Layr Labs

EigenDA Offchain

Security Assessment Report

Version: 2.0

April, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Layr Labs Golang program. The review focused solely on the security aspects of the Golang source code, 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 Golang program. 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 Golang program contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were used during this assessment are available upon request.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Layr Labs Golang program.

Overview

EigenDA is the premier Data Availability service built on top of Ethereum using EigenLayer for restaking designed by EigenLabs. Restakers using EigenLayer will be able to delegate their staked ETH to node operators working on EigenDA. These node operators will perform validation of data posted to the EigenDA service. Allowing a level of confidence as to the correctness of posted data. This will enable layer 2 rollups to move DA away from Ethereum L1 and onto EigenDA, reducing transaction costs while enabling higher transaction throughput.

EigenDA achieves this security and performance by splitting proposed blobs into multiple chunks using erasure coding, these chunks are then distributed to EigenDA nodes who verify the correctness of the data they receive using a KZG commit/proof system. Proof-of-Custody is then used to ensure node operators truthfully store their allocated blob chunks, this system works by requiring regular computation from node operators that can only be calculated if they possess the data they were allocated to store. Failure to do so results in the slashing of that node's ETH.

Client rollups also benefit from the increased predictability of transaction costs due to the decoupling from the Ethereum Layer 1 gas market as well as the ability to settle transaction costs in their native token and set customisable parameters as they deem fit, such as a dual quorum staking where their own token is also staked to guarantee data availability.

Security Assessment Summary

Scope

The scope of this time-boxed review was strictly limited to files at commit 91838ba.

Note: third party libraries and dependencies, such as gnark-crypto, were excluded from the scope of this assessment.

Approach

The review was conducted on the files hosted on the LayrLabs EigenDA repository at commit 91838ba.

Retesting activities were performed on commit 8f24e8a.

The manual code review section of the report is focused on identifying issues/vulnerabilities associated with the business logic implementation of the components in scope.

For the Golang libraries and modules, this includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime and use of the Ethereum protocol.

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

- `golangci-lint`: `https://github.com/golangci/golangci-lint`
- `semgrep-go`: `https://github.com/dgryski/semgrep-go`
- `go-geiger`: `https://github.com/jlauinger/go-geiger`
- `libfuzzer`: `https://github.com/mdempsky/go114-fuzz-build`
- `go-fuzz`: `https://github.com/dvyukov/go-fuzz/go-fuzz-build`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorised by their severity:

- Critical: 3 issues.
- High: 3 issues.
- Medium: 2 issues.
- Low: 1 issue.
- Informational: 1 issue.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Layr Labs smart contracts. 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
EDA-01	Lack Of Nil Checks On Parameters Of Incoming Requests	Critical	Resolved
EDA-02	Missing `IsOnCurve` & `IsInSubgroup` Checks For Elliptic Curve Points	Critical	Resolved
EDA-03	Unvalidated Batch Root	Critical	Resolved
EDA-04	Unsafe Downcasting Of Integers	High	Resolved
EDA-05	Length Proofs Always Allows Appending Zero Field Elements To Change Length	High	Closed
EDA-06	Index Out Of Bounds Panics	High	Resolved
EDA-07	Insufficient Arithmetic Checks	Medium	Closed
EDA-08	Lack of Transport Layer Encryption	Medium	Closed
EDA-09	Additional Checks Required	Low	Closed
EDA-10	Miscellaneous General Comments	Informational	Closed

EDA-01	Lack Of Nil Checks On Parameters Of Incoming Requests		
Asset	`node/grpc/utils.go, node/node.go, disperser/encoder/server.go, disperser/apiserver/server.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Node and Disperser do not implement `nil` checks on parameters of incoming requests, leading to unhandled `nil` pointer dereference panics and, subsequently, unexpected crashes.

Protobuf decoding allows any fields which are pointers to decode into `nil` by default. Therefore, any structs which are a pointer or contain fields which are pointers may decode to `nil`.

Node

The following code in `node/grpc/utils.go` line [19] will result in `nil` pointer dereference if `BatchHeader` is `nil`:

```
// Constructs a core.BatchHeader from a proto of pb.StoreChunksRequest.
func GetBatchHeader(in *pb.StoreChunksRequest) (*core.BatchHeader, error) {
    var batchRoot [32]byte
    copy(batchRoot[:], in.GetBatchHeader().GetBatchRoot())
    batchHeader := core.BatchHeader{
    ReferenceBlockNumber: uint(in.BatchHeader.ReferenceBlockNumber), // @audit this could nil reference if BatchHeader is nil. Use
        ↪  GetBatchHeader() instead and GetReferenceBlockNumber()
    BatchRoot:            batchRoot,
    }
    return &batchHeader, nil
}
```

Similarly, on line [41] if `GetQuorumHeaders()` is `nil`, then the enumerating through it will panic:

```
quorumID := blob.GetHeader().GetQuorumHeaders()[i].QuorumId // @audit this could nil dereference too, if GetQuorumHeaders() is nil,
    ↪  then `[i]` and `.QuorumId` reference will crash
```

Furthermore, there are no checks on line [61] to ensure `BlobHeader` or `X` and `Y` are not `nil`:

```
func GetBlobHeaderFromProto(h *pb.BlobHeader) (*core.BlobHeader, error) {
    commitX := new(fp.Element).SetBytes(h.GetCommitment().GetX()) // @audit if h (BlobHeader) is nil, GetCommitment() will nil
        ↪  dereference.
    commitY := new(fp.Element).SetBytes(h.GetCommitment().GetY()) // @audit as above
```

In `node/node.go` on line [266] there is no check for `rawBlobs` being `nil` in `ProcessBatch()` function, called from `handleStoreChunksRequest()`, which will then cause a panic in `store.go` on line [227]:

```
blobHeaderBytes, err := proto.Marshal(blobsProto[idx].GetHeader()) // @audit this could nil dereference
```

Disperser

`EncodingParams` in `disperser/encoder/server.go` line [79] is not checked, which could `nil` dereference as follows:

```
// Convert to core EncodingParams
var encodingParams = encoding.EncodingParams{
    ChunkLength: uint64(req.EncodingParams.ChunkLength), // @audit EncodingParams could be nil, and cause nil dereference panic
    NumChunks:   uint64(req.EncodingParams.NumChunks), // @audit as above
}
```

On an authenticated challenge responses, the following could `nil` dereference via `DisperseBlobAuthenticated()` in `disperser/apiserver/server.go` line [143]:

```
blob.RequestHeader.AuthenticationData = challengeReply.AuthenticationData.AuthenticationData // @audit `.AuthenticationData` could
    ↪ be nil and result in nil dereference panic
```

In the following code from line [84], `req.Data` could also be `nil`, however, at the time of testing, it did not appear that it will cause any direct crashes and, instead, will result in errors thrown further down in the execution chain. Nonetheless, an explicit check before calling the `EncodeAndProve()` function is advised, which could also slightly improve performance:

```
commits, chunks, err := s.prover.EncodeAndProve(req.Data, encodingParams)
```

Recommendations

Validate incoming requests for expected values and implement `nil` checks on all request parameters, particularly for ones that are defined as struct pointers or with `omitempty` protobuf.

Consider building a generic recursive function that will check all protobuf decoded structus and their fields for `nil` pointers.

Resolution

The issue has been resolved by adding `nil` checks in PR #427.

EDA-02	Missing `IsOnCurve` & `IsInSubgroup` Checks For Elliptic Curve Points		
Asset	`core/bn254/attestation.go, encoding/serialization.go, node/grpc/utils.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

There are no checks to ensure that elliptic curve points for BN254 are on the curve and in the correct subgroup.

Points which are not on the curve will have undefined addition and scalar multiplication operations.

For points not in the correct subgroup multiplications such as those in `MultiExp()` may or may not return a point in the correct subgroup, depending on the point and scalar multiplier. Furthermore, the pairing operation has undefined behaviour on points outside the subgroup.

The issue is rated as high likelihood as it operates directly on untrusted user data received from GRPC. The impact is rated as high as it performs undefined curve operations that may or may not pass pairing checks.

core/bn254/attestation.go

The following deserialisation operations do not check if points are on the curve or in the correct subgroup after decoding. Additionally, there are no checks that each field element `e` is within the range `0 <= e < p`. This allows multiple encodings of the same point.

```
func DeserializeG1(b []byte) *bn254.G1Affine {
    p := new(bn254.G1Affine)
    p.X.SetBytes(b[0:32]) // @audit may be larger than p
    p.Y.SetBytes(b[32:64]) // @audit  may be larger than p
    return p // @audit missing IsInSubgroup
}

// ... snipped

func DeserializeG2(b []byte) *bn254.G2Affine {
    p := new(bn254.G2Affine)
    p.X.A0.SetBytes(b[0:32]) // @audit  may be larger than p
    p.X.A1.SetBytes(b[32:64]) // @audit  may be larger than p
    p.Y.A0.SetBytes(b[64:96]) // @audit  may be larger than p
    p.Y.A1.SetBytes(b[96:128]) // @audit  may be larger than p
    return p // @audit missing IsInSubgroup
}
```

encoding/serialization.go

The following functions use the `encoding/gob` decoder which recursively decodes items into the structs without performing additional checks. Therefore, it is lacking checks to ensure the validity of points.

```
func (c *G1Commitment) Deserialize(data []byte) (*G1Commitment, error) {
    err := decode(data, c) // @audit missing IsInSubgroup and field element checks
    return c, err
}

func (c *G1Commitment) UnmarshalJSON(data []byte) error {
    var g1Point bn254.G1Affine
    err := json.Unmarshal(data, &g1Point) // @audit will perform canonical and subgroup checks
    if err != nil {
        return err
    }
    c.X = g1Point.X
    c.Y = g1Point.Y
    return nil
}

// ... snipped

func (c *G2Commitment) Deserialize(data []byte) (*G2Commitment, error) {
    err := decode(data, c) // @audit missing IsInSubgroup and field element checks
    return c, err
}

func (c *G2Commitment) UnmarshalJSON(data []byte) error {
    var g2Point bn254.G2Affine
    err := json.Unmarshal(data, &g2Point)  // @audit will perform canonical and subgroup checks
    if err != nil {
        return err
    }
    c.X = g2Point.X
    c.Y = g2Point.Y
    return nil
}
```

node/grpc/utils.go

The function `SetBytes()` for `fp.Element` does not perform canonical checks to ensure the decoded bytes are less than the field modulus. Moreover, there are no checks that points are on the curve or within the correct subgroup.

The following decoding of commitments, length commitments and proofs do not guarantee valid points.

```
func GetBlobHeaderFromProto(h *pb.BlobHeader) (*core.BlobHeader, error) {
    commitX := new(fp.Element).SetBytes(h.GetCommitment().GetX())
    commitY := new(fp.Element).SetBytes(h.GetCommitment().GetY())
    commitment := &encoding.G1Commitment{ // @audit missing field element, point on curve and subgroup checks
        X: *commitX,
        Y: *commitY,
    }
    var lengthCommitment, lengthProof encoding.G2Commitment // @audit missing field element, point on curve and subgroup checks
    if h.GetLengthCommitment() != nil {
        lengthCommitment.X.A0 = *new(fp.Element).SetBytes(h.GetLengthCommitment().GetXA0())
        lengthCommitment.X.A1 = *new(fp.Element).SetBytes(h.GetLengthCommitment().GetXA1())
        lengthCommitment.Y.A0 = *new(fp.Element).SetBytes(h.GetLengthCommitment().GetYA0())
        lengthCommitment.Y.A1 = *new(fp.Element).SetBytes(h.GetLengthCommitment().GetYA1())
    }
    if h.GetLengthProof() != nil {
        lengthProof.X.A0 = *new(fp.Element).SetBytes(h.GetLengthProof().GetXA0())
        lengthProof.X.A1 = *new(fp.Element).SetBytes(h.GetLengthProof().GetXA1())
        lengthProof.Y.A0 = *new(fp.Element).SetBytes(h.GetLengthProof().GetYA0())
        lengthProof.Y.A1 = *new(fp.Element).SetBytes(h.GetLengthProof().GetYA1())
    }

    // ...
```

Recommendations

It is recommended to perform checks on each:

- field element such that it is in the desired range;
- point is on the curve; AND
- point is in the correct subgroup.

This can be achieved by using the `G1Affine.SetBytes()` and `G2Affine.SetBytes()` functions from `gnark-crypto` which perform the required checks. Note that these functions allow for both compressed and uncompressed encodings.

Resolution

PR #422 introduces a number of changes to resolve the issue. First, deserialisation methods now make use of the relevant `gnark-crypto` `SetBytes()` functions. Furthermore, subgroup and on curve checks were added to GRPC functions to ensure valid points are consumed.

EDA-03	Unvalidated Batch Root		
Asset	`node/node.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Nodes do not validate `BatchHeader.BatchRoot` matches the merkle root of `BlobHeaders`. As a result during `Node.StoreChunks()` the validated blobs may not match `batchHeaderHash`.

A malicious disperser, or user with access to the `Node.StoreChunks()` endpoints, may call the endpoint with a set of blobs which are not associated to the `batchHeaderHash`. The node will sign the `batchHeaderHash` and return the signature.

```
func (n *Node) ProcessBatch(ctx context.Context, header *core.BatchHeader, blobs []*core.BlobMessage, rawBlobs []*node.Blob)
    ↪ (*core.Signature, error) {

    // ... snipped

    batchHeaderHash, err := header.GetBatchHeaderHash()
    if err != nil {
        return nil, err
    }

    // ... snipped

    err = n.ValidateBatch(ctx, header, blobs) // @audit does not validate `header.BatchRoot` matches `blobs' merkle root

    // ... snipped

    sig := n.KeyPair.SignMessage(batchHeaderHash)

    // ... snipped

    return sig, nil
}
```

The malicious user may combine the signatures of multiple nodes over a `batchHeaderHash` without storing the correct blobs. From the signatures the user may create a valid certificate.

Recommendations

Nodes should reconstruct the merkle tree from the received blob headers and ensure `header.BatchRoot` matches the root of the blob merkle tree.

Resolution

Validation of the `header.BatchRoot` matching the root of the blob merkle tree was added in the function `ValidateBatchHeaderRoot()` in PR #312.

EDA-04	Unsafe Downcasting Of Integers		
Asset	/*		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Unsafe downcasting instances were identified, which could lead to unexpected execution results.

node/grpc/server.go

```
func (s *Server) GetBlobHeader(ctx context.Context, in *pb.GetBlobHeaderRequest) (*pb.GetBlobHeaderReply, error) {
    var batchHeaderHash [32]byte
    copy(batchHeaderHash[:], in.GetBatchHeaderHash())

    blobHeader, protoBlobHeader, err := s.getBlobHeader(ctx, batchHeaderHash, int(in.BlobIndex), uint8(in.GetQuorumId()))  //
        ↪  @audit unsafe cast uint32 -> int and uint32 -> uint8
    if err != nil {
        return nil, err
    }

    blobHeaderHash, err := blobHeader.GetBlobHeaderHash()
    if err != nil {
        return nil, err
    }

    tree, err := s.rebuildMerkleTree(batchHeaderHash, uint8(in.GetQuorumId())) // @audit unsafe cast uint32 -> uint8
    if err != nil {
        return nil, err
    }

    proof, err := tree.GenerateProof(blobHeaderHash[:], 0)
    if err != nil {
        return nil, err
    }

    return &pb.GetBlobHeaderReply{
        BlobHeader: protoBlobHeader,
        Proof: &pb.MerkleProof{
            Hashes: proof.Hashes,
            Index:  uint32(proof.Index), // @audit unsafe cast uint64 -> uint32
        },
    }, nil
}
```

```
func (s *Server) RetrieveChunks(ctx context.Context, in *pb.RetrieveChunksRequest) (*pb.RetrieveChunksReply, error) {
    timer := prometheus.NewTimer(prometheus.ObserverFunc(func(sec float64) {
        s.node.Metrics.ObserveLatency("RetrieveChunks", "total", sec*1000) // make milliseconds
    }))
    defer timer.ObserveDuration()

    if in.GetQuorumId() > core.MaxQuorumID {
        return nil, fmt.Errorf("invalid request: quorum ID must be in range [0, %d], but found %d", core.MaxQuorumID,
            ↪ in.GetQuorumId())
    }

    var batchHeaderHash [32]byte
    copy(batchHeaderHash[:], in.GetBatchHeaderHash())

    blobHeader, _, err := s.getBlobHeader(ctx, batchHeaderHash, int(in.BlobIndex), uint8(in.GetQuorumId())) // @audit unsafe cast
        ↪ uint32 -> int (note in.GetQuorumId() has been validated < 254)
    if err != nil {
        return nil, err
    }

    // ...
```

node/grpc/utils.go

In `GetBlobHeaderFromProto()` and number of unsafe downcasts occur which may truncate integers.

```
for i, header := range h.GetQuorumHeaders() {
  quorumHeaders[i] = &core.BlobQuorumInfo{
    SecurityParam: core.SecurityParam{
      QuorumID:           core.QuorumID(header.GetQuorumId()), // @audit unsafe downcast
      AdversaryThreshold: uint8(header.GetAdversaryThreshold()), // @audit unsafe downcast
      QuorumThreshold:    uint8(header.GetQuorumThreshold()), // @audit unsafe downcast
      QuorumRate:         header.GetRatelimit(),
    },
    ChunkLength: uint(header.GetChunkLength()),
  }
}
```

Similarly on line [40], the downcast of `uint32 quorumID` value in `bundles[uint8(quorumID)]` could result in an invalid reference and an existing bundle being overwritten:

```
for i, chunks := range blob.GetBundles() {
    quorumID := blob.GetHeader().GetQuorumHeaders()[i].QuorumId
    bundles[uint8(quorumID)] = make([]*encoding.Frame, len(chunks.GetChunks())) // @audit downcasting issue, could potentially
        ↪ overwrite existing bundles
    for j, data := range chunks.GetChunks() {
    chunk, err := new(encoding.Frame).Deserialize(data)
    if err != nil {
        return nil, err
    }
    bundles[uint8(quorumID)][j] = chunk // @audit as above - downcasting
}
```

disperser/apiserver/server.go

In `disperser/apiserver/server.go` line [633] downcast of `AdversaryThreshold` and `QuorumThreshold` may also yield unexpected results:

```
for i, param := range req.GetSecurityParams() {
    params[i] = &core.SecurityParam{
        QuorumID:           core.QuorumID(param.QuorumId),
        AdversaryThreshold: uint8(param.AdversaryThreshold), // @audit downcast issues
        QuorumThreshold:    uint8(param.QuorumThreshold),    // @audit downcast issues
    }
}
```

encoding/kzg/rs/utils.go

In the `GetLeadingCosetIndex()` function on line [**78**] there is are two downcasts from `uint64` arguments to `uint32`.

```
func GetLeadingCosetIndex(i uint64, numChunks uint64) (uint32, error) {

  if i < numChunks {
    j := rb.ReverseBitsLimited(uint32(numChunks), uint32(i)) //@audit unsafe downcasts
    return j, nil
  } else {
    return 0, errors.New("cannot create number of frame higher than possible")
  }
}
```

Any `uint64` values larger than 2^{32} will be truncated.

encoding/params.go

If the integer is a signed integer or larger than 64 bits there will be unsafe casting in `ParamsFromMins`.

```
func ParamsFromMins[T constraints.Integer](minChunkLength, minNumChunks T) EncodingParams {
    return EncodingParams{
        NumChunks:   NextPowerOf2(uint64(minNumChunks)), // @audit unsafe cast
        ChunkLength: NextPowerOf2(uint64(minChunkLength)), // @audit unsafe cast
    }
}
```

encoding/rs/encode.go

```
func (g *Encoder) MakeFrames(
    polyEvals []fr.Element,
) ([]Frame, []uint32, error) {
    // reverse dataFr making easier to sample points
    err := rb.ReverseBitOrderFr(polyEvals)
    if err != nil {
        return nil, nil, err
    }
    k := uint64(0)

    indices := make([]uint32, 0)
    frames := make([]Frame, g.NumChunks)

    for i := uint64(0); i < uint64(g.NumChunks); i++ { // @audit unnecessary cast uint64 -> uint64

        // finds out which coset leader i-th node is having
        j := rb.ReverseBitsLimited(uint32(g.NumChunks), uint32(i)) // @audit unsafe cast uint64 -> uint32

        // mutltiprover return proof in butterfly order
        frame := Frame{}
        indices = append(indices, j)

        ys := polyEvals[g.ChunkLength*i : g.ChunkLength*(i+1)]
        err := rb.ReverseBitOrderFr(ys)
        if err != nil {
            return nil, nil, err
        }
        coeffs, err := g.GetInterpolationPolyCoeff(ys, uint32(j)) // @audit unnecessary cast uint32 -> uint32
```

core/eth/state.go

```
func (cs *ChainState) GetOperatorStateByOperator(ctx context.Context, blockNumber uint, operator core.OperatorID)
    ↪ (*core.OperatorState, error) {
    operatorsByQuorum, quorumIds, err := cs.Tx.GetOperatorStakes(ctx, operator, uint32(blockNumber)) // @audit unsafe downcast uint
        ↪ -> uint32
    if err != nil {
        return nil, err
    }

    return getOperatorState(operatorsByQuorum, quorumIds, uint32(blockNumber)) // @audit unsafe downcast uint -> uint32

}

func (cs *ChainState) GetOperatorState(ctx context.Context, blockNumber uint, quorums []core.QuorumID) (*core.OperatorState, error)
    ↪ {
    operatorsByQuorum, err := cs.Tx.GetOperatorStakesForQuorums(ctx, quorums, uint32(blockNumber)) // @audit unsafe downcast uint
        ↪ -> uint32
    if err != nil {
        return nil, err
    }

    return getOperatorState(operatorsByQuorum, quorums, uint32(blockNumber)) // @audit unsafe downcast uint -> uint32
}
```

core/eth/tx.go

Within the function `RegisterOperatorWithChurn()` there is an unsafe cast from `uint32` to `uint8`.

```
  operatorsToChurn := make([]regcoordinator.IRegistryCoordinatorOperatorKickParam, len(churnReply.OperatorsToChurn))
  for i := range churnReply.OperatorsToChurn {
    operatorsToChurn[i] = regcoordinator.IRegistryCoordinatorOperatorKickParam{
      QuorumNumber: uint8(churnReply.OperatorsToChurn[i].QuorumId), // @audit unsafe cast uint32 -> uint8
      Operator:     gethcommon.BytesToAddress(churnReply.OperatorsToChurn[i].Operator),
    }
}
```

core/serialization.go

An unsafe downcast exists in `GetQuorumBlobParamsHash()` as `q.ChunkLength` is a `uint` which will be 64 bits if the OS architecture is 64 bits. It will downcast to a `uint32`.

```
func (h *BlobHeader) GetQuorumBlobParamsHash() ([32]byte, error) {

    // ... snipped

  type quorumBlobParams struct {
    QuorumNumber                 uint8
    AdversaryThresholdPercentage uint8
    QuorumThresholdPercentage    uint8
    ChunkLength                  uint32
  }

  qbp := make([]quorumBlobParams, len(h.QuorumInfos))
  for i, q := range h.QuorumInfos {
    qbp[i] = quorumBlobParams{
      QuorumNumber:                 uint8(q.QuorumID), // @audit unnecessary cast uint8 -> uint8
      AdversaryThresholdPercentage: uint8(q.AdversaryThreshold), // @audit unnecessary cast uint8 -> uint8
      QuorumThresholdPercentage:    uint8(q.QuorumThreshold), // @audit unnecessary cast uint8 -> uint8
      ChunkLength:                  uint32(q.ChunkLength), // @audit unsafe cast uint -> uint32
    }
  }
```

```
func (h *BlobHeader) Encode() ([]byte, error) {

    // ... snipped

    qbp := make([]quorumBlobParams, len(h.QuorumInfos))
    for i, q := range h.QuorumInfos {
        qbp[i] = quorumBlobParams{
            QuorumNumber:                 uint8(q.QuorumID), // @audit unnecessary cast uint8 -> uint8
            AdversaryThresholdPercentage: uint8(q.AdversaryThreshold), // @audit unnecessary cast uint8 -> uint8
            QuorumThresholdPercentage:    uint8(q.QuorumThreshold), // @audit unnecessary cast uint8 -> uint8
            ChunkLength:                  uint32(q.ChunkLength), // @audit uint -> uint32
        }
    }
    slices.SortStableFunc[[]quorumBlobParams](qbp, func(a, b quorumBlobParams) int {
        return int(a.QuorumNumber) - int(b.QuorumNumber)
    })

    s := struct {
        Commitment       commitment
        DataLength       uint32
        QuorumBlobParams []quorumBlobParams
    }{
        Commitment: commitment{
            X: h.Commitment.X.BigInt(new(big.Int)),
            Y: h.Commitment.Y.BigInt(new(big.Int)),
        },
        DataLength:       uint32(h.Length), // @audit uint -> uint32
        QuorumBlobParams: qbp,
    }
```

core/aggregation.go

```
func GetSignedPercentage(state *OperatorState, quorum QuorumID, stakeAmount *big.Int) uint8 {

  stakeAmount = stakeAmount.Mul(stakeAmount, new(big.Int).SetUint64(percentMultiplier))
  quorumThresholdBig := stakeAmount.Div(stakeAmount, state.Totals[quorum].Stake)

  quorumThreshold := uint8(quorumThresholdBig.Uint64()) // @audit unsafe cast Big -> uint64 and uint64 -> uint8

  return quorumThreshold
}
```

common/geth/client.go

```
func (c *EthClient) GetCurrentBlockNumber(ctx context.Context) (uint32, error) {
    bn, err := c.Client.BlockNumber(ctx)
    return uint32(bn), err // @audit uint64 -> uint32
}
```

Recommendations

Implement checks before downcasting values to ensure they are within an expected range and will not truncate the value.

Resolution

The Layr Lab team checked all noted downcasts and addressed those which could cause issues. Unfixed castings were deemed to be non-exploitable due to existing restrictions on inputs. Relevant fixes can be found in PR #413.

EDA-05	Length Proofs Always Allows Appending Zero Field Elements To Change Length		
Asset	/*		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Field elements of value zero can be appended to the end of blobs to increase the length without changing the commitment, length commitment or length proof.

Taking the zero polynomial as an example it can be shown that appending more zeros to the polynomial will not change the commitment. This attack can be generalised for all polynomials by adding zero field elements to the end of a polynomial.

Zero polynomials will always evaluate to the point at infinity. Due to the pairing always passing when each side contains a point at infinity, it is possible to set any value for the length in a length proof.

The length proof will evaluate the polynomial on G2 with co-effecients shifted by s^{t-l} as follows.

$s^{t-l}, s^{t-l+1}, ..., s^{t}$

It uses the extended powers of tau in G2 such that the highest co-efficient s^t is larger than those available on G1.

The length commitment will be the evaluation of the polynomial on G1 without shifting co-efficients as follows.

$s^0, s^1, ..., s^l$.

However, since the polynomial is the zero polynomial all evaluations will be zero (or the point at infinity). Therefore, the shift does not modify the evaluation as it would for non-zero co-efficients.

Validating the length proof consists of checking the pairing $e(Comm, [s^{t-l}]) == e(G_1, Proof)$. The zero polynomial will have the same evaluations for $Comm$ and $Proof$ irrelevant of the length. Each evaluation is the point at infinity.

Therefore, the length pairing check will contain infinity on both sides of the equation and pass for any value of l.

$e([s^{t-l}], 0) == e(G_1, 0)$

Similarly, the equivalence pairing will also have infinity on both sides and therefore pass.

$e(0, G_2) == (G_1, 0)$

The impact is a malicious user can shrink or extend the blob with zero co-efficients if it is the zero polynomial.

Recommendations

One remediation is to disallow zero as the last co-efficient of a polynomial. This will ensure prevent appending zeros to the end of a polynomial and increasing the length will change the length commitment and proof.

Resolution

The Layr Labs team gave the following comment on the issue:

> *"The disperser has been a trusted entity. [...] With that assumption, if the user's send us zero's as data, the disperser will treat those zeros as data, and code them accordingly and the users will pay the number of zeros they send."*

Due to the fact an attack of this type would not affect liveness of data availability and the complexity of resolving the issue, the Layr Labs team has decided to not resolve this issue at the current time.

EDA-06	Index Out Of Bounds Panics		
Asset	`core/aggregation.go, core/bn254/attestation.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There are some potential index out of bounds panics that are reachable from input received over the net.

core/bn254/attestation.go

The two functions `DeserializeG1()` and `DeserializeG2()` are both vulnerable to index out of bounds panics if insufficient data is passed. `DeserializeG1()` is called on reply data for calls to `StoreChunks()`. If the disperser replies with invalid signature bytes it will cause an index out of bounds panic.

```
func DeserializeG1(b []byte) *bn254.G1Affine {
    p := new(bn254.G1Affine)
    p.X.SetBytes(b[0:32]) // @audit index out of bounds panic if len(b) < 64
    p.Y.SetBytes(b[32:64])
    return p
}

// ... snipped

func DeserializeG2(b []byte) *bn254.G2Affine {
    p := new(bn254.G2Affine)
    p.X.A0.SetBytes(b[0:32]) // @audit index out of bounds panic if len(b) < 128
    p.X.A1.SetBytes(b[32:64])
    p.Y.A0.SetBytes(b[64:96])
    p.Y.A1.SetBytes(b[96:128])
    return p
}
```

In `disperser/batcher/grpc/dispatcher.go::sendChunks()` the deserialisation of G1 points is called on untrusted reply.

```
reply, err := gc.StoreChunks(ctx, request, opt)

if err != nil {
    return nil, err
}

sigBytes := reply.GetSignature()
sig := &core.Signature{G1Point: new(core.Signature).Deserialize(sigBytes)} // @audit calls DeserializeG1()
```

core/aggregation.go

If the function `AggregateSignatures()` is called with `len(quorumIDs) = 0` an index out of bounds panic will occur.

The bug occurs since `aggSigs` and `aggPubKeys` will have length zero if `quorumIDs` has length zero. The return value attempts to index these arrays at index zero which causes an index out of bounds panic when the length is zero.

```
func (a *StdSignatureAggregator) AggregateSignatures(ctx context.Context, state *IndexedOperatorState, quorumIDs []QuorumID,
    ↪  message [32]byte, messageChan chan SignerMessage) (*SignatureAggregation, error) {

  // Ensure all quorums are found in state
  for _, id := range quorumIDs {
    _, found := state.Operators[id]
    if !found {
      return nil, errors.New("quorum not found")
    }
  }

  stakeSigned := make([]*big.Int, len(quorumIDs))
  for ind := range quorumIDs {
    stakeSigned[ind] = big.NewInt(0)
  }
  aggSigs := make([]*Signature, len(quorumIDs))
  aggPubKeys := make([]*G2Point, len(quorumIDs))

  signerMap := make(map[OperatorID]bool)

  // Aggregate Signatures
  numOperators := len(state.IndexedOperators)

  for numReply := 0; numReply < numOperators; numReply++ {

    // ... snipped

  // Aggregate the aggregated signatures. We reuse the first aggregated signature as the accumulator
  for i := 1; i < len(aggSigs); i++ {
    aggSigs[0].Add(aggSigs[i].G1Point)
  }

  // Aggregate the aggregated public keys. We reuse the first aggregated public key as the accumulator
  for i := 1; i < len(aggPubKeys); i++ {
    aggPubKeys[0].Add(aggPubKeys[i])
  }

    // ... snipped

  return &SignatureAggregation{
    NonSigners:       nonSignerKeys,
    QuorumAggPubKeys: quorumAggPubKeys,
    AggPubKey:        aggPubKeys[0], // @audit index out of bounds if length is zero
    AggSignature:     aggSigs[0],    // @audit index out of bounds if length is zero
    QuorumResults:    quorumResults,
  }, nil

}
```

encoding/kzg/verifier/batch_commit_equivalence.go

An index out of bounds panic will occur if the length of both `g1commits` and `g2commits` is zero. This may arise when `Node.ProcessBatch()` is called with empty `blobs` and `rawBlobs`.

```
func CreateRandomnessVector(g1commits []bn254.G1Affine, g2commits []bn254.G2Affine) ([]fr.Element, error) {
  r, err := GenRandomFactorForEquivalence(g1commits, g2commits)
  if err != nil {
    return nil, err
  }
  n := len(g1commits)

  if len(g1commits) != len(g2commits) {
    return nil, errors.New("inconsistent number of blobs for g1 and g2")
  }

  randomsFr := make([]fr.Element, n)

  randomsFr[0].Set(&r) // @audit index out of bounds if zero commits

  // power of r
  for j := 0; j < n-1; j++ {
    randomsFr[j+1].Mul(&randomsFr[j], &r)
  }

  return randomsFr, nil
}
```

encoding/kzg/verifier/multiframe.go

```
func GenRandomnessVector(samples []Sample) ([]fr.Element, error) {
  // root randomness
  r, err := GenRandomFactor(samples)
  if err != nil {
    return nil, err
  }

  n := len(samples)

  randomsFr := make([]fr.Element, n)
  randomsFr[0].Set(&r) // @audit index out of bounds if len(samples) = 0

}
```

Recommendations

The issue may be remediated as follows.

- `core/bn254/attestation.go` by performing length checks and returning an error if insufficient bytes are received.
- `core/aggregation.go` by returning an error if `len(quorumIDs) = 0`.
- `encoding/kzg/verifier/batch_commit_equivalence.go` by returning an error if `len(g1commits) = 0`. additionally `Node.ProcessBatch()` should return an error if no blobs are submitted.
- `encoding/kzg/verifier/multiframe.go` by returning an error if `len(samples) = 0`.

Resolution

Fixes were made in PR #417.

EDA-07	Insufficient Arithmetic Checks		
Asset	/*		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A number of arithmetic operations occur on unvalidated or partially validated input which leads to overflows, division by zero and memory exhaustion issues.

encoding/kzg/prover/prover.go

A malicious user may cause an overflow by setting large values of `params.ChunkLength` and `params.NumChunks` such that the multiplication overflows. This allows by passing the blob length checks. Note that this data is partially validated in calls to `EncodeBlob()` which takes each of `ChunkLength` and `NumChunks` as `uint32` then casts them to a `uint64`. However, it is not always validated in other call paths.

```
func (g *Prover) newProver(params encoding.EncodingParams) (*ParametrizedProver, error) {

  // Check that the parameters are valid with respect to the SRS.
  if params.ChunkLength*params.NumChunks >= g.SRSOrder { // @audit multiplication overflow allows passing check
    return nil, fmt.Errorf("the supplied encoding parameters are not valid with respect to the SRS. ChunkLength: %d, NumChunks: %d,
          ↪  SRSOrder: %d", params.ChunkLength, params.NumChunks, g.SRSOrder)
  }
```

encoding/kzg/prover/precompute.go

Overflows and an out of memory panic may occur if a large `l` value is passed to `Precompute()`.

```
// m = len(poly) - 1, which is deg
func (p *SRSTable) Precompute(dim, dimE, l, m uint64, filePath string, numWorker uint64) [][]bn254.G1Affine {
    order := dimE * l // @audit multiplication overflow
    if l == 1 {
        order = dimE * 2 // @audit multiplication overflow
    }
    // TODO, create function only read g1 points
    //s1 := ReadG1Points(p.SrsFilePath, order)
    n := uint8(math.Log2(float64(order)))
    fs := fft.NewFFTSettings(n)

    fftPoints := make([][]bn254.G1Affine, l) // @audit out of memory if large l
```

encoding/fft/fft_fr.go

Fast Fourier Transform will cause a division by zero if the number of elements passed is zero. Noting that polynomials are padded with zero co-efficients before the evaluations are calculated, thus it is not expected to be called with zero `vals`.

```
func (fs *FFTSettings) InplaceFFT(vals []fr.Element, out []fr.Element, inv bool) error {
    n := uint64(len(vals))
    if n > fs.MaxWidth {
        return fmt.Errorf("got %d values but only have %d roots of unity", n, fs.MaxWidth)
    }
    if !IsPowerOfTwo(n) {
        return fmt.Errorf("got %d values but not a power of two", n)
    }
    if inv {
        var invLen fr.Element

        invLen.SetInt64(int64(n))

        invLen.Inverse(&invLen)
        rootz := fs.ReverseRootsOfUnity[:fs.MaxWidth]
        stride := fs.MaxWidth / n // @audit division by zero

        fs._fft(vals, 0, 1, rootz, stride, out)
        var tmp fr.Element
        for i := 0; i < len(out); i++ {
            tmp.Mul(&out[i], &invLen)
            out[i].Set(&tmp)
        }
        return nil
    } else {
        rootz := fs.ExpandedRootsOfUnity[:fs.MaxWidth]
        stride := fs.MaxWidth / n // @audit division by zero
        // Regular FFT
        fs._fft(vals, 0, 1, rootz, stride, out)
        return nil
    }
}
```

core/data.go

Multiple overflows can occur in `Validate()` for a `BlobRequestHeader` when `quorum.AdversaryThreshold+10` overflows a `uint8`. Furthermore, it is possible to cause a number of overflows in the function `EncodedSizeALlQuorums()`.

```
func (h *BlobRequestHeader) Validate() error {
    for _, quorum := range h.SecurityParams {
        if quorum.QuorumThreshold < quorum.AdversaryThreshold+10 { // @audit addition overflow
            return errors.New("invalid request: quorum threshold must be >= 10 + adversary threshold")
        }
        if quorum.QuorumThreshold > 100 {
            return errors.New("invalid request: quorum threshold exceeds 100")
        }
        if quorum.AdversaryThreshold == 0 {
            return errors.New("invalid request: adversary threshold equals 0")
        }
    }
    return nil
}

// ... snipped

// Returns the total encoded size in bytes of the blob across all quorums.
func (b *BlobHeader) EncodedSizeAllQuorums() int64 {
  size := int64(0)
  for _, quorum := range b.QuorumInfos {

    size += int64(roundUpDivide(b.Length*percentMultiplier*encoding.BYTES_PER_COEFFICIENT,
        ↪  uint(quorum.QuorumThreshold-quorum.AdversaryThreshold))) // @audit multiplication, addition and subtraction overflows
  }
  return size
}
```

core/assignment.go

The following functions are called before `ValidateChunkLength()` and may cause panics.

```
    func roundUpDivideBig(a, b *big.Int) *big.Int {

  one := new(big.Int).SetUint64(1)
  num := new(big.Int).Sub(new(big.Int).Add(a, b), one) // a + b - 1        @audit subtraction overflow
  res := new(big.Int).Div(num, b)                      // (a + b - 1) / b  @audit division by zero
  return res

}

func roundUpDivide(a, b uint) uint {
  return (a + b - 1) / b // @audit division by zero

}
```

encoding/params.go

Overflows and division by zero issues may occur in `params.go`.

```
func (p EncodingParams) ChunkDegree() uint64 {
    return p.ChunkLength - 1 // @audit subtraction overflow
}

func (p EncodingParams) NumEvaluations() uint64 {
    return p.NumChunks * p.ChunkLength // @audit multiplication overflow
}

// ... snipped

func ParamsFromMins[T constraints.Integer](minChunkLength, minNumChunks T) EncodingParams {
    return EncodingParams{
        NumChunks:   NextPowerOf2(uint64(minNumChunks)), // @audit unsafe cast
        ChunkLength: NextPowerOf2(uint64(minChunkLength)), // @audit unsafe cast
    }
}

func ParamsFromSysPar(numSys, numPar, dataSize uint64) EncodingParams {

    numNodes := numSys + numPar // @audit addition overflow
    dataLen := roundUpDivide(dataSize, BYTES_PER_COEFFICIENT)
    chunkLen := roundUpDivide(dataLen, numSys)
    return ParamsFromMins(chunkLen, numNodes)

}

func GetNumSys(dataSize uint64, chunkLen uint64) uint64 {
    dataLen := roundUpDivide(dataSize, BYTES_PER_COEFFICIENT)
    numSys := dataLen / chunkLen // @audit division by zero
    return numSys
}
```

```
// ValidateEncodingParams takes in the encoding parameters and returns an error if they are invalid.
func ValidateEncodingParams(params EncodingParams, blobLength, SRSOrder int) error {

    if int(params.ChunkLength*params.NumChunks) >= SRSOrder { // @audit multiplication overflow
        return fmt.Errorf("the supplied encoding parameters are not valid with respect to the SRS. ChunkLength: %d, NumChunks: %d,
            ↪  SRSOrder: %d", params.ChunkLength, params.NumChunks, SRSOrder)
    }

    if int(params.ChunkLength*params.NumChunks) < blobLength { // @audit as above
        return fmt.Errorf("the supplied encoding parameters are not sufficient for the size of the data input")
    }

    return nil

}
```

encoding/utils.go

Listed below are a number of potential overflows and divisions by zero. However, many of these functions operate on partially validated input and cannot be exploited.

```
// GetBlobLength converts from blob size in bytes to blob size in symbols
func GetBlobLength(blobSize uint) uint {
  symSize := uint(BYTES_PER_COEFFICIENT)
  return (blobSize + symSize - 1) / symSize // @audit addition and subtraction overflow
}

// GetBlobSize converts from blob length in symbols to blob size in bytes. This is not an exact conversion.
func GetBlobSize(blobLength uint) uint {
  return blobLength * BYTES_PER_COEFFICIENT // @audit multiplication overflow
}

// GetBlobLength converts from blob size in bytes to blob size in symbols
func GetEncodedBlobLength(blobLength uint, quorumThreshold, advThreshold uint8) uint {
  return roundUpDivide(blobLength*100, uint(quorumThreshold-advThreshold)) // @audit multiplication and subtraction overflow
}

func NextPowerOf2(d uint64) uint64 {
  nextPower := math.Ceil(math.Log2(float64(d)))
  return uint64(math.Pow(2.0, nextPower)) // @audit potential overflow
}

func roundUpDivide[T constraints.Integer](a, b T) T {
  return (a + b - 1) / b // @audit division by zero, addition and subtraction overflow

}
```

encoding/kzg/rs/utils.go

```
// ToByteArray converts a list of Fr to a byte array
func ToByteArray(dataFr []fr.Element, maxDataSize uint64) []byte {
    n := len(dataFr)
    dataSize := int(math.Min(
        float64(n*encoding.BYTES_PER_COEFFICIENT), // @audit multiplication overflow feasible if uint `n` is 32 bits
        float64(maxDataSize),
    ))
    data := make([]byte, dataSize)
    for i := 0; i < n; i++ {
        v := dataFr[i].Bytes()

        start := i * encoding.BYTES_PER_COEFFICIENT
        end := (i + 1) * encoding.BYTES_PER_COEFFICIENT

        if uint64(end) > maxDataSize {
            copy(data[start:maxDataSize], v[1:])
            break
        } else {
            copy(data[start:end], v[1:])
        }
    }

    return data
}

func GetNumElement(dataLen uint64, CS int) uint64 {
    numEle := int(math.Ceil(float64(dataLen) / float64(CS))) // @audit division by zero
    return uint64(numEle)
}

// helper function
func RoundUpDivision(a, b uint64) uint64 {
    return uint64(math.Ceil(float64(a) / float64(b))) // @audit division by zero
}

func NextPowerOf2(d uint64) uint64 {
    nextPower := math.Ceil(math.Log2(float64(d)))
    return uint64(math.Pow(2.0, nextPower)) // @audit potential overflow
}
```

encoding/kzg/verifier/verifier.go

The verifier does not check if `Length` field submitted in `encoding.BlobCommitments` is larger than `Verifier.SRSOrder`.

```
172 func (v *Verifier) VerifyCommit(lengthCommit *bn254.G2Affine, lowDegreeProof *bn254.G2Affine, length uint64) error {
    g1Challenge, err := kzg.ReadG1Point(v.SRSOrder-length, v.KzgConfig) // @audit underflow could happen here
174 if err != nil {
        return err
176 }
    err = VerifyLowDegreeProof(lengthCommit, lowDegreeProof, &g1Challenge)
178 if err != nil {
        return fmt.Errorf("%v . %v ", "low degree proof fails", err)
180 } else {
        return nil
182 }
}
```

On line [174] in `encoding/kzg/verifier/verifier.go` such a case would lead to an underflow and passing the check on line [42] in `encoding/kzg/pointsIO`. This allows the attacker to get a valid `g1Challenge` for a higher degree commitment.

```
41 func ReadG1Point(n uint64, g *KzgConfig) (bn254.G1Affine, error) {
    if n > g.SRSOrder { // @audit underflow allows passing the check
43      return bn254.G1Affine{}, fmt.Errorf("requested power %v is larger than SRSOrder %v", n, g.SRSOrder)
    }
45  g1point, err := ReadG1PointSection(g.G1Path, n, n+1, 1)
    if err != nil {
47      return bn254.G1Affine{}, err
    }
49  return g1point[0], nil
}
```

The result will be an out of memory bug in `ReadG1PointSection()` which will attempt to allocate `n*G1PointBytes` bytes.

```
func ReadG1PointSection(filepath string, from, to uint64, numWorker uint64) ([]bn254.G1Affine, error) {
    if to <= from {
        return nil, fmt.Errorf("The range to read is invalid, from: %v, to: %v", from, to)
    }
    g1f, err := os.Open(filepath)
    if err != nil {
        log.Println("ReadG1PointSection.ERR.o", err)
        return nil, err
    }

    defer func() {
        if err := g1f.Close(); err != nil {
            log.Printf("g1 close error %v\n", err)
        }
    }()

    n := to - from

    g1r := bufio.NewReaderSize(g1f, int(n*G1PointBytes)) // @audit either negative or large memory allocation causing Out of Memory
```

Recommendations

It is recommended to perform overflow checks on arithmetic operations which stem from user input. Update each of the functions listed above to return an error if an arithmetic overflow or division by zero will occur.

Resolution

The Layr Labs team plan to address this issue in a future release.

EDA-08	Lack of Transport Layer Encryption		
Asset	`server.go of Churner, Disperser, Node, Retriever`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Servers of Churner, Disperser, Node and Retrieve do not utilise any transport layer encryption such as TLS, exposing sensitive data transmitted between clients and the server to interception and manipulation. Without TLS, communications occur over plaintext protocols, leaving them susceptible to eavesdropping and man-in-the-middle attacks.

Note, it was observed that Node and Disperser have an optional flag to use secure GRPC with `disperseBlob()` and `requestChurnApproval()`. The optional configuration is disabled by default.

Without authentication, it would be possible for anyone to call the `Node.StoreChunks()` method. Each Node would then start storing chunks for arbitrary users without receiving funding. The Node endpoint is susceptible to DoS attacks and free storage without authentication as operators will not receive funding for the storage requests if the certificate is not posted on-chain.

Furthermore, the remaining endpoints perform a range of CPU intensive cryptographic tasks, caching both in memory and disk and network consumption.

Recommendations

Implement TLS encryption across all communication channels between clients and servers. This involves configuring the server to support HTTPS (HTTP over TLS) for web traffic and enforcing TLS for all other protocols.

Resolution

The team noted:

"We understand the requirement of TLS and it's uses. Regarding the finding, there are multiple controls inplace including the nodes whitelisting the disperser address. Node's will not accept requests from random address, so those requests will be dropped. The chunk request received goes through validation of the structure and the message data itself goes through the validateStoreChunkRequest function which checks the validity of the data and any changes to it will almost always result in rejection of the request. With these protections in-place not having TLS does not seem a major problem but we do agree that there are unlikely scenario's of DoS possible."

With this in mind, the issue has been closed as an accepted risk.

EDA-09	Additional Checks Required		
Asset	`/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Additional checks should be implemented to avoid unexpected results or unhandled errors in the following:

- check `polyEvals` length prior to being used in `ys := polyEvals[g.ChunkLength*i : g.ChunkLength*(i+1)]` on line [**89**] in `encoding/rs/encode.go`
- check `start` is also less than `len(leves)` on line [**251**] in `encoding/fft/zero_poly.go`
- `readG1Worker()` and `readG2Worker()` requires input sanitisation in `encoding/kzg/pointsIO.go`
- `readWorker()` requires input sanitisation on line [**283**] in `encoding/kzg/prover/precompute.go`
- check `if len(hh) != 0` on line [**51**] in `indexer/header.go`
- check `if len(opStates[i]) != 0` on line [**295**] in `core/eth/tx.go`
- check `len(t.V) != 0` on line [**101**] in `encoding/utils/toeplitz/toeplitz.go`
- check `if len(t.V)+1 >= numRow * 2` on line [**109**] in `encoding/utils/toeplitz/toeplitz.go`
- check for minimum size required of `g.G2Trailing` line [**83**] in `encoding/kzg/prover/parametrized_prover.go`
- check if `m-(j+i*l)` is within bounds of `polyFr` on line [**253**] in `encoding/kzg/prover/parametrized_prover.go`

Recommendations

Implement additional checks where recommended and as required.

Resolution

The Layr Labs team plan to address this issue in a future release.

EDA-10	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. **Variable Overriding**

   Related Asset(s): node/grpc/utils.go

   The variable `i` is overwritten in consecutive loops.

```
func GetBlobMessages(in *pb.StoreChunksRequest) ([]*core.BlobMessage, error) {
  blobs := make([]*core.BlobMessage, len(in.GetBlobs()))
  for i, blob := range in.GetBlobs() { // @audit first i

    blobHeader, err := GetBlobHeaderFromProto(blob.GetHeader())
    if err != nil {
      return nil, err
    }

    bundles := make(map[core.QuorumID]core.Bundle, len(blob.GetBundles()))
    for i, chunks := range blob.GetBundles() { // @audit second i
      quorumID := blob.GetHeader().GetQuorumHeaders()[i].QuorumId
      bundles[uint8(quorumID)] = make([]*encoding.Frame, len(chunks.GetChunks()))
      for j, data := range chunks.GetChunks() {
        chunk, err := new(encoding.Frame).Deserialize(data)
        if err != nil {
          return nil, err
        }
        bundles[uint8(quorumID)][j] = chunk
      }
    }

    blobs[i] = &core.BlobMessage{
      BlobHeader: blobHeader,
      Bundles:    bundles,
    }
  }
  return blobs, nil
}
```

   Rename the variables in subsequent loops to avoid overriding.

2. **Unused Error Returns**

   Related Asset(s): encoding/kzg/srs.go, encoding/kzg/kzg.go

   Several functions do not make use of the error field they return and instead always return `nil`. This is unnecessary and can therefore be removed.

   Examples of this behaviour are:

   - `NewKZGSetting()` in `encoding/kzg/kzg.go`,
   - `NewSrs()` in `encoding/kzg/srs.go`

   Make use of errors returned by the documented functions or remove the returned error fields.

3. **Unnecessary Casting**

   Related Asset(s): core/thegraph/state.go, disperser/apiserver/server.go, core/validator.go

   In the following code snippet `id` is of type `uint8` and does not need to be case to `uint8`.

   `core/thegraph/state.go`:

```
func (ics *indexedChainState) getQuorumAPKs(ctx context.Context, quorumIDs []core.QuorumID, blockNumber uint32)
    ↪ map[uint8]*quorumAPK {
    quorumAPKs := make(map[uint8]*quorumAPK)
    for i := range quorumIDs {
        id := quorumIDs[i]
        apk, err := ics.getQuorumAPK(ctx, id, blockNumber)
        if err != nil {
            quorumAPKs[id] = &quorumAPK{
                QuorumNumber:  uint8(id), // @audit unnecessary cast
                AggregatePubk: nil,
                Err:           err,
            }
            continue
        }
        if apk == nil {
            quorumAPKs[id] = &quorumAPK{
                QuorumNumber:  uint8(id), // @audit unnecessary cast
                AggregatePubk: nil,
                Err:           fmt.Errorf("quorum APK not found for quorum %d", id),
            }
            continue
        }
        quorumAPKs[id] = &quorumAPK{
            QuorumNumber:  uint8(id), // @audit unnecessary cast
            AggregatePubk: apk,
            Err:           nil,
        }
    }
    return quorumAPKs
}
```

   `disperser/apiserver/server.go` function `checkRateLimitsAndAddRates()` line [**381**]:

```
length := encoding.GetBlobLength(uint(blobSize)) // @audit unsafe cast int -> uint
encodedLength := encoding.GetEncodedBlobLength(length, uint8(param.QuorumThreshold), uint8(param.AdversaryThreshold)) //
    ↪ @audit unnecessary casts both uint8 -> uint8
```

   `core/validator.go` line [**149**]:

```
for ind := range chunks {
    samples[ind] = encoding.Sample{
        Commitment:      blob.BlobHeader.BlobCommitments.Commitment,
        Chunk:           chunks[ind],
        AssignmentIndex: uint(indices[ind]), // @audit unnecessary cast uint -> uint
        BlobIndex:       blobIndex,
    }
}
```

   It is recommended to remove the unnecessary casting.

4. **Actionable Comments**

   Related Asset(s): All files

   Some files contain `"TODO"` comments or commented out code, these should be checked and removed if unneeded or updated if the features mentioned do not currently exist.

   - line [**126**] of `encoding/kzg/verifier/multiframe.go` the commented out code should be removed or replaced with comments that explain the purpose of lines [**125-128**].
   - line [**65**] and line [**87**] of `brownie` contains `"TODO"` comments.
   - line [**80**] of `core/aggregation.go` contains a `"TODO"` comment.

- line [386] of `encoding/kzg/verifier/pointsIO.go` contains a `"TODO"` comment.
- line [77] of `disperser/dispatcher/grpc/dispatcher.go` contains a `"TODO"` comment.

Review noted areas and make alterations as seen fit.

5. **Include Capacity During Array Creation Where Possible**

   Related Asset(s): disperser/batcher/batcher.go

   When the capacity of an array will be known ahead of time it is desirable to supply the capacity parameter to `make()`. The capacity will increase performance by reducing the number of array resize operations.

```
func serializeProof(proof *merkletree.Proof) []byte {
    proofBytes := make([]byte, 0) // @audit include capacity as len(proof.Hashes) * 32
    for _, hash := range proof.Hashes {
        proofBytes = append(proofBytes, hash[:]...)
    }
    return proofBytes
}
```

6. **Invalid Regex**

   Related Asset(s): All files

   The regex in `core/serialization.go` on line [405] will accept alphabetic characters as a port number - `regex := regexp.MustCompile(`^([^:]+):([^;]+);([^;]+)$`)`.

   Modify regex to only accept numbers as port numbers.

7. **Typos**

   Related Asset(s): All files

   Typos were noticed in some file, these should be corrected for clarity:

   - line [170] of `encoding/kzg/prover/parametrized_prover.go` *"scaler mulplication"* should be *"scalar multiplication"*.
   - line [199] of `encoding/kzg/prover/parametrized_prover.go` *"buttefly"* should be *"butterfly"*.
   - line [271] of `encoding/kzg/verifier/multiframe.go` *"Acessed"* should be *"Accessed"*.
   - line [85] of `encoding/rs/encode.go` *"mutltiprover"* should be *"multiprover"*.
   - line [120] of `encoding/kzg/verifier/batch_commit_equivalence.go` *"paring"* should be *"pairing"*.

   Review noted areas and make alterations as seen fit.

Recommendations

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

Resolution

The development team has acknowledged the above findings and advised that they will be addressed at a later date.

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.

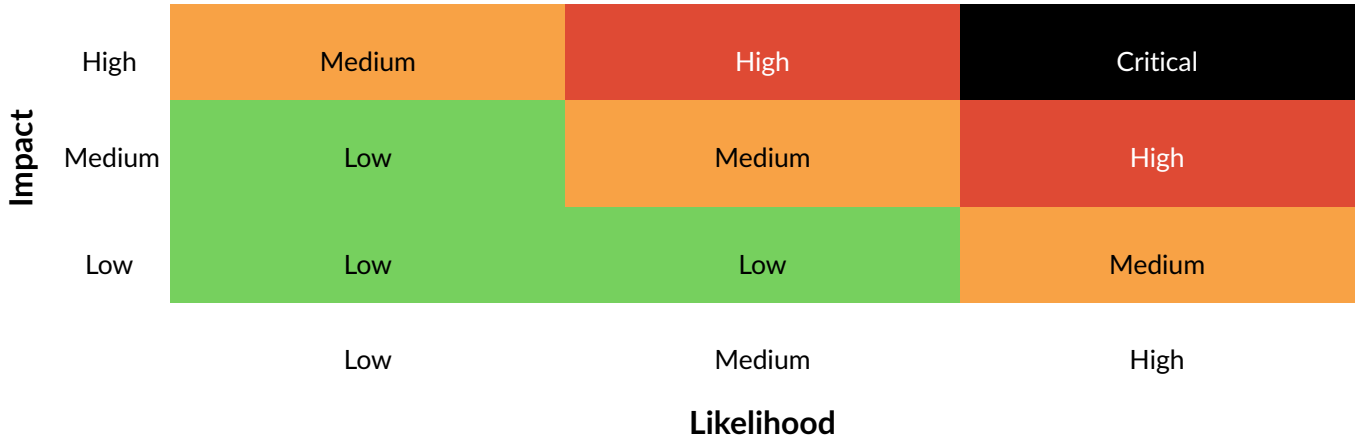

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.