Layr Labs

EigenDA Blazar

Security Assessment Report

Version: 2.0

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the EigenDA components in scope. The review focused solely on the security aspects of the implementation, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the EigenDA 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 EigenDA components in scope.

Overview

EigenDA is a Data Availability service built on top of Ethereum. Restakers using EigenLayer will be able to delegate their staked ETH to node operators working on EigenDA.

In this Blazar update, Layr Labs has focused on making EigenDA more performant, robust, and user-friendly. This has been achieved through the introduction of a new role, Relays, which store and serve blobs and encoded chunks. Layr Labs has also made changes to the codebase of the Disperser and Validator Node roles to achieve these goals.

Onchain changes for EigenDA Blazar include the introduction of a new payments system which allows users to either make use of a pay-as-you-go usage model or a reserved bandwidth model managed by the EigenDA governance body. This split enables occasional users and those with constant demand who desire predictable pricing and throughput.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Layr-Labs/eigenda and Layr-Labs/eigenlayer-middleware repositories.

The scope of this time-boxed review was strictly limited to files at commit 89b42a20 and 11b0061 for each repository respectively.

As this was an upgrade of the EigenDA system, only offchain changes since the first Sigma Prime review at commit 91838ba were in scope. For the onchain aspect the following files were in scope only:

- From the `eigenda` repository:
  - `IEigenDAStructs.sol`
  - `EigenDAServiceManager.sol`
  - `EigenDACertVerifier.sol`
  - `EigenDACertVerificationUtils.sol`
  - `EigenDAThresholdRegistry.sol`
  - `EigenDARelayRegistry.sol`
  - `EigenDADisperserRegistry.sol`
- From the `eigenlayer-middleware` repository:
  - `RegistryCoordinator.sol`
  - `EjectionManager.sol`
  - `SocketRegistry.sol`

The fixes of the identified issues were assessed at commit 6a70c33 for the `eigenda` repository.

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

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 also utilised the following automated testing tools:

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

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 also utilised the following automated testing tools:

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

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 28 issues during this assessment. Categorised by their severity:

- Critical: 5 issues.
- High: 5 issues.
- Medium: 8 issues.
- Low: 6 issues.
- Informational: 4 issues.

Detailed Findings

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

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EDA2-01	`nil` Pointer Dereference In `HashGetChunksRequest()` Function	Critical	Resolved
EDA2-02	Missing Signature Replay Protection In `GetChunks()` & `StoreChunks()` Requests	Critical	Resolved
EDA2-03	`nil` Pointer Dereference In `StoreChunks()` Due To Incorrect Validation Order	Critical	Resolved
EDA2-04	`AccountID` String Manipulation	Critical	Resolved
EDA2-05	Malicious User May Manipulate `ValidatePayment()` By Injecting Invalid Payments	Critical	Resolved
EDA2-06	Ambiguous Hashing Implementation Without Length Encoding Creates Collision Risk	High	Resolved
EDA2-07	Index Out Of Bounds Panic In `DeserializeGnark()`	High	Resolved
EDA2-08	Usage Is Not Rolled Back On Failed Requests	High	Closed
EDA2-09	`PricePerSymbol` Is Liable To Change Impacting Existing Payments	High	Resolved
EDA2-10	Commitment Validation Should Be Performed Before Metering	High	Resolved
EDA2-11	Lack Of Domain Separation In Message Hashing Functions	Medium	Resolved
EDA2-12	Multiplication Overflow In `ValidateEncodingParams()`	Medium	Resolved
EDA2-13	Incorrect Overflow Usage Check In Client	Medium	Closed
EDA2-14	Incorrect Reservation Period Validation	Medium	Resolved
EDA2-15	Changes In Reservation Period Interval Affects Current Reservations	Medium	Resolved
EDA2-16	`header.CumulativePayment` Is Modified In `ValidatePayment()`	Medium	Resolved
EDA2-17	Lossy & Unnecessary Casting	Medium	Resolved
EDA2-18	Lack Of Replay Protection In Blob Authentication Allows Signature Reuse	Medium	Resolved
EDA2-19	`nil` Pointer Dereference Upon Malformed `EncodeBlob()` Reply	Low	Resolved
EDA2-20	Lack Of Signature Replay Protection In `AuthenticatePaymentStateRequest()`	Low	Resolved
EDA2-21	Incorrect Check For `GetBlobCommitment()` `blobSize` Validation	Low	Resolved
EDA2-22	Incorrect Usage Recording For Failed Dispersal Requests	Low	Resolved
EDA2-23	Duplicate Validator Node Quorum Replies Increase Total Stake	Low	Resolved
EDA2-24	Inconsistent Use Of `blockNumber` In `GetPaymentVaultParams()`	Low	Resolved

EDA2-25	Use `ctx.Done()` And `select` Statement Instead Of Reading Directly From `runningRequests`	Informational	Resolved
EDA2-26	Duplicated Error Checks in `ValidateBlobs()`	Informational	Resolved
EDA2-27	Signed Stake Percentage Not Capped	Informational	Open
EDA2-28	Miscellaneous General Comments	Informational	Resolved

EDA2-01	`nil` Pointer Dereference In `HashGetChunksRequest()` Function		
Asset	`api/hashing/relay_hashing.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `HashGetChunksRequest()` function in `relay_hashing.go` contains a `nil` pointer dereference vulnerability that could lead to application crashes. The function does not properly validate that `GetByRange()` returns a non-nil value before attempting to access its fields.

```
// HashGetChunksRequest hashes the given GetChunksRequest.
func HashGetChunksRequest(request *pb.GetChunksRequest) []byte {
    hasher := sha3.NewLegacyKeccak256()

    hasher.Write(request.GetOperatorId())
    for _, chunkRequest := range request.GetChunkRequests() {
        if chunkRequest.GetByIndex() != nil {
            getByIndex := chunkRequest.GetByIndex()
            hasher.Write(iByte)
            hasher.Write(getByIndex.BlobKey)
            for _, index := range getByIndex.ChunkIndices {
                hashUint32(hasher, index)
            }
        } else {
            getByRange := chunkRequest.GetByRange()
            hasher.Write(rByte)
            hasher.Write(getByRange.BlobKey)
            hashUint32(hasher, getByRange.StartIndex)
            hashUint32(hasher, getByRange.EndIndex)
        }
    }

    return hasher.Sum(nil)
}
```

The code assumes that if `GetByIndex()` is nil, then `GetByRange()` must be non-nil. However, in protobuf's `oneof` field, it's possible that neither is set, which would cause both methods to return `nil`.

This vulnerability could be exploited by sending a malformed request to the gRPC endpoint `GetChunks()` in `relay/server.go`, where both fields are unset. The impact is rated as high as this would cause the relayer server to crash whenever a user calls `GetChunks()`. The likelihood is rated as high as the occurrence of the `nil` pointer is before signature validation in authentication. Therefore, unauthenticated users can call this function.

Recommendations

It is recommended to perform validation over these fields in `AuthenticateGetChunksRequest()` or `GetChunks()` such that a situation where `GetByRange()` and `GetByIndex()` do not both return `nil`. Consider adding a dedicated validation function for `GetChunksRequest`.

Additionally, avoid directly accessing gRPC structs. Use `getByRange.GetBlobKey()` rather than `getByRange.BlobKey`.

Resolution

The recommendation has been implemented in PR #1338.

EDA2-02	Missing Signature Replay Protection In `GetChunks()` & `StoreChunks()` Requests		
Asset	`node/auth/authenticator.go, relay/auth/authenticator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The authentication mechanism for `StoreChunks()` requests lacks replay protection, allowing attackers to capture and reuse legitimate signed requests to gain authentication. Once authenticated, the attacker's IP address is cached in the `authenticatedDispersers` LRU cache, enabling them to send arbitrary malicious data in subsequent requests without needing to pass authentication again.

The vulnerability stems from two parts, first in the `isAuthenticationStillValid()` function, which checks if the request origin (IP address) has been recently authenticated and if so authentication is skipped.

The second component is the lack of replay protection in `VerifyStoreChunksRequest()`, which allows any signature to be replayed indefinitely if it has been captured.

```
func (a *requestAuthenticator) AuthenticateStoreChunksRequest(
    ctx context.Context,
    origin string,
    request *grpc.StoreChunksRequest,
    now time.Time) error {

    if a.isAuthenticationStillValid(now, origin) {
        // We've recently authenticated this client. Do not authenticate again for a while.
        return nil
    }

    key, err := a.getDisperserKey(ctx, now, request.DisperserID)
    if err != nil {
        return fmt.Errorf("failed to get operator key: %w", err)
    }

    err = VerifyStoreChunksRequest(*key, request)
    if err != nil {
        return fmt.Errorf("failed to verify request: %w", err)
    }

    a.cacheAuthenticationResult(now, origin)
    return nil
}
```

Using these two issues, an attacker may:

1. Capture a valid authenticated request.
2. Replay this request to gain authentication status.
3. Send malicious payloads in subsequent requests during the authentication timeout window.
4. Bypass all security checks for those subsequent requests.
5. Replay the original authenticated request to restart the timeout window.

This vulnerability is particularly dangerous because the system explicitly trusts previously authenticated IP addresses for the duration of `authenticationTimeoutDuration` and a single valid signature can be replayed multiple times to gain another `authenticationTimeoutDuration` period of time.

An equivalent issue is present in the relay `AuthenticateGetChunksRequest()`. There is insufficient replay protection, while authentication caching exists, allowing an attacker to gain authentication indefinitely from a single captured signature.

Recommendations

Authentication requests must include replay protection. Consider adding a timestamp based solution where the signer must include a timestamp within 1 minute of the current time. The consumed timestamps can be cached by the server until the recency window expires (1 minute in this example).

It is recommended to remove the authentication caching to prevent front-running and man-in-the-middle attacks and IP address spoofing. Enforce each request to be signed.

Additionally, consider creating a common library for the similar logic in the node and relay authenticators.

Resolution

The replay protection issue was resolved by adding a timestamp to the signature and verifying that the timestamp is within a certain range.

Changes can be found in the following PR #1365, where a common library `ReplayGuardian` was created to handle the replay protection logic for both the node and relay authenticators.

EDA2-03	`nil` Pointer Dereference In `StoreChunks()` Due To Incorrect Validation Order		
Asset	`node/grpc/server_v2.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The gRPC endpoint `StoreChunks()` in `server_v2.go` is vulnerable to `nil` pointer dereference as the request authentication is performed before input validation. The authenticator attempts to access fields in the request that may be `nil`, leading to a panic.

The `nil` pointer error will occur further down the authentication call path in `HashStoreChunksRequest()`. The issue occurs as `request.Batch.Header` does not account for `Header` being `nil`.

```
func HashStoreChunksRequest(request *grpc.StoreChunksRequest) []byte {
    hasher := sha3.NewLegacyKeccak256()

    hashBatchHeader(hasher, request.Batch.Header) // @audit batch may be nil
    for _, blobCertificate := range request.Batch.BlobCertificates {
        hashBlobCertificate(hasher, blobCertificate)
    }
    hashUint32(hasher, request.DisperserID)

    return hasher.Sum(nil)
}
```

The issue can be seen higher up the call path in `StoreChunks()` where `AuthenticateStoreChunksRequest()` is called before `validateStoreChunksRequest()`, where `validateStoreChunksRequest()` is responsible for performing `nil` pointer checks.

```
func (s *ServerV2) StoreChunks(ctx context.Context, in *pb.StoreChunksRequest) (*pb.StoreChunksReply, error) {
    start := time.Now()

    if !s.config.EnableV2 {
        return nil, api.NewErrorInvalidArg("v2 API is disabled")
    }

    if s.authenticator != nil {
        disperserPeer, ok := peer.FromContext(ctx)
        if !ok {
            return nil, errors.New("could not get peer information")
        }
        disperserAddress := disperserPeer.Addr.String()

        err := s.authenticator.AuthenticateStoreChunksRequest(ctx, disperserAddress, in, time.Now()) // @audit index out of bounds
            ↪ panics in here, should do `validateStoreChunksRequest()` first
        if err != nil {
            return nil, fmt.Errorf("failed to authenticate request: %w", err)
        }
    }

    if s.node.StoreV2 == nil {
        return nil, api.NewErrorInternal("v2 store not initialized")
    }

    if s.node.BLSSigner == nil {
        return nil, api.NewErrorInternal("missing bls signer")
    }

    batch, err := s.validateStoreChunksRequest(in) // @audit nil pointer validation occurs in here
    if err != nil {
        return nil, err
    }
...
```

The impact is rated as high as it would allow peers to crash a node by sending a malformed `StoreChunks()` request. The issue occurs before the signature is validated and so any user is able to call this endpoint without signature authorisation, therefore the likelihood is also rated as high.

Recommendations

It is recommended to move the function `validateStoreChunksRequest()` before `AuthenticateStoreChunksRequest()`.

Furthermore, within `AuthenticateStoreChunksRequest()` and subcalls, do not directly access gRPC fields. Instead, use getter functions such as `request.GetBatch().GetHeader()`.

Resolution

The issue has been resolved in PR #1166 by altering the order of operations. Validation and `nil` pointer checks now occur before authentication.

Additionally, PR #1339 has been raised to ensure that fields are accessed using getter functions.

EDA2-04	`AccountID` String Manipulation		
Asset	`core/data.go, core/meterer/offchain_store.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `ConvertToPaymentMetadata()` function does not ensure that the `AccountID` string encoding is unique for a single address. This can lead to an issue when the `AccountID` is used in `offchain_store.go`. Specifically, different strings representing the same account could result in duplicate entries in the on-demand payment or reservation tables.

The function `ConvertToPaymentMetadata()` does not strictly validate the `ph.AccountId` string.

```
// ConvertToProtoPaymentHeader converts a PaymentMetadata to a protobuf payment header
func ConvertToPaymentMetadata(ph *commonpbv2.PaymentHeader) *PaymentMetadata {
    if ph == nil {
        return nil
    }

    return &PaymentMetadata{
        AccountID:         ph.AccountId,
        ReservationPeriod: ph.ReservationPeriod,
        CumulativePayment: new(big.Int).SetBytes(ph.CumulativePayment),
    }
}
```

For example, when this account is used in the on-demand table, the `AccountID` may be manipulated to create duplicate entries. The follow are a set of valid `string` entries for account `0x0000000011223344556677889911223344556677`:

- `"0x0000000011223344556677889911223344556677"`
- `"0000000011223344556677889911223344556677"`
- `"1122334455667788991122334455667 7"`
- `"0x1122334455667788991122334455667 7"`

Each of these duplicate entries can spend the same onchain `CumulativeAmount` balance.

```
func (s *OffchainStore) AddOnDemandPayment(ctx context.Context, paymentMetadata core.PaymentMetadata, symbolsCharged uint32) error
    ↪ {
    result, err := s.dynamoClient.GetItem(ctx, s.onDemandTableName,
        commondynamodb.Item{
            "AccountID":          &types.AttributeValueMemberS{Value: paymentMetadata.AccountID}, // @audit AccountID is not unique
                ↪ to an address
            "CumulativePayments": &types.AttributeValueMemberN{Value: paymentMetadata.CumulativePayment.String()},
        },
    )
    if err != nil {
        fmt.Println("new payment record: %w", err)
    }
    if result != nil {
        return fmt.Errorf("exact payment already exists")
    }
    err = s.dynamoClient.PutItem(ctx, s.onDemandTableName,
        commondynamodb.Item{
            "AccountID":          &types.AttributeValueMemberS{Value: paymentMetadata.AccountID},
            "CumulativePayments": &types.AttributeValueMemberN{Value: paymentMetadata.CumulativePayment.String()},
            "DataLength":         &types.AttributeValueMemberN{Value: strconv.FormatUint(uint64(symbolsCharged), 10)},
        },
    )

    if err != nil {
        return fmt.Errorf("failed to add payment: %w", err)
    }
    return nil
}
```

Recommendations

To resolve this issue, modify the type of `PaymentMetadata.AccountID` to be of `[20]byte` or a wrapper type such as `common.Address` from Geth. This will ensure different transmissions of the same address over the wire are treated the same when being processed by the disperser. Then, when required in string form convert the `[20]byte` type into a string to ensure the encoding is unique for an address.

Resolution

The issue has been resolved in PR #1335. Updates ensure all occurrences of `AccountID` are unique to a single address by enforcing the type as `common.Address`.

EDA2-05	Malicious User May Manipulate `ValidatePayment()` By Injecting Invalid Payments		
Asset	`core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A malicious user is able to inject a fraudulent payment into the `onDemandTable` of DynamoDB by calling `DisperseBlob()` with invalid parameters. This invalid payment will be treated as a valid payment for future operations.

The attack scenario consists of a user who has existing payments. Say there is an account with a single payment, C, with (`CumulativeAmount = 100` , `DataLength = 100`) and assuming that `pricePerSymbol = 1` .

To exploit this situation the attacker sends a request, B, with `numSymbols = 1` and `CumulativeAmount = 99` (assuming 1 is the minimum). This request will call `AddOnDemand()` and be inserted into the database. However, as it is an invalid payment it will cause an error in `ValidatePayment()` and the blob will not be dispersed.

```
func (m *Meterer) ServeOnDemandRequest(ctx context.Context, header core.PaymentMetadata, onDemandPayment *core.OnDemandPayment,
    ↪  numSymbols uint, headerQuorums []uint8) error {
    m.logger.Info("Recording and validating on-demand usage", "header", header, "onDemandPayment", onDemandPayment)
    quorumNumbers, err := m.ChainPaymentState.GetOnDemandQuorumNumbers(ctx)
    if err != nil {
        return fmt.Errorf("failed to get on-demand quorum numbers: %w", err)
    }

    if err := m.ValidateQuorum(headerQuorums, quorumNumbers); err != nil {
        return fmt.Errorf("invalid quorum for On-Demand Request: %w", err)
    }
    // update blob header to use the miniumum chargeable size
    symbolsCharged := m.SymbolsCharged(numSymbols)
    err = m.OffchainStore.AddOnDemandPayment(ctx, header, symbolsCharged) // @audit writes invalid requests to DB
    if err != nil {
        return fmt.Errorf("failed to update cumulative payment: %w", err)
    }
    // Validate payments attached
    err = m.ValidatePayment(ctx, header, onDemandPayment, numSymbols)
    if err != nil {  // @audit will not remove payment from DB if invalid
        // No tolerance for incorrect payment amounts; no rollbacks
        return fmt.Errorf("invalid on-demand payment: %w", err)
    }
...
```

Now, the attacker's `onDemandTable` will consist of two conflicting entries:

- B: (CumulativeAmount = 99, DataLength = 1)
- C: (CumulativeAmount = 100, DataLength = 100)

With the user's table in this state they are able to re-use the `CumulativeAmount` in the range 0-98. For example, they may request a new payment, A, with `CumulativeAmount = 98` and `DataLength = 98` . This will pass `ValidatePayment()` as `nextPmt` and `nextPmtnumSymbols` will be that of entry B rather than C.

The following lines of `ValidatePayment()` will succeed due to the injected payment.

```
// the current request must not break the payment magnitude for the next payment if the two requests were delivered out-of-order
if nextPmt.Cmp(big.NewInt(0)) != 0 && header.CumulativePayment.Add(header.CumulativePayment,
      ↪  m.PaymentCharged(uint(nextPmtnumSymbols))).Cmp(nextPmt) > 0 {
    return fmt.Errorf("breaking cumulative payment invariants")
}
```

Hence, the blob will be dispersed and the final table for the user will be as follows:

- A: (CumulativeAmount = 98, DataLength = 98)
- B: (CumulativeAmount = 99, DataLength = 1)
- C: (CumulativeAmount = 100, DataLength = 100)

The impact is rated as high as users may spend the same balance numerous times allowing for significantly reduced cost of storage.

Recommendations

To resolve the issue, it is recommended to only write valid payments into the `onDemandTable` of DynamoDB. To achieve this `AddOnDemandPayment()` should occur after `RemoveOnDemandPayment()`.

Resolution

The issue has been resolved in PR #1386.

Modifications reduce the DynamoDB to a single entry for each account. The entry is updated with the latest cumulative payment amount. This ensures that the `onDemandTable` contains only the current cumulative amount.

Furthermore, the disperser client has been updated to only allow a single request at a time. Changes occur in PR #1389, which adds a mutex lock prevent concurrent threads from accessing the same account.

EDA2-06	Ambiguous Hashing Implementation Without Length Encoding Creates Collision Risk		
Asset	`api/hashing/node_hashing.go, api/hashing/relay_hashing.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The hashing implementations in `node_hashing.go` and `relay_hashing.go` do not properly handle variable-length fields, creating a risk of hash collisions where different inputs could produce identical hashes. This vulnerability could allow an attacker to craft malicious messages that generate the same hash as legitimate messages, bypassing signature verification.

The key issue is that the hashing functions directly concatenate byte arrays without encoding their lengths or using proper delimiters:

```
func hashBlobCertificate(hasher hash.Hash, blobCertificate *common.BlobCertificate) {
    hashBlobHeader(hasher, blobCertificate.BlobHeader)
    hasher.Write(blobCertificate.Signature)  // No length prefix
    for _, relayKey := range blobCertificate.RelayKeys { // No array length included
        hashUint32(hasher, relayKey)
    }
}

func hashPaymentHeader(hasher hash.Hash, header *common.PaymentHeader) {
    hasher.Write([]byte(header.AccountId))  // No length prefix
    hashUint32(hasher, header.ReservationPeriod)
    hasher.Write(header.CumulativePayment)  // No length prefix
}
```

This creates ambiguity where different combinations of field values could produce identical byte sequences. For example:

- AccountId = "abc", ReservationPeriod = "0000" CumulativePayment = "1234"
- AccountId = "abc00000", ReservationPeriod = "1234", CumulativePayment = ""

Would result in the same byte sequence when concatenated without length encoding, `abc000001234`.

Recommendations

Implement length prefixing for all variable-length fields and array counts in the hashing functions. Length prefixing should be fixed size, for example 32 bits, which should be sufficient for most of the lengths of most fields.

Furthermore, consider adding additional validation for fields which have a fixed size. This may be achieved by first calling `validateStoreChunksRequest()` and implementing a similar function for the relay's `GetChunks()`.

Resolution

The recommendation has been implemented in PR #1351.

EDA2-07	Index Out Of Bounds Panic In `DeserializeGnark()`		
Asset	`encoding/serialization.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `DeserializeGnark()` function in `serialization.go` contains an index out of bounds panic when attempting to unmarshal `data`.

In the function below, there are no checks to ensure that the input `data` buffer has a length of at least `bn254.SizeOfG1AffineCompressed` before accessing the slice.

```
func (c *Frame) DeserializeGnark(data []byte) (*Frame, error) {
    var f Frame
    buf := data
    err := f.Proof.Unmarshal(buf[:bn254.SizeOfG1AffineCompressed]) // @audit index out of bounds panic
    if err != nil {
        return nil, err
    }
...
```

If a malicious or malformed input with a length less than `bn254.SizeOfG1AffineCompressed` is provided, the slice operation `buf[:bn254.SizeOfG1AffineCompressed]` will trigger a runtime panic.

This is rated as high impact because it could allow an attacker to crash services that process externally-provided serialised frames.

Recommendations

Add a length check at the beginning of the function to ensure the buffer has sufficient data before attempting to unmarshal.

Resolution

A length check has been added in PR #1340.

EDA2-08	Usage Is Not Rolled Back On Failed Requests		
Asset	`core/meterer/meterer.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

While serving dispersal requests, validation occurs after updating the database entries. However, if validation fails, the database entries are not rolled back, causing invalid payment entries to persist.

In `ServeOnDemandRequest()`, new usage is added to the database before payment validation.

```
// update blob header to use the miniumum chargeable size
symbolsCharged := m.SymbolsCharged(numSymbols)
err = m.OffchainStore.AddOnDemandPayment(ctx, header, symbolsCharged) // @audit use payment stored
if err != nil {
    return fmt.Errorf("failed to update cumulative payment: %w", err)
}

// Validate payments attached
err = m.ValidatePayment(ctx, header, onDemandPayment, numSymbols) // @audit user payment validated
if err != nil {
    // No tolerance for incorrect payment amounts; no rollbacks
    return fmt.Errorf("invalid on-demand payment: %w", err)
}
```

In some cases, such as failed reads from DynamoDB due to network issues, `ValidatePayment()` may fail. When this happens, the newly added entry should be removed, but it is not. Therefore, the user will be charged if there is a failed DB read during their payment.

A similar issue occurs with reservation requests. In `IncrementBinUsage()`, the reservation bin is updated first, and overflow usage is validated after.

```
func (m *Meterer) IncrementBinUsage(ctx context.Context, header core.PaymentMetadata, reservation *core.ReservedPayment, numSymbols
    ↪ uint) error {
    symbolsCharged := m.SymbolsCharged(numSymbols)
    newUsage, err := m.OffchainStore.UpdateReservationBin(ctx, header.AccountID, uint64(header.ReservationPeriod),
        ↪ uint64(symbolsCharged))
    if err != nil {
        return fmt.Errorf("failed to increment bin usage: %w", err)
    }

    // metered usage stays within the bin limit
    usageLimit := m.GetReservationBinLimit(reservation)
    if newUsage <= usageLimit {
        return nil
    } else if newUsage-uint64(symbolsCharged) >= usageLimit {
        // metered usage before updating the size already exceeded the limit
        return fmt.Errorf("bin has already been filled")
    }
    if newUsage <= 2*usageLimit && header.ReservationPeriod+2 <= GetReservationPeriod(reservation.EndTimestamp,
        ↪ m.ChainPaymentState.GetReservationWindow()) {
        _, err := m.OffchainStore.UpdateReservationBin(ctx, header.AccountID, uint64(header.ReservationPeriod+2),
            ↪ newUsage-usageLimit)
        if err != nil {
            return err
        }
        return nil
    }
    return fmt.Errorf("overflow usage exceeds bin limit")``
}
```

However, if the overflow usage exceeds the bin limit or the `UpdateReservationBin()` query fails, the usage is not rolled back. Since these entries correspond to account usage, users may experience losses due to incorrectly recorded usage when they attempt to use dispersal requests later.

Recommendations

The issue may be mitigated by performing all validation before updating the database.

Resolution

The issue is a know design choice by the development team. The following response was provided by the development team:

> *The writes are done first by the design of high concurrency atomic writes. We've documented this behavior in the docs that there's no usage rollback on failed requests.*

EDA2-09	`PricePerSymbol` Is Liable To Change Impacting Existing Payments		
Asset	`core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When an onchain update is made to the `pricePerSymbol` variable, this will impact all previous offchain payments.

Payments are allowed to be made retrospectively in case they arrive out of order. The retroactive payment system uses the current onchain `pricePerSymbol`. The previous payments are calculated in the function `PaymentCharged()` as:

```
paymentCharged = symbolsCharged * pricePerSymbol
```

This will impact the calculations in `ValidatePayment()`, specifically the line related to `nextPmt`.

```
if nextPmt.Cmp(big.NewInt(0)) != 0 && header.CumulativePayment.Add(header.CumulativePayment,
    ↪  m.PaymentCharged(uint(nextPmtnumSymbols))).Cmp(nextPmt) > 0 { // @audit modifies header.CumulativePayment
```

If the `pricePerSymbol` has decreased, this will reduce the charge of the next payment. Therefore, a gap may exist between cumulative amounts `prevPmt` and `nextPmt` when taking into account the charged amount. A malicious user may then spend the balance equal to the gap between these two accounts.

Consider the following example: Say we currently have two payments A and C and the price per symbol is 2.

- A: (CumulativeAmount 60, symbols 30)
- C: (CumulativeAmount 160, symbols 50)

As it sits with price per symbol of 2, this has consumed all of the `CumulativeAmount` between 0 to 160. Now, say that the price per symbol drops from 2 to 1. We will then be able to insert a value between A and C. Such as:

- B: (CumulativeAmount 110, symbols 50)

This would make our final table sorted as follows:

- A: (CumulativeAmount 60, symbols 30)
- B: (CumulativeAmount 110, symbols 50)
- C: (CumulativeAmount 160, symbols 50)

(Note: Which holds true when the price per symbol is 1.)

The inverse is true if the price per symbol increases. The user would end up with a situation where the previous payments will have the charged amount increase. This will only pose an issue to users if they have existing gaps between their payments that were intended for future use.

The issue occurs as the price per symbol of a payment is not locked in when the payment occurs but is variable over time. The impact is long term users may get significantly more (or less) value depending on how many on-demand payments they have already made.

Recommendations

The issue may be resolved by fixing the price of each payment when it is received. That is, by storing the `PricePerSymbol` along with the `CumulativeAmount` and `DataLength`. Thereby, fixing the price of a payment.

Alternatively, consider storing the `Charge` of the payment rather than `DataLength` and `PricePerSymbol`.

Resolution

The recommendation was implemented in PR #1346. However, PR #1386 reduces each account to a single database record. Hence, the issue is no longer applicable.

EDA2-10	Commitment Validation Should Be Performed Before Metering		
Asset	`disperser/apiserver/disperse_blob_v2.go, core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

While validating dispersal requests in the function `validateDispersalRequest()`, the blob is checked to ensure that the commitments provided in the header match the actual blob in the request.

```
paymentHeader := core.PaymentMetadata{
    AccountID:         accountID,
    ReservationPeriod: reservationPeriod,
    CumulativePayment: cumulativePayment,
}

err = s.meterer.MeterRequest(ctx, paymentHeader, blobLength, blobHeader.QuorumNumbers) // @audit metering occurs here
if err != nil {
    return api.NewErrorResourceExhausted(err.Error())
}

commitments, err := s.prover.GetCommitmentsForPaddedLength(blob) // @audit verification begins here
if err != nil {
    return api.NewErrorInternal(fmt.Sprintf("failed to get commitments: %v", err))
}
if !commitments.Equal(&blobHeader.BlobCommitments) {
    return api.NewErrorInvalidArg("invalid blob commitment")
}
```

However, this validation occurs after the usage is metered for the user's account. If an attacker were to obtain a valid signature, they could front run the genuine call to `DisperseBlob` with a malicious `blob` parameter. The attacker may use the same signature and modify the blob in the request to its maximum possible length, ensuring that the OnDemand payment validation passes and is charged the maximum value. Note this must match the `CumulativePayments` bounds checks when metering payments.

```
err = m.ValidatePayment(ctx, header, onDemandPayment, numSymbols) // @audit does not validate blob
if err != nil {
    // No tolerance for incorrect payment amounts; no rollbacks
    return fmt.Errorf("invalid on-demand payment: %w", err)
}
```

Since the commitment validation happens after payment metering, the dispersal will fail, but the usage for user's deposits will still be maxed out due to metering. As a result, the user may have their on-demand payments consumed and the genuine deposit will not be processed.

Recommendations

All validations on the request should be performed before metering is persisted. To resolve this issue, commitment validation logic should be performed before executing `s.meterer.MeterRequest()`.

However, this resolution has a trade-off in that the commitment generation is computationally heavy and may represent a DoS vector in terms of resource constraints. An alternative solution may be to include a fast but collision resistant hash digest of the blob in the authentication message.

Resolution

Validation of commitments has been updated to occur before payments in PR #1170.

EDA2-11	Lack Of Domain Separation In Message Hashing Functions		
Asset	`api/hashing/node_hashing.go, api/hashing/relay_hashing.go, core/auth/v2/authenticator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The hashing functions in `node_hashing.go` and `relay_hashing.go` lack proper domain separation when hashing different types of objects for signature verification. This creates a vulnerability where two different message types could produce identical hashes if they happen to contain the encoded bytes array.

Domain separation is a cryptographic best practice that ensures hashes created in one context cannot be reused in another. In the current implementation, various hashing functions (e.g., `HashGetChunksRequest()`, `HashStoreChunksRequest()`, etc.) directly hash the raw data without including any type identifiers.

Similarly, message hashing in `AuthenticateBlobRequest()` and `AuthenticatePaymentStateRequest()` do not contain domain separation between request types.

Without domain separation, there's a risk of signature substitution attacks where a signature valid for one message type could potentially be reused for another message type with the same binary representation.

Recommendations

Implement domain separation by prefixing each hash with a fixed string that identifies the type of object being hashed.

Resolution

Domain separation has been added in PR #1358.

EDA2-12	Multiplication Overflow In `ValidateEncodingParams()`		
Asset	`encoding/params.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

In the `ValidateEncodingParams()` function, there is a potential multiplication overflow when calculating `params.ChunkLength*params.NumChunks`.

If the values of `params.ChunkLength` and `params.NumChunks` are large enough, their product could exceed the maximum value representable by a `uint64`, leading to an overflow.

```
// ValidateEncodingParams takes in the encoding parameters and returns an error if they are invalid.
func ValidateEncodingParams(params EncodingParams, SRSOrder uint64) error {

    // Check that the parameters are valid with respect to the SRS. The precomputed terms of the amortized KZG
    // prover use up to order params.ChunkLen*params.NumChunks-1 for the SRS, so we must have
    // params.ChunkLen*params.NumChunks-1 <= g.SRSOrder. The condition below could technically
    // be relaxed to params.ChunkLen*params.NumChunks > g.SRSOrder+1, but because all of the paramters are
    // powers of 2, the stricter condition is equivalent.
    if params.ChunkLength*params.NumChunks > SRSOrder {
        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) // @audit overflow
    }

    return nil

}
```

When an overflow occurs, it is possible to have the multiplication lower than `SRSOrder` but each of `ChunkLength` and `NumChunks` larger than `SRSOrder`.

Recommendations

Add a check to ensure that the multiplication of `params.ChunkLength` and `params.NumChunks` will not overflow. Consider the following code snippet.

```
const maxUint64 = ^uint64(0)

if params.ChunkLength != 0 && params.NumChunks > maxUint64/params.ChunkLength {
    return fmt.Errorf("multiplication overflow: ChunkLength: %d, NumChunks: %d", params.ChunkLength, params.NumChunks)
}

if params.ChunkLength*params.NumChunks > SRSOrder {
    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)
}
```

Resolution

The overflow check has been added in PR #1341.

EDA2-13	Incorrect Overflow Usage Check In Client		
Asset	`api/clients/v2/accountant.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The current validation in `BlobPaymentInfo()` incorrectly checks if the symbol usage is less than or equal to the `binLimit`.

```
if overflowPeriodRecord.Usage == 0 && relativePeriodRecord.Usage-symbolUsage < binLimit && symbolUsage <= binLimit { // @audit
    ↪  incorrectly checks symbolUsage <= binLimit
    overflowPeriodRecord.Usage += relativePeriodRecord.Usage - binLimit
    if err := QuorumCheck(quorumNumbers, a.reservation.QuorumNumbers); err != nil {
        return 0, big.NewInt(0), err
    }
    return currentReservationPeriod, big.NewInt(0), nil
}
```

Consider this example:

- BinLimit = 3000
- CurrentBin usage = 2900
- New usage request = 3100
- (currentBin + 2) is empty

In this scenario:

1. Total usage would be 2900 + 3100 = 6000
2. This should be considered valid because:
    - `currentBin` would take 100 (reaching its limit of 3000)
    - `(currentBin + 2)` would take the remaining 3000

However, the current check `symbolUsage <= binLimit` fails because 3100 > 3000, incorrectly rejecting a valid overflow case. As a result, a valid reservation request through the client might fail. The validation should be modified to `symbolUsage < 2 * binLimit`. This ensures consistency with the dispersal client's overflow usage check and correctly handles valid overflow cases.

Recommendations

Modify the overflow usage check in `BlobPaymentInfo()` as shown below.

```
if overflowPeriodRecord.Usage == 0 && relativePeriodRecord.Usage-symbolUsage < binLimit && symbolUsage <= (2 * binLimit) {
// ... snipped
}
```

Resolution

After discussions with the development team it has been determined that the issue describes the intended behaviour of the system. The current behaviour requires each request to fit within a single bin, which is not the case if the recommendation were to be implemented. Therefore, no changes will be made to the code.

EDA2-14	Incorrect Reservation Period Validation		
Asset	`core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The validation logic in `ValidateReservationPeriod()` inclusively checks for `StartReservationPeriod` and `EndReservationPeriod`, allowing additional reservation periods to be included.

```
func (m *Meterer) ValidateReservationPeriod(header core.PaymentMetadata, reservation *core.ReservedPayment) bool {
    now := uint64(time.Now().Unix())
    reservationWindow := m.ChainPaymentState.GetReservationWindow()
    currentReservationPeriod := GetReservationPeriod(now, reservationWindow)
    // Valid reservation periodes are either the current bin or the previous bin
        // @audit inclusive check for reservation period
    if (header.ReservationPeriod != currentReservationPeriod && header.ReservationPeriod != (currentReservationPeriod-1)) ||
        ↪  (GetReservationPeriod(reservation.StartTimestamp, reservationWindow) > header.ReservationPeriod ||
        ↪  header.ReservationPeriod > GetReservationPeriod(reservation.EndTimestamp, reservationWindow)) {
        return false
    }
    return true
}
```

Since `GetReservationPeriod()` performs integer division by rounding down the result, the reservation period remains valid for an additional duration equal to the `reservationWindow`.

```
func GetReservationPeriod(timestamp uint64, binInterval uint32) uint32 {
    if binInterval == 0 {
        return 0
    }
    return uint32(timestamp) / binInterval // @audit integer division rounds down
}
```

Consider this example:

- StartTime = 1740096000 (Feb 21, 2025 00:00:00 GMT)
- EndTime = 1740268800 (Feb 23, 2025 00:00:00 GMT)
- ReservationWindow = 86400 (1 day)
- StartReservationPeriod = 20140
- EndReservationPeriod = 20142

In this case, the user is charged for 2 days of reservation. However, if a user attempts to use the reservation on February 23, 2025 23:00:00 GMT (23 hours after the reservation's end time):

- Timestamp = 1740351600
- ReservationPeriod = 20142

The `ValidateReservationPeriod()` check still passes. As a result, every account effectively gets to use their reservation for an additional 24 hours after their paid reservation period ends.

This causes significant revenue leakage for the protocol. While the impact is currently medium because the `reservationWindow` is set to 5 minutes, the impact would be higher if the window duration is increased in the future.

Recommendations

Currently, the validation check passes if `ReservationPeriod` is equal to `EndReservationPeriod`. Modify it so that it fails in this case.

```
if (header.ReservationPeriod != currentReservationPeriod && header.ReservationPeriod != (currentReservationPeriod-1)) ||
    ↪ (GetReservationPeriod(reservation.StartTimestamp, reservationWindow) > header.ReservationPeriod ||
    ↪ header.ReservationPeriod >= GetReservationPeriod(reservation.EndTimestamp, reservationWindow)) {
    return false
}
```

Resolution

The validation logic in `ValidateReservationPeriod()` has been updated to fail if `ReservationPeriod` is equal to `endPeriod`.

Changes can be seen in PR #1321.

EDA2-15	Changes In Reservation Period Interval Affects Current Reservations		
Asset	`core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `reservationPeriodInterval` can be updated by governance at any time. If this value is modified, it may break existing reservations for accounts. In the offchain code, this parameter is referred to as `reservationWindow`.

When users submit reservation requests, their usage is updated for the current bin. In the database, every time a user makes a reservation request, the usage is recorded using `UpdateReservationBin()` for the entry `(accountId, reservationPeriod, symbolCharged)`. The `currentReservationPeriod` represents the current bin and is calculated by dividing the current timestamp by the `reservationWindow`.

```
func GetReservationPeriod(timestamp uint64, binInterval uint32) uint32 {
    if binInterval == 0 {
        return 0
    }
    return uint32(timestamp) / binInterval
}
```

However, if the `reservationWindow` interval changes, the identifier for the current bin also changes. Since there will be no existing entry in the database corresponding to the new `reservationPeriod`, the current bin effectively resets, allowing the user to utilise its full capacity again.

Additionally, the `(reservationPeriod - 1)` and `(reservationPeriod + 2)` bins will also be reset in a similar manner. Because previous reservations and overflow usage are permitted, users will be able to use these bins again, gaining an unintended advantage.

Consider the example below:

- Start Timestamp: 1740132000 (2025-02-21T10:00:00Z)
- End Timestamp: 1740133800 (2025-02-21T10:30:00Z)
- Reservation Window: 5 minutes (300 seconds)
- Symbols Per Second: 1 (for simplicity)

With a 5-minute reservation window, the reservation periods are divided into 6 bins:

- 5800440 (Starts at 10:00:00)
- 5800441 (Starts at 10:05:00)
- 5800442 (Starts at 10:10:00)
- 5800443 (Starts at 10:15:00)
- 5800444 (Starts at 10:20:00)

- 5800445 (Starts at 10:25:00)

Current Usage:

- Assume the current time is 10:17:00, meaning the current bin is 5800443.
- The user has fully used up 5800440, 5800441, 5800442, and 50% of 5800443.
- At this point, the user has consumed 1050 symbols (300 symbols per bin for three bins + 150 symbols from the current bin).
- The user has 750 symbols remaining in the current and future bins.

After `reservationWindow` Change:

Now, the reservation window is changed to 10 minutes, resulting in the following reservation periods:

- 2900220 (Starts at 10:00:00)
- 2900221 (Starts at 10:10:00)
- 2900222 (Starts at 10:20:00)

Since the current time is 10:17:00, the bins 2900220 and 2900221 have no previous entries in the database. This effectively resets usage for those periods, allowing the user to:

- Immediately use 1200 symbols (600 per bin from 2900220 and 2900221).
- Use 600 more symbols later (from 2900222).

Due to the change in `reservationWindow`, the user now has access to 1800 additional symbols, even though they were originally allowed to use only 750 more symbols.

Furthermore, there is a possibility that the new reservation periods will overlap with already consumed periods. This could prevent a user from using their reservation periods. Using the example above, if a long term user has already filled the periods 2900220 onwards, these database entries would be non-empty.

Recommendations

A number of alternatives exist to partially mitigate this issue in accounting for overlapping periods.

First is to prevent changes to `reservationPeriodInterval`. This will ensure the periods never change.

A second option is to clear the `reservationTable` in DynamoDB when `reservationPeriodInterval` has passed. This mitigation will prevent any overlap between previous periods and the new periods.

Third, an additional field could be added to the table `reservationPeriodInterval`. Thus, database queries could be made such that they only intersect if the interval is the same.

Fully mitigating the issue is non-trivial and would require accounting for previous periods after a change in window size.

Resolution

The development team have decided to implement the resolution in PR #1350. The resolution involves changing the reservation period index to the start of the period timestamp.

EDA2-16	`header.CumulativePayment` Is Modified In `ValidatePayment()`		
Asset	`core/meterer/meterer.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In the function `ValidatePayment()`, the call to `header.CumulativePayment.Add(...)` will modify the value of `header.CumulativePayment` to the addition. This issue will have a high impact as this value is used at a later point to remove payments from the database.

```
func (m *Meterer) ValidatePayment(ctx context.Context, header core.PaymentMetadata, onDemandPayment *core.OnDemandPayment,
    ↪  numSymbols uint) error {

        // ... snipped

    // the current request must not break the payment magnitude for the next payment if the two requests were delivered
        ↪  out-of-order
    if nextPmt.Cmp(big.NewInt(0)) != 0 && header.CumulativePayment.Add(header.CumulativePayment,
        ↪  m.PaymentCharged(uint(nextPmtnumSymbols))).Cmp(nextPmt) > 0 { // @audit modifies header.CumulativePayment
        return fmt.Errorf("breaking cumulative payment invariants")
    }

    return nil
}
```

The following code snippet is taken from `ServeOnDemandRequest()` and demonstrates how `header.CumulativePayment` is used after it has been modified in `ValidatePayment()`.

```
func (m *Meterer) ServeOnDemandRequest(ctx context.Context, header core.PaymentMetadata, onDemandPayment *core.OnDemandPayment,
    ↪  numSymbols uint, headerQuorums []uint8) error {

        // ... snipped

    // Validate payments attached
    err = m.ValidatePayment(ctx, header, onDemandPayment, numSymbols) // @audit modifies header.CumulativePayment
    if err != nil {
        // No tolerance for incorrect payment amounts; no rollbacks
        return fmt.Errorf("invalid on-demand payment: %w", err)
    }

    // Update bin usage atomically and check against bin capacity
    if err := m.IncrementGlobalBinUsage(ctx, uint64(symbolsCharged)); err != nil {
        //TODO: conditionally remove the payment based on the error type (maybe if the error is store-op related)
        dbErr := m.OffchainStore.RemoveOnDemandPayment(ctx, header.AccountID, header.CumulativePayment) // @audit uses modified
            ↪  header.CumulativePayment
        if dbErr != nil {
            return dbErr
        }
        return fmt.Errorf("failed global rate limiting: %w", err)
    }
```

The issue is rated as high impact as it will remove this payment from the store if there is an error in the call to `IncrementGlobalBinUsage()`. However, the likelihood is rated as low due to the requirements for `IncrementGlobalBinUsage()` to cause an error.

Recommendations

Update the addition statement in `ValidatePayment()` such that it does not modify `header.CumulativePayment`.

Resolution

The issue has been resolved in PR #1322 by updating the addition statement in `ValidatePayment()` such that it does not modify `header.CumulativePayment`.

Note that the function `ValidatePayment()` has been removed in PR #1386.

EDA2-17	Lossy & Unnecessary Casting		
Asset	/*		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

core/meterer/meterer.go

The input to `PaymentCharged()` is a `uint` which is then cast to `int64` before being set as a `big.Int`. It is recommended to update this function to take `numSymbols uint64` as the function parameter. Then use `big.SetUint64()` to avoid any casting overflows.

```
func (m *Meterer) PaymentCharged(numSymbols uint) *big.Int {
    symbolsCharged := big.NewInt(int64(m.SymbolsCharged(numSymbols)))
    pricePerSymbol := big.NewInt(int64(m.ChainPaymentState.GetPricePerSymbol()))
    return symbolsCharged.Mul(symbolsCharged, pricePerSymbol)
}
```

In the function `SymbolsCharged()`, the value `numSymbols uint` is multiplied and divided before being cast to a `uint32`. This may potentially cause truncation if the value of `numSymbols` does not fit in a `uint32`. It is recommended to set `numSymbols` to a `uint64` and check for overflows.

```
func (m *Meterer) SymbolsCharged(numSymbols uint) uint32 {
    if numSymbols <= uint(m.ChainPaymentState.GetMinNumSymbols()) {
        return m.ChainPaymentState.GetMinNumSymbols()
    }
    // Round up to the nearest multiple of MinNumSymbols
    return uint32(core.RoundUpDivide(uint(numSymbols), uint(m.ChainPaymentState.GetMinNumSymbols())) *
        ↪  m.ChainPaymentState.GetMinNumSymbols()
}
```

In `GetReservationPeriod()`, the `timestamp` value is downcast to a `uint32` after division, potentially truncating the value during casting. It is recommended to use `uint64` for both `binInterval` and the return value.

```
func GetReservationPeriod(timestamp uint64, binInterval uint32) uint32 {
    if binInterval == 0 {
        return 0
    }
    return uint32(timestamp) / binInterval
}
```

core/meterer/onchain_state.go

Numerous interface functions in `OnchainPayment` represent onchain `uint64` values but are stored offchain as `uint32`s. Update each of the following types to be `uint64` in both `OnchainPaymentState` and `OnchainPayment`.

- `GetGlobalRatePeriodInterval() uint32`

- `GetMinNumSymbols() uint32`
- `GetPricePerSymbol() uint32`
- `GetReservationWindow() uint32`

`core/eth/reader.go`

On line [**742**] and line [**743**] of `core/eth/reader.go`, the variables `params.NumChunks` and `params.MaxNumOperators` are both already `uint32` and do not need to be cast.

```
return &core.BlobVersionParameters{
    CodingRate:      uint32(params.CodingRate),
    NumChunks:       uint32(params.NumChunks),
    MaxNumOperators: uint32(params.MaxNumOperators),
}, nil
```

Recommendations

Ensure these issues are understood and implement the related recommendations where applicable.

Resolution

PR #1327 updates the required functions to avoid unnecessary casting. The changes include updating struct fields and parameter types to `uint64`.

EDA2-18	Lack Of Replay Protection In Blob Authentication Allows Signature Reuse		
Asset	`core/v2/serialization.go, core/auth/v2/authenticator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Dispersal requests are authenticated using signatures. The signed message is the `BlobKey`, which is generated by combining the header hash and payment metadata hash. However, it does not include a `nonce` value.

```
// Pack the fields in of the blob header
packedBytes := arguments.Pack(
    b.BlobVersion,
    b.QuorumNumbers,
    abiBlobCommitments{
        Commitment: ...
        LengthCommitment: ...
        LengthProof: ...
        DataLength: ...
    },
    b.Salt,
)

// Pack and hash the two hashes together
paymentMetadataHash := b.PaymentMetadata.Hash()
packedBytes = arguments.Pack(struct {
    BlobHeaderHash      [32]byte
    PaymentMetadataHash [32]byte
}{
    headerHash,
    paymentMetadataHash,
})
```

Additionally, `AuthenticateBlobRequest()` does not enforce any mechanism to prevent the same signature from being replayed.

```
func (*authenticator) AuthenticateBlobRequest(header *core.BlobHeader, signature []byte) error {

    blobKey, err := header.BlobKey()
    if err != nil {
        return fmt.Errorf("failed to get blob key: %v", err)
    }

    // Recover public key from signature
    sigPublicKeyECDSA, err := crypto.SigToPub(blobKey[:], signature)
    if err != nil {
        return fmt.Errorf("failed to recover public key from signature: %v", err)
    }

    accountId := header.PaymentMetadata.AccountID
    accountAddr := common.HexToAddress(accountId)
    pubKeyAddr := crypto.PubkeyToAddress(*sigPublicKeyECDSA)

    if accountAddr.Cmp(pubKeyAddr) != 0 {
        return errors.New("signature doesn't match with provided public key")
    }

    return nil
}
```

If an attacker obtains a user's signature, they can replay it multiple times, which will be treated as a legitimate request from the user. In the case of a reservation request, the `Meterer` records the dispersal usage for the user's account. However, since the same dispersal request already exists in the blob store, it fails.

An attacker can repeatedly replay the signature within a reservation period to deplete the current reservation bin and overflow bin. It can disrupt the user's ability to make valid dispersal requests. The impact is medium as it only affects the current and overflow bins rather than the entire reservation period.

This attack would not work with `OnDemand` requests because replayed requests would be blocked when attempting to add them to the database.

```
func (s *OffchainStore) AddOnDemandPayment(ctx context.Context, paymentMetadata core.PaymentMetadata, symbolsCharged uint32) error
    ↪ {
    result, err := s.dynamoClient.GetItem(ctx, s.onDemandTableName,
        commondynamodb.Item{
            "AccountID":          &types.AttributeValueMemberS{Value: paymentMetadata.AccountID},
            "CumulativePayments": &types.AttributeValueMemberN{Value: paymentMetadata.CumulativePayment.String()},
        },
    )
    if result != nil {
        return fmt.Errorf("exact payment already exists")
    }
}
```

Recommendations

To resolve this issue, include a `nonce` in the message and store the current `nonce` in a persistent database. In `AuthenticateBlobRequest()`, validate the nonce in the message to ensure that the same signature cannot be reused.

Resolution

The issue is mitigated by first checking if the blob exists before metering payments. Changes can be seen in PR #1370.

EDA2-19	`nil` Pointer Dereference Upon Malformed `EncodeBlob()` Reply		
Asset	`disperser/encoder/client_v2.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

If the encoder server replies to an `EncodeBlob()` request with a malformed reply, the disperser server may trigger a `nil` pointer dereference panic.

In the function `EncodeBlob()` below, if `reply.FragmentInfo` is `nil` then the dereference will cause a panic.

```
func (c *clientV2) EncodeBlob(ctx context.Context, blobKey corev2.BlobKey, encodingParams encoding.EncodingParams, blobSize uint64)
    ↪ (*encoding.FragmentInfo, error) {

        // ... snipped

    // Make the RPC call
    reply, err := client.EncodeBlob(ctx, req)
    if err != nil {
        return nil, fmt.Errorf("failed to encode blob: %w", err)
    }

    // Extract and return fragment info
    return &encoding.FragmentInfo{
        TotalChunkSizeBytes: reply.FragmentInfo.TotalChunkSizeBytes, // @audit nil pointer if `reply.FragmentInfo` is nil
        FragmentSizeBytes:   reply.FragmentInfo.FragmentSizeBytes,
    }, nil
}
```

The issue is rated as low likelihood as the encoder server is not expected to be malicious.

Recommendations

It is recommended to use the gRPC getter functions to avoid `nil` pointer dereferences as seen in the code snippet below.

```
return &encoding.FragmentInfo{
    TotalChunkSizeBytes: reply.GetFragmentInfo().GetTotalChunkSizeBytes(),
    FragmentSizeBytes:   reply.GetFragmentInfo().GetFragmentSizeBytes(),
}, nil
```

Resolution

The recommendation has been implemented in PR #1404.

EDA2-20	Lack Of Signature Replay Protection In `AuthenticatePaymentStateRequest()`		
Asset	`core/auth/v2/authenticator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The signature for a payment state request does not expire. If an attacker were to gain access to the signature they could replay requests indefinitely.

```
func (*authenticator) AuthenticatePaymentStateRequest(sig []byte, accountId string) error {
    // Ensure the signature is 65 bytes (Recovery ID is the last byte)
    if len(sig) != 65 {
        return fmt.Errorf("signature length is unexpected: %d", len(sig))
    }

    // Verify the signature
    hash := sha256.Sum256([]byte(accountId))
    sigPublicKeyECDSA, err := crypto.SigToPub(hash[:], sig)
    if err != nil {
        return fmt.Errorf("failed to recover public key from signature: %v", err)
    }

    accountAddr := common.HexToAddress(accountId)
    pubKeyAddr := crypto.PubkeyToAddress(*sigPublicKeyECDSA)

    if accountAddr.Cmp(pubKeyAddr) != 0 {
        return errors.New("signature doesn't match with provided public key")
    }

    return nil
}
```

Recommendations

Consider including a timestamp in the request and reject requests outside of a reasonable bounds of the current server time, say 30 seconds.

Note that this solution has a limitation in that a significantly skewed server clock to client clock will result in rejected requests.

Alternate solutions include using and storing request nonces and rejecting replayed nonces.

Resolution

PR #1411 adds the `ReplayGuardian` to the payment state request end point.

EDA2-21	Incorrect Check For `GetBlobCommitment()` `blobSize` Validation		
Asset	`disperser/apiserver/server_v2.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In the function `GetBlobCommitment()`, there is a check that the length of the blob is less than `maxNumSymbolsPerBlob*encoding.BYTES_PER_SYMBOL`. However, this does not account for the rounding up of a blob to the next power of two.

```
func (s *DispersalServerV2) GetBlobCommitment(ctx context.Context, req *pb.BlobCommitmentRequest) (*pb.BlobCommitmentReply, error)
    ↪ {
    start := time.Now()
    defer func() {
        s.metrics.reportGetBlobCommitmentLatency(time.Since(start))
    }()

    if s.prover == nil {
        return nil, api.NewErrorUnimplemented()
    }
    blobSize := len(req.GetBlob()) // @audit GetBlobLengthPowerOf2
    if blobSize == 0 {
        return nil, api.NewErrorInvalidArg("data is empty")
    }
    if uint64(blobSize) > s.maxNumSymbolsPerBlob*encoding.BYTES_PER_SYMBOL {
        return nil, api.NewErrorInvalidArg(fmt.Sprintf("blob size cannot exceed %v bytes",
            ↪ s.maxNumSymbolsPerBlob*encoding.BYTES_PER_SYMBOL))
    }
```

The issue is rated as low likelihood and impact as it is expected that `maxNumSymbolsPerBlob` will be set as a power of two.

Recommendations

Consider matching the behaviour of `validateDispersalRequest()`:

```
blob := req.GetBlob()
 blobSize := len(blob)
 if blobSize == 0 {
     return api.NewErrorInvalidArg("blob size must be greater than 0")
 }
 blobLength := encoding.GetBlobLengthPowerOf2(uint(blobSize))
 if blobLength > uint(s.maxNumSymbolsPerBlob) {
     return api.NewErrorInvalidArg("blob size too big")
 }
```

Resolution

The length is rounded to the next power of two in PR #1407 before validating the number of symbols used.

EDA2-22	Incorrect Usage Recording For Failed Dispersal Requests		
Asset	`api/clients/v2/accountant.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

While generating a dispersal request in the client, validations are performed after the usage is recorded, resulting in usage being incorrectly marked even for failed requests

In the `BlobPaymentInfo` blob, the `relativePeriodRecord` is updated with usage before the `QuorumCheck()` validation is performed. If the `QuorumCheck()` fails, the dispersal ultimately fails. However, the `relativePeriodRecord` is still incremented, incorrectly marking usage for the failed request.

Similarly, `overflowPeriodRecord` and `cumulativePayment` are also incremented for failed requests, even when the request fails due to a failed `QuorumCheck()` validation.

```
func (a *Accountant) BlobPaymentInfo(ctx context.Context, numSymbols uint32, quorumNumbers []uint8) (uint32, *big.Int, error) {
        // ...snipped

    a.usageLock.Lock()
    defer a.usageLock.Unlock()
    relativePeriodRecord := a.GetRelativePeriodRecord(currentReservationPeriod)
    relativePeriodRecord.Usage += symbolUsage

    // first attempt to use the active reservation
    binLimit := a.reservation.SymbolsPerSecond * uint64(a.reservationWindow)
    if relativePeriodRecord.Usage <= binLimit {
                // @audit quorum validation is performed after relativePeriodRecord is incremented
        if err := QuorumCheck(quorumNumbers, a.reservation.QuorumNumbers); err != nil {
            return 0, big.NewInt(0), err
        }
        return currentReservationPeriod, big.NewInt(0), nil
    }

    overflowPeriodRecord := a.GetRelativePeriodRecord(currentReservationPeriod + 2)
    // Allow one overflow when the overflow bin is empty, the current usage and new length are both less than the limit
    if overflowPeriodRecord.Usage == 0 && relativePeriodRecord.Usage-symbolUsage < binLimit && symbolUsage <= binLimit {
        overflowPeriodRecord.Usage += relativePeriodRecord.Usage - binLimit
                // @audit quorum validation is performed after overflowPeriodRecord is incremented
        if err := QuorumCheck(quorumNumbers, a.reservation.QuorumNumbers); err != nil {
            return 0, big.NewInt(0), err
        }
        return currentReservationPeriod, big.NewInt(0), nil
    }

    // reservation not available, rollback reservation records, attempt on-demand
    //todo: rollback on-demand if disperser respond with some type of rejection?
    relativePeriodRecord.Usage -= symbolUsage
    incrementRequired := big.NewInt(int64(a.PaymentCharged(numSymbols)))
    a.cumulativePayment.Add(a.cumulativePayment, incrementRequired)
    if a.cumulativePayment.Cmp(a.onDemand.CumulativePayment) <= 0 {
                // @audit quorum validation is performed after cumulativePayment is incremented
        if err := QuorumCheck(quorumNumbers, requiredQuorums); err != nil {
            return 0, big.NewInt(0), err
        }
        return 0, a.cumulativePayment, nil
    }

    return 0, big.NewInt(0), fmt.Errorf("neither reservation nor on-demand payment is available")
}
```

This issue would affect regular users as almost all users rely on the client to generate dispersal requests. There is also a chance that users may sometimes provide invalid quorum numbers and the `accountant` will not be reset until the current client instance is destroyed. These two factors mean the issue has been rated as a medium impact.

Recommendations

To prevent this issue, perform quorum check before recording the usage.

Resolution

Bin usage is reverted when a dispersal request fails in PR #1406.

EDA2-23	Duplicate Validator Node Quorum Replies Increase Total Stake		
Asset	`core/aggregation.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When requesting replies from validator nodes that a chunk has been successfully stored, there is no validation that a node has not already replied. While no direct exploitation scenario was determined, this could lead to the stake voting on a quorum being incorrectly inflated if a validator managed to respond more than once.

When receiving validator node replies for chunk storage in the function `ReceiveSignatures()` the variable `signerMap` is used to mark each successful reply received.

However, the mapping lacks checks to ensure a validator has not already replied for this quorum. This makes it possible for a validator to reply more than once, incorrectly increasing the quorum stake that has been received.

```
for numReply := 0; numReply < numOperators; numReply++ {
    var err error
    r := <-messageChan

            // ... snipped

    op, found := state.IndexedOperators[r.Operator]
    if !found {
        a.Logger.Error("Operator not found in state", "operatorID", operatorIDHex, "operatorAddress", operatorAddr, "socket",
            ↪ socket)
        continue
    }

    //... snipped signature verification code

    operatorQuorums := make([]uint8, 0, len(quorumIDs))
    for _, quorumID := range quorumIDs {
        // Get stake amounts for operator
        ops := state.Operators[quorumID]
        opInfo, ok := ops[r.Operator]
        // If operator is not in quorum, skip
        if !ok {
            continue
        }
        operatorQuorums = append(operatorQuorums, quorumID)

        signerMap[r.Operator] = true

        // Add to stake signed
        stakeSigned[quorumID].Add(stakeSigned[quorumID], opInfo.Stake)

        //... snipped logging code
}
```

Currently, this issue would only present itself if duplicated operators existed in `state.IndexedOperators` and as this state is fetched from the onchain registry that prevents duplicated operators, this should never occur. As such, it has been assigned a low likelihood.

However, the testing team felt it was of importance to mention, given that if it becomes possible with future codebase changes, it could allow quorums to pass without the threshold quantity of validator nodes confirming successful chunk

storage.

Recommendations

It is recommended to check if a validator has already replied to a chunk storage request and if so not increment the `stakeSigned` of that quorum. This can be done using the `signerMap` already used for forming the aggregate signature of non-signers.

Resolution

The recommended duplication checks can be seen in PR #1356.

EDA2-24	Inconsistent Use Of `blockNumber` In `GetPaymentVaultParams()`		
Asset	`core/meterer/onchain_state.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `GetPaymentVaultParams()` function of the `OnchainPaymentState` struct, the `blockNumber` is retrieved using the `GetCurrentBlockNumber()` method. However, this `blockNumber` is only used for the `GetRequiredQuorumNumbers()` RPC call and is not utilised in any other RPC calls within the function. This could lead to inconsistencies in the data fetched pointing to different blocks.

An issue will occur if a block is produced between any of the RPC calls which update the payment vault parameters.

```
func (pcs *OnchainPaymentState) GetPaymentVaultParams(ctx context.Context) (*PaymentVaultParams, error) {
    blockNumber, err := pcs.tx.GetCurrentBlockNumber(ctx)
    if err != nil {
        return nil, err
    }
    quorumNumbers, err := pcs.tx.GetRequiredQuorumNumbers(ctx, blockNumber)
    if err != nil {
        return nil, err
    }

    globalSymbolsPerSecond, err := pcs.tx.GetGlobalSymbolsPerSecond(ctx)
    if err != nil {
        return nil, err
    }

    globalRatePeriodInterval, err := pcs.tx.GetGlobalRatePeriodInterval(ctx)
    if err != nil {
        return nil, err
    }

    minNumSymbols, err := pcs.tx.GetMinNumSymbols(ctx)
    if err != nil {
        return nil, err
    }

    pricePerSymbol, err := pcs.tx.GetPricePerSymbol(ctx)
    if err != nil {
        return nil, err
    }

    reservationWindow, err := pcs.tx.GetReservationWindow(ctx)
    if err != nil {
        return nil, err
    }

    return &PaymentVaultParams{
        OnDemandQuorumNumbers:    quorumNumbers,
        GlobalSymbolsPerSecond:   globalSymbolsPerSecond,
        GlobalRatePeriodInterval: globalRatePeriodInterval,
        MinNumSymbols:            minNumSymbols,
        PricePerSymbol:           pricePerSymbol,
        ReservationWindow:        reservationWindow,
    }, nil
}
```

The issue is rated as low likelihood as it would require a block to be produced between the RPC calls and furthermore, for that block to contain updates to the payment value configuration.

Recommendations

It is recommended to add the `blockNumber` parameter to each of the RPC calls such that they all occur at the same height.

Resolution

`blockNumber` has been added to each request in PR #1323.

EDA2-25	Use `ctx.Done()` And `select` Statement Instead Of Reading Directly From `runningRequests`
Asset	`disperser/encoder/server_v2.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the `EncodeBlob()` function, the `runningRequests` channel is used to limit the number of concurrent requests. However, the current implementation reads directly from the `runningRequests` channel without considering the context's cancellation or timeout.

```
// Limit the number of concurrent requests
s.runningRequests <- struct{}{}
defer s.popRequest()
if ctx.Err() != nil {
    s.metrics.IncrementCanceledBlobRequestNum(int(blobSize))
    return nil, status.Error(codes.Canceled, "request was canceled")
}
```

Recommendations

Update the code to use a select statement to handle context cancellation or timeout.

```
// Limit the number of concurrent requests
select {
case s.runningRequests <- struct{}{}:
    defer s.popRequest()
case <-ctx.Done():
    s.metrics.IncrementCanceledBlobRequestNum(int(blobSize))
    return nil, status.Error(codes.Canceled, "request was canceled")
}
```

Resolution

The recommendation has been implemented in PR #1405.

EDA2-26	Duplicated Error Checks in `ValidateBlobs()`
Asset	`core/v2/validator.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the function `ValidateBlobs()` on line [**158**], there are duplicate checks for `err != nil`. Only one occurrence is required.

```
params, err := blob.BlobHeader.GetEncodingParams(blobParams)
if err != nil {
    return err
}

if err != nil {
    return err
}
```

Recommendations

Remove the duplicate error check.

Resolution

The superfluous check has been removed in PR #1194.

EDA2-27	Signed Stake Percentage Not Capped
Asset	`core/aggregation.go`
Status	Open
Rating	Informational

Description

When determining the percentage of the quorum stake that has attested for a chunk's storage, there is no validation that this value lies in the acceptable value range of `[0,100]`. Given that this value is also then cast to a `uint8`, it is advisable to check this to prevent the possibility for overflows and prevent incorrect values being propagated onchain.

```
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())

    return quorumThreshold
}
```

This issue was noted as informational as there are currently no found instances which lead to the quorum becoming more than 100 percent.

Recommendations

Add validation logic to ensure the signed quorum percentage lies in the acceptable bound of `[0,100]`.

EDA2-28	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. **Converting `uint64` To String Then Reading `float64` From String Is Inefficient**

   Related Asset(s): common/aws/dynamodb/client.go

   The function `IncrementBy()` will convert the `uint64` variable `value` to a string before reading it into a `float64`.

   ```
   func (c *client) IncrementBy(ctx context.Context, tableName string, key Key, attr string, value uint64) (Item, error) {
       // ADD numeric values
       f, err := strconv.ParseFloat(strconv.FormatUint(value, 10), 64)
       if err != nil {
           return nil, err
       }
   ```

   It is recommended to simplify the process by using the Go in-built type conversion. This may contain small amounts of precision loss if the `uint64` value is large and cannot be representing as a `float64`.

   ```
   f := float64(value)
   ```

2. **Spelling & Grammar**

   The following spelling mistakes occur in the codebase.

   - `core/meterer/meterer.go`
     - line [**124**] `requrests` -> `requests`
     - line [**146**] `periodes` -> `periods`
     - line [**239**] `nextPmtnumSymbols` -> `nextPmtNumSymbols`
   - `disperser/dataapi/queried_operators_handlers.go`
     - line [**100**] `timestemp` -> `timestamp`
   - `core/data.go`
     - line [**23**] `wil` -> `will`
     - line [**374**] `formated` -> `formatted`
   - `core/assignment.go`
     - line [**191**] `amont` -> `amount`
   - `core/aggregation.go`
     - line [**320**] `accending` -> `ascending`
   - `api/clients/disperser_client.go`
     - line [**55**] `PROCESSSING` -> `PROCESSING`

- `api/docs/disperser.html`
  - line [**868**] `differet` -> `different`
  - line [**899**] `containg` -> `containing`
- `api/docs/disperser.md`
  - line [**318**] `differet` -> `different`
  - line [**332**] `containg` -> `containing`
- `api/docs/eigenda-protos.html`
  - line [**1792**] `differet` -> `different`
  - line [**1823**] `containg` -> `containing`
  - line [**3315**] `simiar` -> `similar`
- `api/docs/eigenda-protos.md`
  - line [**686**] `differet` -> `different`
  - line [**700**] `containg` -> `containing`
  - line [**1389**] `simiar` -> `similar`
- `api/docs/node.html`
  - line [**1012**] `simiar` -> `similar`
- `api/docs/node.md`
  - line [**388**] `simiar` -> `similar`
- `common/ratelimit/limiter_cli.go`
  - line [**39**] `multipiers` -> `multipliers`
- `common/abis/EigenDAServiceManager.json`
  - line [**1139**] `availabilty` -> `availability`

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 above can be seen in PR #1324.

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.

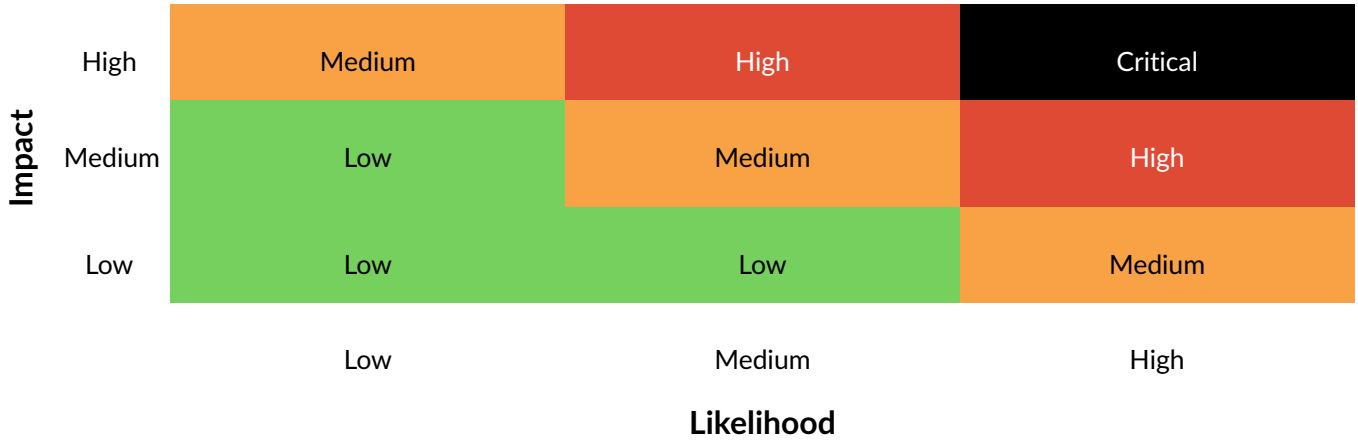

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].