Layr Labs

EigenDA Proxy Secure Integration

Security Assessment Report

Version: 2.0

October, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Layr Labs components in scope. 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 components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

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

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

Overview

EigenDA is a Data Availability (DA) protocol that can be used to temporarily store data.

In this update, Layr Labs has made different changes in the EigenDA proxy in order to make EigenDA compatible with Optimism.

The main modifications affect the two main paths:

- **Write Path**: When the proxy receives data from the OP batcher, it encodes the data into a blob and sends it to EigenDA for processing and verification. Once the blob is aggregated, it is returned to the proxy which converts it into a DACert.
- **Read Path**: This path focuses on the secure integration of a DACert. It involves deriving a rollup payload from a DACert while performing a series of validation checks.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to the following files at commit 066f8ef:

- `api/clients/*`
- `api/proxy/*`
- `contracts/src/integrations/cert/*` with `legacy` folder is out of scope.

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

The fixes of the identified issues were assessed at commit 794c356.

Approach

The security assessment covered components written in Solidity and Golang.

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

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

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

To support the Solidity components of the review, the testing team may use the following automated testing tools:

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

For the Golang components, 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 may use 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 16 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 7 issues.
- Informational: 8 issues.

Detailed Findings

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

A number of additional properties of the components, including 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 components(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
EDA-01	Incorrect HTTP Header Key In JSON Response	Medium	Resolved
EDA-02	Missing Context Cancellation In `RetrieveBlobChunks()`	Low	Resolved
EDA-03	Connection Resource Leak In `GetChunks()`	Low	Resolved
EDA-04	Nil Pointer Dereference In gRPC Connection Cleanup	Low	Resolved
EDA-05	Unlimited Hex String Length Allows DoS Via Memory Exhaustion	Low	Closed
EDA-06	Division By Zero Vulnerability In Payment System	Low	Resolved
EDA-07	Missing Default Case In Chunk Encoding Format Switch	Low	Resolved
EDA-08	Missing Length Validation For Keccak Commitment	Low	Resolved
EDA-09	Incomplete Error Handling For Empty Revert Reasons	Informational	Resolved
EDA-10	Missing Length Validation In Security Parameter Verification Loop	Informational	Resolved
EDA-11	Integer Downcast Without Bounds Checking In EigenDA V1 Certificate Processing	Informational	Resolved
EDA-12	Debug Print Statement In Production Code	Informational	Resolved
EDA-13	Unauthenticated Admin API Endpoints	Informational	Resolved
EDA-14	Inconsistent Maximum Quorum Limits	Informational	Resolved
EDA-15	Missing Failover For Ethereum RPC Call Failures In Certificate Verification	Informational	Resolved
EDA-16	Miscellaneous General Comments	Informational	Resolved

EDA-01	Incorrect HTTP Header Key In JSON Response		
Asset	`api/proxy/server/handlers_misc.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `writeJSON()` function uses the wrong constant as the HTTP header key. This causes the response to set an incorrect header that clients cannot parse properly. The bug is in the `writeJSON()` function:

api/proxy/server/handlers_misc.go::writeJSON()

```
func (svr *Server) writeJSON(w http.ResponseWriter, r *http.Request, response interface{}) {
    jsonData, err := json.Marshal(response)
    if err != nil {
        svr.log.Error("failed to marshal response to json", "method", r.Method, "path", r.URL.Path, "error", err)
        w.WriteHeader(http.StatusInternalServerError)
        _, _ = fmt.Fprintf(w, "failed to marshal response to json: %v", err)
        return
    }

    w.Header().Set(contentTypeJSON, "application/json") // @audit Uses contentTypeJSON instead of headerContentType
    w.WriteHeader(http.StatusOK)
    _, err = w.Write(jsonData)
    if err != nil {
        svr.log.Error("failed to write response", "method", r.Method, "path", r.URL.Path, "error", err)
    }
}
```

The code uses `contentTypeJSON` (which equals `"application/json"`) as the header key instead of `headerContentType` (which equals `"Content-Type"`). These constants are defined at the top of the file:

api/proxy/server/handlers_misc.go

```
const (
    headerContentType = "Content-Type"
    contentTypeJSON = "application/json"
)
```

This creates the HTTP header `"application/json": "application/json"` instead of the correct `"Content-Type": "application/json"`. Clients expect the standard `Content-Type` header to determine response format. Without this header, clients may fail to parse the `JSON` response or treat it as plain text. The same function correctly uses `headerContentType` earlier in `handleSetEigenDADispersalBackend()`, showing this is an inconsistency error.

Recommendations

Change the header key in `writeJSON()` to use the correct constant.

api/proxy/server/handlers_misc.go::writeJSON()

```
w.Header().Set(headerContentType, contentTypeJSON)
```

Resolution

The recommendation has been implemented in PR #2125.

EDA-02	Missing Context Cancellation In `RetrieveBlobChunks()`		
Asset	`api/clients/retrieval_client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The chunk collection loop in `RetrieveBlobChunks()` performs unbuffered channel reads without checking context cancellation, causing goroutine leaks and preventing timely cleanup when blob retrieval operations are cancelled.

The `RetrieveBlobChunks()` function in `api/clients/retrieval_client.go` spawns multiple goroutines to fetch chunks from operators, then collects results in a loop. However, this collection loop uses a direct channel read without a `select` statement to check for context cancellation:

api/clients/retrieval_client.go::RetrieveBlobChunks()

```
// Fetch chunks from all operators
chunksChan := make(chan RetrievedChunks, len(operators))
pool := workerpool.New(r.numConnections)
for opID := range operators {
  opInfo := operators[opID]
  pool.Submit(func() {
    r.nodeClient.GetChunks(ctx, opID, opInfo, batchHeaderHash, blobIndex, quorumID, chunksChan)
  })
}

// ... snip ...

for i := 0; i < len(operators); i++ {
  reply := <-chunksChan  // @audit No select with ctx.Done()
  if reply.Err != nil {
    r.logger.Warn("failed to get chunks from operator", "operator", reply.OperatorID.Hex(), "err", reply.Err)
    continue
  }
  // ... process chunks ...
}
```

If the parent context is cancelled whilst waiting for operator responses, this loop blocks indefinitely rather than returning immediately. Whilst each worker goroutine respects its own timeout via `context.WithTimeout()`, the main goroutine completely ignores the parent context's cancellation signal.

This creates a race condition where context cancellation leads to resource leaks. When a caller cancels the operation (e.g., due to timeout or user cancellation), the main goroutine remains blocked waiting for all operators to respond, even though the caller has moved on. The goroutine and its associated resources remain allocated until all operators eventually respond or time out.

The missing context check has low impact as it prevents timely resource cleanup and causes goroutine leaks. The likelihood is medium as context cancellation is a normal operation pattern. Callers may cancel blob retrievals for various reasons including timeouts, user cancellation, or service shutdown.

Recommendations

Modify the chunk collection loop to use a `select` statement that respects context cancellation:

api/clients/retrieval_client.go::RetrieveBlobChunks()

```
for i := 0; i < len(operators); i++ {
  select {
  case <-ctx.Done():
    return nil, fmt.Errorf("context cancelled whilst retrieving chunks: %w", ctx.Err())
  case reply := <-chunksChan:
    if reply.Err != nil {
      r.logger.Warn("failed to get chunks from operator", "operator", reply.OperatorID.Hex(), "err", reply.Err)
      continue
    }
    // ... process chunks ...
  }
}
```

This allows the function to return immediately when the context is cancelled, enabling proper resource cleanup and preventing goroutine leaks.

Resolution

The issue has been resolved by updating the chunk collection loop in `RetrieveBlobChunks()` to use a `select` statement that checks for context cancellation. The resolution can be found in the PR #2126.

EDA-03	Connection Resource Leak In `GetChunks()`		
Asset	`api/clients/node_client.go, api/clients/retrieval_client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `GetChunks()` function creates gRPC connections to operator nodes but fails to close them, causing connection leaks on every blob retrieval operation that accumulate over time and exhaust available file descriptors.

The `GetChunks()` function in `api/clients/node_client.go` creates a new gRPC connection but never closes it. In contrast, the `GetBlobHeader()` function in the same file properly closes its connection using `defer core.CloseLogOnError(conn, ...)`.

api/clients/node_client.go::GetChunks()

```
func (c client) GetChunks(
  ctx context.Context,
  opID core.OperatorID,
  opInfo *core.OperatorInfo,
  batchHeaderHash [32]byte,
  blobIndex uint32,
  quorumID core.QuorumID,
  chunksChan chan RetrievedChunks,
) {
  conn, err := grpc.NewClient(
    core.OperatorSocket(opInfo.Socket).GetV1RetrievalSocket(),
    grpc.WithTransportCredentials(insecure.NewCredentials()),
  )
  if err != nil {
        chunksChan <- RetrievedChunks{
            OperatorID: opID,
      Err:        err,
      Chunks:     nil,
        }
        return
    }
    // @audit Missing: defer conn.Close() or defer core.CloseLogOnError(conn, ...)

  n := grpcnode.NewRetrievalClient(conn)
  nodeCtx, cancel := context.WithTimeout(ctx, c.timeout)
  defer cancel()

  reply, err := n.RetrieveChunks(nodeCtx, request)
  if err != nil {
    chunksChan <- RetrievedChunks{
      OperatorID: opID,
      Err:        err,
      Chunks:     nil,
    }
    return
  }

  // ... process chunks and send to channel ...
}
```

The `GetChunks()` function is invoked from `RetrieveBlobChunks()` in `api/clients/retrieval_client.go`, which spawns one goroutine per operator. For a quorum with 10 operators, each blob retrieval leaks 10 gRPC connections. These connections remain open until the process terminates, consuming file descriptors and network resources.

The connection leak has medium impact as it causes gradual resource exhaustion. Under normal operation with frequent blob retrievals, the system will eventually exhaust available file descriptors, causing service disruption or complete failure.

The likelihood is medium as the issue occurs on every blob retrieval operation. However, the time to manifest depends on retrieval frequency and system file descriptor limits.

Recommendations

Add proper connection cleanup in `GetChunks()` immediately after connection creation, matching the pattern used in `GetBlobHeader()`:

api/clients/node_client.go::GetChunks()

```
conn, err := grpc.NewClient(
  core.OperatorSocket(opInfo.Socket).GetV1RetrievalSocket(),
  grpc.WithTransportCredentials(insecure.NewCredentials()),
)
if err != nil {
  chunksChan <- RetrievedChunks{
    OperatorID: opID,
    Err:        err,
    Chunks:     nil,
  }
  return
}
defer core.CloseLogOnError(conn, "connection to node client", nil)
```

Resolution

The recommendation has been implemented in PR #2127.

EDA-04	Nil Pointer Dereference In gRPC Connection Cleanup		
Asset	`api/clients/v2/validator/internal/validator_grpc_manager.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `DownloadChunks()` function will panic with a nil pointer dereference if `grpc.NewClient()` fails, as the deferred cleanup function attempts to call `Close()` on a nil connection object.

api/clients/v2/validator/grpc_manager.go::DownloadChunks()

```
func (m *validatorGRPCManager) DownloadChunks(
    ctx context.Context,
    key v2.BlobKey,
    operatorID core.OperatorID,
) (*grpcnode.GetChunksReply, error) {

    // TODO(cody.littley) we can get a tighter bound?
    maxBlobSize := 16 * units.MiB // maximum size of the original blob
    encodingRate := 8             // worst case scenario if one validator has 100% stake
    fudgeFactor := units.MiB      // to allow for some overhead from things like protobuf encoding
    maxMessageSize := maxBlobSize*encodingRate + fudgeFactor

    socket, ok := m.socketMap[operatorID]
    if !ok {
        return nil, fmt.Errorf("operator %s not found in socket map", operatorID.Hex())
    }

    conn, err := grpc.NewClient(
        socket.GetV2RetrievalSocket(),
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithDefaultCallOptions(grpc.MaxCallRecvMsgSize(maxMessageSize)),
    )
    defer func() {
        err := conn.Close() // @audit conn is nil if grpc.NewClient failed
        if err != nil {
            m.logger.Error("validator retriever failed to close connection", "err", err)
        }
    }()
    if err != nil {
        return nil, fmt.Errorf("failed to create connection to operator %s: %w", operatorID.Hex(), err)
    }

    client := grpcnode.NewRetrievalClient(conn)
    request := &grpcnode.GetChunksRequest{
        BlobKey: key[:],
    }

    reply, err := client.GetChunks(ctx, request)
    if err != nil {
        return nil, fmt.Errorf("failed to get chunks from operator %s: %w", operatorID.Hex(), err)
    }

    return reply, nil
}
```

When `grpc.NewClient()` returns an error, `conn` will be nil. The code then enters the early return path, but the deferred function still executes and attempts to call `conn.Close()`, causing a panic.

Recommendations

Move the deferred cleanup function to run after the error check. This ensures the cleanup only happens when a valid connection exists. The fix is simple and prevents the nil pointer crash.

Resolution

The issue has been resolved by relocating the deferred connection cleanup to occur only after confirming that the connection was successfully established. The resolution can be found in the PR #2131.

EDA-05	Unlimited Hex String Length Allows DoS Via Memory Exhaustion		
Asset	`api/proxy/server/routing.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The routing pattern for OP generic endpoints accepts unlimited-length hex strings, potentially allowing denial-of-service attacks through memory exhaustion.

The code at lines [**49-52**] defines a route pattern that accepts hex strings of arbitrary length.

api/proxy/server/routing.go

```
// op generic commitments (write to EigenDA)
subrouterGET.HandleFunc(
    "/"+
        "{optional_prefix:(?:0x)?}"+ // commitments can be prefixed with 0x
        "{"+routingVarNameCommitTypeByteHex+":01}"+ // 01 for generic commitments
        "{da_layer_byte:[0-9a-fA-F]{2}}"+ // should always be 0x00 for eigenDA but we let others through to return a 404
        "{"+routingVarNameVersionByteHex+":[0-9a-fA-F]{2}}"+ // should always be 0x00 for now but we let others through to return a
            ↪ 404
        "{"+routingVarNamePayloadHex+"}", // @audit no length check
    middleware.WithCertMiddlewares(
        svr.handleGetOPGenericCommitment,
        svr.log,
        svr.m,
        commitments.OptimismGenericCommitmentMode,
    ),
)
```

The `routingVarNamePayloadHex` parameter has no upper bound on length. This extracted value is later processed by `hex.DecodeString()` in the `handleGetShared()` function without prior length validation. An attacker could send extremely long hex strings, causing Go to attempt large memory allocations and potentially crash the service through memory exhaustion.

Recommendations

Add length validation to the hex string parameters before processing, or implement regex patterns with reasonable upper bounds.

Resolution

The recommended length validation is not able to be implemented directly without raising potential liveness issues for 3rd party users. This occurs as the protocol does not have any direct requirements for certificate lengths and therefore any arbitrary length could be valid. The potential liveness issues for 3rd party users are rated as higher risk than the unvalidated hex length. As such no changes have been implemented.

Further discussions can be found in issue #2168.

EDA-06	Division By Zero Vulnerability In Payment System		
Asset	`api/clients/v2/accountant.go, contracts/src/core/PaymentVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

A division by zero vulnerability exists in the payment system that can cause immediate client crashes when processing payment operations. The vulnerability occurs when the smart contract's `reservationPeriodInterval` parameter is set to zero, which gets synchronised to the Go client as `reservationWindow`. The issue occurs in the `getOrRefreshRelativePeriodRecord()` function in `accountant.go`:

```
func (a *Accountant) getOrRefreshRelativePeriodRecord(index uint64, reservationWindow uint64) *PeriodRecord {
    relativeIndex := uint32((index / reservationWindow) % uint64(len(a.periodRecords))) // @audit Division by zero if
        ↪  reservationWindow == 0
    if relativeIndex >= uint32(len(a.periodRecords)) {
        panic(fmt.Sprintf("relativeIndex %d is greater than the number of bins %d cached", relativeIndex, len(a.periodRecords)))
    }
    if a.periodRecords[relativeIndex].Index < uint32(index) {
        a.periodRecords[relativeIndex] = PeriodRecord{
            Index: uint32(index),
            Usage: 0,
        }
    }

    return &a.periodRecords[relativeIndex]
}
```

The `reservationWindow` value originates from the `reservationPeriodInterval` parameter in `PaymentVault.sol`, which can be set through the `setReservationPeriodInterval()` function:

```
function setReservationPeriodInterval(uint64 _reservationPeriodInterval) external onlyOwner {
    emit ReservationPeriodIntervalUpdated(reservationPeriodInterval, _reservationPeriodInterval);
    reservationPeriodInterval = _reservationPeriodInterval;
}
```

The smart contract lacks validation to prevent `reservationPeriodInterval` from being set to zero. When the Go client synchronises onchain state via `SetPaymentState()`, it directly assigns this value without validation, creating a crash vector for any subsequent payment operation.

The likelihood is classified as low because contract administrators are unlikely to intentionally set the reservation period to zero based on the payment model design. However, this could occur during testing phases, contract deployment, or human error during configuration.

Recommendations

Add validation in the `SetPaymentState()` function to reject zero values for `reservationWindow` or add validation in the smart contract's `setReservationPeriodInterval()` function to prevent zero values at the source.

Resolution

The issue has been resolved by validating `reservationWindow` is non-zero in PR #2157.

EDA-07	Missing Default Case In Chunk Encoding Format Switch		
Asset	`api/clients/node_client.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `GetChunks()` function in `api/clients/node_client.go` lacks a default case in its switch statement that handles chunk encoding formats. If a future protobuf version adds a new encoding format (e.g., `ZSTD = 3`) and operator nodes are upgraded before the client code, the switch statement will fall through with both `chunk` and `err` remaining `nil`. This causes `nil` to be assigned to `chunks[i]` at line [**150**], which could lead to nil pointer dereferences downstream.

api/clients/node_client.go:124

```
switch reply.GetChunkEncodingFormat() {
case grpcnode.ChunkEncodingFormat_GNARK:
    chunk, err = new(encoding.Frame).DeserializeGnark(data)
case grpcnode.ChunkEncodingFormat_GOB:
    chunk, err = new(encoding.Frame).Deserialize(data)
case grpcnode.ChunkEncodingFormat_UNKNOWN:
    chunk, err = new(encoding.Frame).Deserialize(data)
    // ... error handling
// @audit Missing default case
}
if err != nil { /* ... */ }
chunks[i] = chunk  // @audit Can assign nil if no case matched
```

The likelihood is low as it requires a version mismatch between protobuf definitions and client code. The impact is medium as it causes service disruption via panic but no data corruption or security compromise.

Recommendations

Add a default case that returns an error for unhandled encoding formats:

api/clients/node_client.go::GetChunks()

```
default:
    chunksChan <- RetrievedChunks{
        OperatorID: opID,
        Err:        fmt.Errorf("unsupported chunk encoding format: %v",
                   reply.GetChunkEncodingFormat()),
        Chunks:     nil,
    }
    return
```

Resolution

The recommendation has been implemented in PR #2158.

EDA-08	Missing Length Validation For Keccak Commitment		
Asset	`api/proxy/server/handlers_cert.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `handleGetOPKeccakCommitment()` function does not validate that the decoded Keccak commitment is exactly 32 bytes, allowing malformed requests to be processed until they fail at a later stage.

The function decodes the hex-encoded Keccak commitment from the request path but does not verify its length before passing it to the storage layer. Keccak256 hashes are always 32 bytes, and the system expects this invariant to hold throughout the processing pipeline.

api/proxy/server/handlers_cert.go::handleGetOPKeccakCommitment()

```
func (svr *Server) handleGetOPKeccakCommitment(w http.ResponseWriter, r *http.Request) error {
  keccakCommitmentHex, ok := mux.Vars(r)[routingVarNameKeccakCommitmentHex]
  if !ok {
    return proxyerrors.NewParsingError(fmt.Errorf("keccak commitment not found in path: %s", r.URL.Path))
  }
  keccakCommitment, err := hex.DecodeString(keccakCommitmentHex) // @audit no length check
  if err != nil {
    return proxyerrors.NewParsingError(
      fmt.Errorf("failed to decode hex keccak commitment %s: %w", keccakCommitmentHex, err))
  }
```

When an attacker provides a commitment of incorrect length, the request proceeds to `GetOPKeccakValueFromS3()`, which uses the malformed key to construct an S3 object path. The S3 `Get()` operation will likely return `ErrKeccakKeyNotFound` since no object exists at the malformed path, but the error message does not clearly indicate that the commitment format was invalid.

The impact is low because the subsequent verification in `GetOPKeccakValueFromS3()` provides defence in depth through its Keccak verification check, and S3 lookups with malformed keys will return not found errors. The likelihood is low because this requires a malformed client request, and the impact is limited to error message clarity rather than security compromise.

Recommendations

Add explicit length validation after hex decoding the Keccak commitment in both `handleGetOPKeccakCommitment()` and `handlePutOPKeccakCommitment()` functions. Return a clear parsing error when the commitment length is not exactly 32 bytes.

api/proxy/server/handlers_cert.go

```
keccakCommitment, err := hex.DecodeString(keccakCommitmentHex)
if err != nil {
    return proxyerrors.NewParsingError(
        fmt.Errorf("failed to decode hex keccak commitment %s: %w", keccakCommitmentHex, err))
}
if len(keccakCommitment) != 32 {
    return proxyerrors.NewParsingError(
        fmt.Errorf("keccak commitment must be 32 bytes, got %d bytes", len(keccakCommitment)))
}
```

Resolution

Length validation has been added as per the recommendation in PR #2162.

EDA-09	Incomplete Error Handling For Empty Revert Reasons
Asset	`cert/EigenDACertVerifier.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract's error handling logic checks for empty revert reasons but does not account for all cases that produce empty reasons, potentially leading to incomplete error classification.

The code handles caught exceptions with empty reasons.

```
    // which means a bug or misconfiguration of the CertVerifier contract itself.
    return uint8(StatusCode.INTERNAL_ERROR);
} catch (bytes memory reason) {
    if (reason.length == 0) {
        // @audit handles empty reason case
```

While the current codebase does not contain functions that use `revert()` or `revert(0,0)`, both of these constructs produce empty revert reasons that would be caught by this condition. The comment suggests this is intended to handle bugs or misconfigurations, but the logic assumes empty reasons only occur from internal errors, when they could also result from intentional reverts with no data.

Recommendations

Consider adding a comment indicating to check whether there is any of `revert()` or `revert(0,0)` if any updates happen in the codebase, or implement more specific error classification to distinguish between these cases if needed.

Resolution

The development team are aware of the issue and documented it for user reference.

EDA-10	Missing Length Validation In Security Parameter Verification Loop
Asset	`api/proxy/store/generated_key/eigenda/verify/verifier.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

A security parameter verification loop assumes equal array lengths without validation, potentially causing array bounds violations if the arrays have mismatched sizes.

The code iterates through security parameters without validating array length equality.

api/proxy/store/generated_key/eigenda/verify/verifier.go::verifySecurityParams()

```
// require that the security param in each blob is met
for i := 0; i < len(blobHeader.QuorumBlobParams); i++ {
    if batchHeader.GetQuorumNumbers()[i] != blobHeader.QuorumBlobParams[i].QuorumNumber {
```

The loop uses `len(blobHeader.QuorumBlobParams)` as the upper bound but accesses `batchHeader.GetQuorumNumbers()[i]` without verifying that the batch header's quorum numbers array has the same length. If `batchHeader.GetQuorumNumbers()` has fewer elements than `blobHeader.QuorumBlobParams`, this will cause an array bounds panic when it exceeds the available indices.

Recommendations

Add length validation before the loop.

Resolution

The development team have added length validation as per the recommendation in PR #2184.

EDA-11	Integer Downcast Without Bounds Checking In EigenDA V1 Certificate Processing
Asset	`api/proxy/store/generated_key/eigenda/verify/certificate.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

A `uint32` value from protobuf is downcast to `uint8` without bounds checking, potentially causing data truncation if the source value exceeds 255.

The code performs an unchecked integer downcast.

api/proxy/store/generated_key/eigenda/verify/verifier.go

```
qps := make([]QuorumBlobParam, len(c.BlobHeader.GetBlobQuorumParams()))
for i, qp := range c.BlobHeader.GetBlobQuorumParams() {
    qps[i] = QuorumBlobParam{
        QuorumNumber: uint8(qp.GetQuorumNumber()), // #nosec G115
```

The `qp.GetQuorumNumber()` function returns a `uint32` value from the protobuf definition, but it is directly cast to `uint8` without validating that the value is within the valid range (0-255). If a malformed or malicious protobuf message contains a quorum number greater than 255, the cast will silently truncate the value, potentially leading to incorrect quorum processing.

Recommendations

Add bounds checking before the downcast.

Resolution

The development team have added bounds checking by implementing a check against the constant `MaxQuorumID = 191` in PR #2108.

EDA-12	Debug Print Statement In Production Code
Asset	`api/proxy/server/routing.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

A debug print statement exists in production code that could potentially leak sensitive information through application logs.

The code at line [**170**] contains a `fmt.Println()` statement within the `parseReturnEncodedPayloadQueryParam()` function.

api/proxy/server/routing.go::parseReturnEncodedPayloadQueryParam()

```
func parseReturnEncodedPayloadQueryParam(r *http.Request) bool {
    returnEncodedPayloadValues, exists := r.URL.Query()["return_encoded_payload"]
    if !exists || len(returnEncodedPayloadValues) == 0 {
        return false
    }
    fmt.Println("returnEncodedPayloadValues:", returnEncodedPayloadValues) // @audit debug statement
    returnEncodedPayload := strings.ToLower(returnEncodedPayloadValues[0])
    if returnEncodedPayload == "" || returnEncodedPayload == "true" || returnEncodedPayload == "1" {
        return true
    }
    return false
}
```

This debug statement prints query parameter values, which may be captured by logging systems and could potentially expose request parameters or reveal internal application behaviour to unauthorised parties.

Recommendations

Remove the debug `fmt.Println()` statement from production code.

Resolution

The development team have removed the debug print statement in PR #2185.

EDA-13	Unauthenticated Admin API Endpoints
Asset	`api/proxy/server/routing.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The admin API endpoints are exposed without any authentication mechanism, allowing unauthorised access to administrative functionality that can modify the EigenDA dispersal backend configuration.

The code in `routing.go` registers admin endpoints without authentication middleware:

api/proxy/server/routing.go

```
if svr.config.IsAPIEnabled(AdminAPIType) {
    svr.log.Warn("Admin API endpoints are enabled")
    // Admin endpoints to check and set EigenDA backend used for dispersal
    r.HandleFunc("/admin/eigenda-dispersal-backend",
svr.handleGetEigenDADispersalBackend).Methods("GET")
    r.HandleFunc("/admin/eigenda-dispersal-backend",
svr.handleSetEigenDADispersalBackend).Methods("PUT")
}
```

The comment at lines [**122-124**] acknowledges this as a known issue but dismisses it based on the assumption that the proxy is not meant to be exposed publicly. However, this creates a security risk where unauthorised users can read and modify the EigenDA dispersal backend configuration through the `/admin/eigenda-dispersal-backend` endpoints.

Recommendations

Implement authentication middleware for admin endpoints.

Resolution

The development team were aware of the issue and noted this is highlighted in the documentation in Admin Routes.

EDA-14	Inconsistent Maximum Quorum Limits
Asset	`core/v2/types.go`, `contracts/src/core/EigenDARegistryCoordinatorStorage.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

There is a discrepancy between the maximum quorum limits defined in the offchain Go code and what is enforced onchain in the Solidity contracts.

The Go code in `core/v2/types.go` defines:

core/v2/types.go

```
const (
    // We use uint8 to count the number of quorums, so we can have at most 255 quorums,
    // which means the max ID can not be larger than 254 (from 0 to 254, there are 255
    // different IDs).
    MaxQuorumID = 254
)
```

However, the onchain contract enforces a lower limit:

contracts/lib/eigenlayer-middleware/src/RegistryCoordinatorStorage.sol

```
uint8 internal constant MAX_QUORUM_COUNT = 192;
```

This creates a mismatch where the offchain code suggests support for up to 255 quorums (IDs 0-254), but the contract will only allow up to 192 quorums. Whilst the offchain code does perform validation against `MaxQuorumID = 254`, operations that attempt to use quorum IDs in the range 192-254 will succeed offchain but fail when interacting with the onchain registry coordinator contract.

Recommendations

Update the offchain `MaxQuorumID` constant to `191` to reflect the actual onchain limit of 192 quorums (IDs 0-191) for consistency and to prevent confusion during development and operations.

Resolution

The development team have added bounds checking by implementing a check against the constant `MaxQuorumID = 191` in PR #2108.

EDA-15	Missing Failover For Ethereum RPC Call Failures In Certificate Verification
Asset	`api/clients/v2/verification/cert_verifier.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Ethereum RPC call failures in certificate verification are treated as `CertVerifierInternalError` without triggering a failover mechanisms. These errors will propagate as a 500 response. For the Optimism case, this would result in the node waiting briefly, and then retries the `/GET` request to fetch a certificate. As a result, the issue presents as an informational severity item that causes short-lived delays rather than a sustained outage.

The code at lines [**94-100**] performs an Ethereum contract call without proper error classification.

api/clients/v2/verification/cert_verifier.go::CheckDACert()

```
returnData, err := cv.ethClient.CallContract(ctx, ethereum.CallMsg{
    To:   &certVerifierAddr,
    Data: callMsgBytes,
}, nil)
if err != nil {
    return &CertVerifierInternalError{Msg: "checkDACert eth call", Err: err}
}
```

Recommendations

Consider classifying RPC call failures as failover-eligible errors to enable retry mechanisms.

Resolution

The issue has been resolved by documenting the behaviour and instructing users to configure and external failover mechanism in PR #2211.

EDA-16	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **Silent Failure When Chain Length Is Insufficient For Confirmation Depth**

   Related Asset(s): api/proxy/store/generated_key/eigenda/verify/cert.go

   The code silently returns `0` when the blockchain is shorter than the required confirmation depth, instead of returning an explicit error indicating the chain is too short for safe operation.

   The code in cert.go contains a condition that handles insufficient chain length without proper error reporting.

   api/proxy/store/generated_key/eigenda/verify/cert.go

   ```
   if err != nil {
       return nil, fmt.Errorf("failed to get latest block number: %w", err)
   }
   if blockNumber < cv.ethConfirmationDepth {
       // @audit silently returns 0 instead of error
       return 0, nil
   }
   ```

   When the current block number is less than the configured confirmation depth, the function returns `0` without indicating why this occurred. This can lead to confusion during debugging or when operating on short test chains where the confirmation depth exceeds the total chain length.

   Return an explicit error when the chain length is insufficient.

2. **Incorrect Comment About Certificate Version**

   Related Asset(s): api/proxy/server/routing.go

   A comment incorrectly states that a certificate version parameter should always be `0x00`, when it actually represents the certificate version and can have multiple valid values.

   The code contains an inaccurate comment about the version byte.

   api/proxy/server/routing.go

   ```
   "{optional_prefix:(?:0x)?}"+ // commitments can be prefixed with 0x
   "{"+routingVarNameCommitTypeByteHex+":01}"+ // 01 for generic commitments
   "{da_layer_byte:[0-9a-fA-F]{2}}"+ // should always be 0x00 for eigenDA but we let others through to return a 404
   "{"+routingVarNameVersionByteHex+":[0-9a-fA-F]{2}}"+ // should always be 0x00 for now but we let others through to return a
       ↪ 404
   ```

   The comment states that `routingVarNameVersionByteHex` "should always be 0x00 for now", but this parameter represents the certificate version and can legitimately have different values (e.g., `0x00` for `v0`, `0x01` for `v1`, `0x02` for `v2`). The comment creates confusion about the expected behaviour and valid values for this routing parameter.

   Update the comment to accurately reflect that this parameter represents the certificate version and can have multiple valid values.

3. **Incorrect Error Message In EigenDA V2 Certificate Decoding**

   Related Asset(s): api/proxy/store/generated_key/v2/eigenda.go

   An error message incorrectly references "EigenDA v1 cert" when the code is actually decoding an EigenDA v2 certificate, causing confusion during debugging.

   The code contains a mismatched error message when RLP decoding fails.

   api/proxy/store/generated_key/v2/eigenda.go

   ```
   err := rlp.DecodeBytes(versionedCert.SerializedCert, &eigenDACertV2)
   if err != nil {
       return coretypes.NewCertParsingFailedError(
           hex.EncodeToString(versionedCert.SerializedCert), fmt.Sprintf("RLP decoding EigenDA v1 cert: %v", err)) // @audit
                ↪  should be "v2 cert"
   ```

   The error message states "RLP decoding EigenDA v1 cert" but the variable being decoded is `eigenDACertV2`, indicating this is actually v2 certificate processing. This inconsistency can mislead developers and operators when troubleshooting certificate parsing failures.

   Correct the error message to accurately reflect the certificate version being processed.

4. **Spelling Mistakes In Comments**

   Related Asset(s): api/proxy/server/server.go, api/clients/v2/validator/validator_client.go

   Multiple spelling mistakes were found in code comments throughout the codebase, reducing code professionalism and potentially causing confusion for developers.

   The following spelling errors were identified:

   - In `api/proxy/server/server.go:132`, the comment contains "neeed" instead of "need":

     api/proxy/server/server.go

     ```
     // POST routes parse them by hand because they neeed to send the entire
     // request (including the type/version header bytes) to the server.
     ```

   - In `api/clients/v2/validator/validator_client.go:176`, the comment contains "occurence" instead of "occurrence":

     api/clients/v2/validator/validator_client.go

     ```
     // If an operator is encountered multiple times, it uses the socket
     // corresponding to the first occurence.
     ```

   While these typos have no functional impact, they detract from code quality and professionalism. Clear, correctly spelled comments improve code maintainability and developer experience.

   Correct these spelling mistakes to maintain code quality standards.

5. **Unnecessary Empty Line In Multi-line Comment**

   Related Asset(s): api/clients/v2/coretypes/eigenda_cert.go

   A multi-line comment contains an unnecessary empty line that disrupts the visual flow and formatting consistency of the documentation.

   The comment at line 218 contains an extra blank line:

   api/clients/v2/coretypes/eigenda_cert.go

   ```
   // This struct represents the composition of an EigenDA V2 certificate
   // NOTE: This type is hardforked from the V3 type and will no longer
   //
   //  be supported for dispersals after the CertV3 hardfork
   type EigenDACertV2 struct {
   ```

   The empty line after "will no longer" breaks the natural flow of the comment. This should be a continuous multi-line comment without the unnecessary blank line separator.

   Remove the empty line to maintain consistent comment formatting throughout the codebase.

Recommendations

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

Resolution

The development team's responses to the raised issues can be seen in PR #2217.

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.

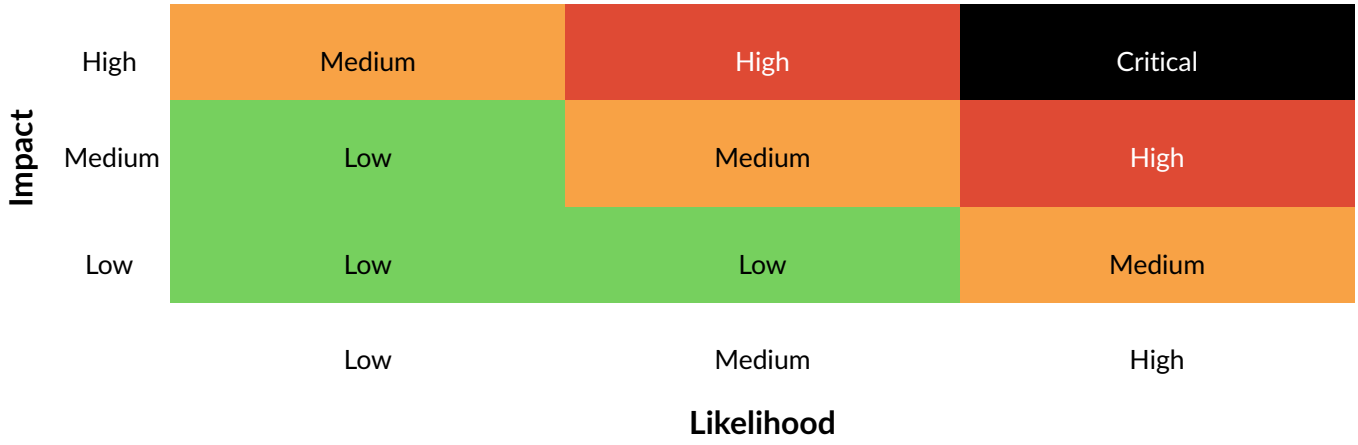

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

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

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