loop that constructs blobs iterates over an empty slice, leading to the creation of an empty blob that is then appended to the transaction candidates.

```
candidates = append(candidates, &txmgr.TxCandidate{
    To:    &l.Rollup.BatchInboxAddress,
    Blobs: blobs,
})
dataInTx = [][]byte{frameData}
encodeData, err = rlp.EncodeToBytes(dataInTx)
if err != nil {
    l.log.Error("op-batcher unable to encode txn", "err", err)
    return nil, err
}
```

The `blobTxCandidates` function is designed to process an array of frame data by accumulating it into a single encoded byte array (`encodeData`). When the accumulated data exceeds a predetermined size threshold, the code should split `encodeData` into chunks (`blobs`) of maximum size `se.MaxBlobDataSize` and create a transaction candidate with these blobs.

An edge case arises if the very first frame's encoded data (`nextEncodeData`) is already larger than the maximum allowed (`se.MaxBlobDataSize * MaxblobNum`). In this scenario, the code checks the size condition and enters the corresponding if block without having updated `encodeData` (which remains empty from its initialization). Consequently, the for-loop that should iterate over `encodeData` :

```
for idx := 0; idx < len(encodeData); idx += se.MaxBlobDataSize {
    // blob creation logic...
}
```

does not run because `len(encodeData) == 0`

The empty blobs slice is then used to build a `txmgr.TxCandidate` , leading to a candidate with no meaningful blob data.

Recommendations

Update the logic in `blobTxCandidates` to ensure that valid data is used when splitting into blobs.

One possible approach is to modify the branch where the size condition is met so that it operates on `nextEncodeData`

rather than the potentially empty `encodeData`.

Resolution

The finding has been resolved in PR-194.

MEDA-15	HTTP Response Body Not Closed on Error Paths		
Asset	`code/mantle-v2/op-service/eigenda/da_proxy.go`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The HTTP request handling in the `RetrieveBlobWithCommitment()` and `GetBlobExtraInfo()` functions fails to close the response body on error paths.

By deferring `resp.Body.Close()` only after checking the response status code, a non-OK status may trigger an early return without closing the body, leading to a resource leak.

```
    resp, err := c.retrieveClient.Do(req)
err = func() error {
    if err != nil {
        return err
    }
    if resp.StatusCode == http.StatusNotFound {
        return ErrNotFound
    }
    if resp.StatusCode != http.StatusOK {
        return fmt.Errorf("failed to get extra info: %v", resp.StatusCode)
    }
    return nil
}()
done(err)
if err != nil {
    return nil, err
}
defer resp.Body.Close()
```

In a long-running or high-load environment, repeated occurrences of this pattern can accumulate unclosed HTTP responses, leading to resource exhaustion.

Recommendations

Ensure that the HTTP response body is closed regardless of the outcome of the status code checks.

One way to achieve this is to call defer `resp.Body.Close()` immediately after a successful HTTP request (i.e., after checking for an error from `Do(req)`), before any processing of `resp.StatusCode`.

Resolution

The development team has advised that this finding will be addressed in the next version.

MEDA-16	Required Casting For `sysCfg.Scalar` Parameter In `EcotoneScalars`		
Asset	`op-service/eth/types.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `EcotoneScalars()` function in `op-service/eth/types.go` calls `CheckEcotoneL1SystemConfigScalar(sysCfg.Scalar)`, but the parameter `sysCfg.Scalar` is of type `Bytes32`, a distinct type defined as type `Bytes32 [32]byte`, whereas the function expects a parameter of type `[32]byte`.

Under the current implementation, the call

```
if err := CheckEcotoneL1SystemConfigScalar(sysCfg.Scalar); err != nil {
    // error handling
}
```

will result in a type mismatch because `sysCfg.Scalar` (of type `Bytes32`) does not meet the expected `[32]byte` parameter type.

The correct approach is to explicitly cast `sysCfg.Scalar` to `[32]byte`.

Recommendations

Update the call within `EcotoneScalars()` to explicitly cast `sysCfg.Scalar` to the expected type `[32]byte`

```
if err := CheckEcotoneL1SystemConfigScalar([32]byte(sysCfg.Scalar)); err != nil {
    // error handling
}
```

Resolution

This finding has been closed as false-positive, no explicit casting is required due to the following assignability rule:

> *"A value x of type V is assignable to variable of type T if V and T have identical underlying types but are not type parameters and at least one of V or T is not a named type."*

MEDA-17	Lack Of `nil` Check In `NodeVersion()` May Lead To Panic		
Asset	`op-service/sources/l1_beacon_client.go`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `NodeVersion()` function in `l1_beacon_client.go` does not validate that `resp.Data` is non-nil before accessing `resp.Data.Version`.

This may lead to a panic if the beacon node returns a malformed or incomplete JSON response.

```
func (cl *BeaconHTTPClient) NodeVersion(ctx context.Context) (string, error) {
    var resp eth.APIVersionResponse
    if err := cl.apiReq(ctx, &resp, versionMethod, nil); err != nil {
        return "", err
    }
    return resp.Data.Version, nil  // @audit, potential panic if resp.Data is nil
}
```

Normally this should not happen in a well defined node beacon implementation, however if the node is misconfigured, returns an error, or responds with partial/malformed JSON, `Data` might be absent or null.

Recommendations

Introduce a `nil` check for `resp.Data` after the JSON decoding step in `NodeVersion()`

Resolution

The development team has advised that this finding will be addressed in the next version.

MEDA-18	`EigenDaClient.DisperseBlob()` Does Not Handle HTTP 503 Error
Asset	`mantle-v2/op-batcher/batcher/driver-da.go, mantle-v2/op-service/eigenda/da_proxy.go`
Status	Open
Rating	Informational

Description

Currently, `DisperseBlob()` function, called by `loopEigenDa()`, does not handle a HTTP 503 error returned by the proxy when it is down:

```
func (c *EigenDAClient) DisperseBlob(ctx context.Context, img []byte) (*disperser.BlobInfo, error) {
    c.log.Info("Attempting to disperse blob to EigenDA")
    if len(img) == 0 {
        return nil, ErrInvalidInput
    }

    body := bytes.NewReader(img)
    url := fmt.Sprintf("%s/put/", c.proxyUrl)
    req, err := http.NewRequestWithContext(ctx, http.MethodPost, url, body)
    if err != nil {
        return nil, fmt.Errorf("failed to create HTTP request: %w", err)
    }
    req.Header.Set("Content-Type", "application/octet-stream")

    done := c.recordInterval("DisperseBlob")
    resp, err := c.disperseClient.Do(req)
    done(err)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()
    if resp.StatusCode != http.StatusOK {
        return nil, fmt.Errorf("failed to store data: %v", resp.StatusCode)
    }

    b, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, err
    }

    comm, err := DecodeCommitment(b)
    if err != nil {
        return nil, err
    }

    blobProof := comm.BlobVerificationProof
    c.log.Info("Dispersed blob to EigenDA successfully", "BatchHeaderHash",
      ↪  hex.EncodeToString(blobProof.BatchMetadata.BatchHeaderHash), "BlobIndex", blobProof.BlobIndex)

    return comm, nil
}
```

Recommendations

Implement 503 handling to allow the system to terminate retries sooner and fall back to an alternative mechanism more quickly.

Resolution

The development team has advised that this finding will be addressed in the next version.

MEDA-19	`SRSOrder` Comment Mismatch
Asset	`op-service/eigenda/config.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The code defines `SRSOrder` as `268435456` with a comment stating it represents `2^32`. However, `268435456` equals `2^28`, not `2^32`.

This discrepancy could either mean that fewer powers are being loaded than intended or that the comment is incorrect.

```
SRSOrder:        268435456,     // 2 ^ 32
SRSNumberToLoad: numBytes / 32, // # of fp.Elements
```

From both a security and correctness standpoint, an erroneous `SRS` size or an incorrect reference to the `SRS`'s order can break the KZG scheme or lead to unexpected cryptographic behavior.

Recommendations

Determine whether the intended `SRS` order is `2^32` or `2^28` and address it accordingly.

Resolution

The finding has been closed as false-positive, the relevant code has already been removed in PR-185.

MEDA-20	Inconsistent Length Check for `MetaTxPrefix`
Asset	`op-geth/core/types/meta_transaction.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The code snippet uses two separate values to determine the length of a meta-transaction prefix. It checks `tx.Data()` against `len(MetaTxPrefix)` and later slices using `MetaTxPrefixLength`.

Although these two values are currently equal, relying on two distinct constants can lead to issues if they ever diverge, potentially causing out-of-bounds slicing or incorrect behavior.

```
if len(tx.Data()) <= len(MetaTxPrefix) {
    return nil, nil
}

if isMantleEverest && bytes.Equal(tx.Data()[:MetaTxPrefixLength], MetaTxPrefix) {
    return nil, ErrMetaTxDisabled
}
```

Recommendations

Consider refactoring the code to rely on a single source of truth for the meta-transaction prefix length.

Resolution

The finding has been acknowledged by the development team and closed without further action being taken.

MEDA-21	Hash Input Considerations
Asset	`publickey.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `newPubKey` focuses solely on constructing a valid ECDSA public key from its coordinate components, ensuring that neither coordinate is `nil` and that the point lies on the P256 curve.

The hash input is assumed to be a securely hashed 32-byte digest (e.g., using SHA-256), however, the code does not enforce any checks on this hash, leaving the responsibility entirely to the caller:

```
// Generates appropriate public key format from given coordinates
func newPublicKey(x, y *big.Int) *ecdsa.PublicKey {
    // Check if the given coordinates are valid
    if x == nil || y == nil || !elliptic.P256().IsOnCurve(x, y) {
        return nil
    }

    return &ecdsa.PublicKey{
        Curve: elliptic.P256(),
        X:     x,
        Y:     y,
    }
}
```

Recommendations

Consider adding the following validations on the input hash:

- Hash is non-nil and exactly 32 bytes in length
- If using protocols requiring domain separation or specific "message prefixing", ensure that is done prior to calling `Verify()`
- If the protocol requires domain separation (e.g., replay protection or context-specific tagging), ensure that is included in the hashed message.
- The hash portion is treated as a 32-byte digest. Ensure, at the protocol level, that this indeed represents a securely hashed message (e.g., SHA-256) of appropriate length.

Resolution

The finding has been closed with the following comment from the development team:

"In the p256Verify precompiled contract, the total length of the input is verified to be 160, if the parameters are abnormal, the verification fails - this ensured that the length of the user provided input parameters meets the requirements."

"In the newPublicKey() method, if the parameters are abnormal, the returned public key is nil. If the public key is nil, the Verify() will fail. This indicates that if the public key provided by the user does not meet the requirements, the verification will directly fail, hence the process is safe".

MEDA-22	Improper Error Handling in Beacon API Initialization Could Result In Silent Misconfiguration
Asset	`op-node/node/node.go`
Status	Open
Rating	Informational

Description

The `initL1BeaconAPI` function logs an error when the `cfg.Beacon` configuration is `nil` but then returns `nil` instead of propagating an error.

This behavior contradicts the comment stating that *"we must have initialized the Beacon API settings"*, potentially impacting post-Ecotone upgrade operations.

```
func (n *OpNode) initL1BeaconAPI(ctx context.Context, cfg *Config) error {
    // Once the Ecotone upgrade is scheduled, we must have initialized the Beacon API settings.
    if cfg.Beacon == nil {
        n.log.Error("missing L1 Beacon API configuration")
        return nil
    }
    ...
}
```

Recommendations

Consider modifying the error handling in the `initL1BeaconAPI()` function so that a missing Beacon API configuration results in an error return.

If backward compatibility is necessary, consider introducing an explicit flag (e.g., `AllowUninitializedBeaconAPI`) that permits operation without the Beacon API for pre-Ecotone scenarios.

Resolution

The development team has advised that this finding will be addressed in the next version.

MEDA-23	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

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

1. **Better Checks On Input gwei**
   Related Asset(s): op-service/eth/ether.go
   In `op-service/eth/ether.go` the `gwei` parameter is `float64`, which could:
   - allow for negative values
   - `.Int(nil)` in `big.Float` will truncate any fractional Wei

   Change the parameter to be `uint64` instead.
2. **Address TODO Comments**
   Related Asset(s): *
   Address TODO comments found throughout the code, e.g. on L12 in `op-service/eigenda/config.go`.
3. **Remain up to date with spec**
   Related Asset(s): *
   Continue remaining up-to-date with any final changes in the `EIP-4844` spec as this may still change.
4. **`SkipEigenDaRpc` Flag Not Restircited**
   Related Asset(s): op-batcher/flags/flags.go
   A `SkipEigenDaRpc` flag used incorrectly in production could effectively bypass EigenDA for an extended period. This might be acceptable for debugging, but consider restricting or protecting it behind a test-only or debug-only build.

Recommendations

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

Resolution

The finding has been acknowledged by the development team and closed without further action being taken.

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.

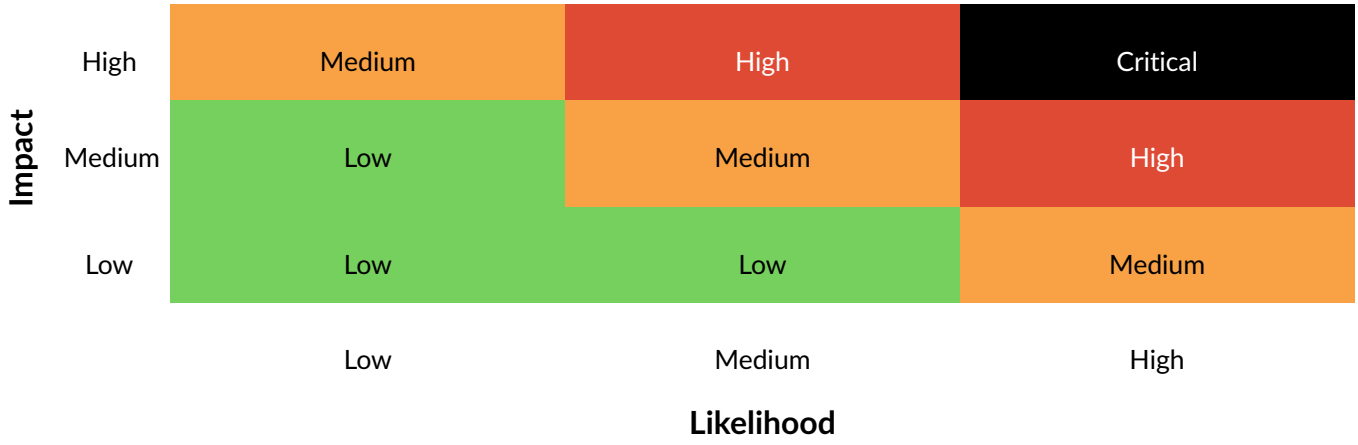

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.