EigenDA vCISO

Auditors

DTheo, Lead Security Researcher

Justin Traglia, Lead Security Researcher

Report prepared by: Lucas Goiriz

July 25, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

EigenLayer is a protocol built on Ethereum that introduces restaking, a new primitive in cryptoeconomic security.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of eigenda according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, EigenLayer engaged with Spearbit to review the eigenda protocol. In this period of time a total of **53** issues were found.

Summary

Project Name	EigenLayer
Repository	eigenda
Commit	91838b...d326
Type of Project	Data Availability, Cryptography
Audit Timeline	Feb 26 to Mar 8

Issues Found

Severity	**Count**
Critical Risk	5
High Risk	7
Medium Risk	12
Low Risk	10
Optimizations	4
Informational	15
Total	**53**

5 Findings

5.1 Critical Risk

5.1.1 `BitmapToQuorumIds()` will panic on malformed input causing DOS in multiple services

Severity: Critical Risk

Context: core/eth/tx.go#L783

Description: This is a similar issue to the one found in `bitmapToBytesArray()`. `BitmapToQuorumIds()` will `panic: underflow` when given `-0` as the `big.Int` to return a bitmap for. This happens when `bitmap.Bit()` is called:

```
func BitmapToQuorumIds(bitmap *big.Int) []core.QuorumID {
    quorumIds := make([]core.QuorumID, 0, maxNumberOfQuorums)
    for i := 0; i < maxNumberOfQuorums; i++ {
        // This call to bitmap.Bit() will panic when bitmap == -0 and i == 1
        if bitmap.Bit(i) == 1 {
            quorumIds = append(quorumIds, core.QuorumID(i))
        }
    }
    return quorumIds
}
```

The panic happens in /usr/local/go/src/math/big/nat.go in `(z nat) sub()`. The issue here is that there is no way to create a valid `-0` `big.Int` in regular practice, but deserialization from protobuf or `github.com/ethereum/go-ethereum/accounts/abi.ConvertType()` takes in raw byte data and assigns values to underlying struct fields. This allows someone craft a malicious `Big.Int` like this:

```
// here is what a big.Int looks like in its struct definition
type Int struct {
    neg bool // sign
    abs nat  // absolute value of the integer
}

// here is a malicious big.Int that is set to -0
// an attacker would have to make special code to set these
// fields directly before sending or would need to send
// a raw byte array with this payload
var malicious_bitmap big.Int
bitmap.neg = true
bitmap.abs = 0

// this call would trigger the panic: underflow
bitmap.bit(1)
```

Recommendation: There is no way to check the subfields directly since they are private so it is best to either add `recover()` handlers to handle this situation or add an explicit check like this:

```
// it is impossible for a big int to be less than 0 AND greater than -1
if bitmap.Cmp(big.NewInt(0)) == -1 && bitmap.Cmp(big.NewInt(-1)) == 1 {
    fmt.Println("Invalid bitmap")
    // error in some way
}
```

5.1.2 `bn254.DeserializeG1()` will panic on signatures smaller than 64 bytes allowing multiple service DOS

Severity: Critical Risk

Context: core/bn254/attestation.go#L111-L113

Description: `bn254.DeserializeG1()` will panic if it receives a byte slice smaller than 64 bytes. This function is reachable with data from multiple external requests, most noteably from the a DA node's response to the disperser's `StoreChunks` request.When the disperser enters its dispersal flow it will eventually make a `StoreChunks` request to each DA node.

```
func (c *dispatcher) sendChunks(ctx context.Context, blobs []*core.BlobMessage, header
↪  *core.BatchHeader, op *core.IndexedOperatorInfo) (*core.Signature, error) {
    conn, err := grpc.Dial(
    core.OperatorSocket(op.Socket).GetDispersalSocket(),
    grpc.WithTransportCredentials(insecure.NewCredentials()),
    )
    // ...
    // ...
    // ...
    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)}
    return sig, nil
}
```

There is no length check in the reply's signature so `sigBytes` can be any size from 0 to 64MBs. `G1Point.Deserialize()` will call `bn254utils.DeserializeG1()` with whatever data is returned from the DA node.

```
func (p *G1Point) Deserialize(data []byte) *G1Point {
    return &G1Point{bn254utils.DeserializeG1(data)}
}
```

The following byte slice indexing in the `SetBytes()` arguments will panic if the signature is less than 64 bytes long:

```
func DeserializeG1(b []byte) *bn254.G1Affine {
    p := new(bn254.G1Affine)
    p.X.SetBytes(b[0:32])
    p.Y.SetBytes(b[32:64])
    return p
}
```

This will bring down the sequencer.

Recommendation: Sanitize incoming signatures or provide a length check in your bn254 deserialization methods.

5.1.3 Unauthenticated disperser clients can fill the global rate limit even if `unauthRates` are 0

Severity: Critical Risk

Context: disperser/apiserver/server.go#L230, disperser/apiserver/server.go#L389

Description: Unauthenticated disperser clients can fill the global rate limits `TotalUnauthThroughput` and `TotalUnauthBlobRate` even if the `unauthRates.PerUserUnauthThroughput` and `unauthRates.PerUserUnauthBlobRate` config values are set to zero. This will prevent authorized clients from being able to request blob dispersal.

Both `DisperseBlobAuthenticated` and `DisperseBlob` API calls eventually call `(s *DispersalServer) disperseBlob()` which collects and logs information about the authenticated address (if provided) as well as the origin IP. Blob request validation is then done but a malicious blob can easily be crafted that will pass these checks. `disperseBlob()` then calls `s.checkRateLimitsAndAddRates()` which is where the issues lies.

`s.getAccountRate()` will correctly return 0 for unauthorized accounts but before the request is refused the following global ratelimit checks are done:

```
// Check System Ratelimit
systemQuorumKey := fmt.Sprintf("%s:%d", systemAccountKey, param.QuorumID)
allowed, err := s.ratelimiter.AllowRequest(ctx, systemQuorumKey, encodedSize,
↪  rates.TotalUnauthThroughput)
if err != nil {
    return fmt.Errorf("ratelimiter error: %v", err)
}
if !allowed {
    s.logger.Warn("system byte ratelimit exceeded", "systemQuorumKey", systemQuorumKey, "rate",
    ↪  rates.TotalUnauthThroughput)
    return errSystemThroughputRateLimit
}

systemQuorumKey = fmt.Sprintf("%s:%d-blobrate", systemAccountKey, param.QuorumID)
allowed, err = s.ratelimiter.AllowRequest(ctx, systemQuorumKey, blobRateMultiplier,
↪  rates.TotalUnauthBlobRate)
if err != nil {
    return fmt.Errorf("ratelimiter error: %v", err)
}
if !allowed {
    s.logger.Warn("system blob ratelimit exceeded", "systemQuorumKey", systemQuorumKey, "rate",
    ↪  float32(rates.TotalUnauthBlobRate)/blobRateMultiplier)
    return errSystemBlobRateLimit
```

The issues is that in these calls to `s.ratelimiter.AllowRequest()` the `bucketParams.LastRequestTime` is updated, effectively counting against the global rate limit even though the request will ultimately be declined as unauthorized in the account rate limit checks.

```
func (d *rateLimiter) AllowRequest(ctx context.Context, requesterID common.RequesterID, blobSize uint,
↪  rate common.RateParam) (bool, error) {
    // Retrieve bucket params for the requester ID
    // This will be from dynamo for Disperser and from local storage for DA node
    bucketParams, err := d.bucketStore.GetItem(ctx, requesterID)
    // ...
    // ...
    // ...
    // Get interval since last request
    interval := time.Since(bucketParams.LastRequestTime)
    bucketParams.LastRequestTime = time.Now().UTC()
    // ...
    // ...
    // ...
    // Update the bucket based on blob size and current rate
    if allowed || d.globalRateParams.CountFailed {
        // Update bucket params
        err := d.bucketStore.UpdateItem(ctx, requesterID, bucketParams)
    if err != nil {
        return allowed, err
    }
    }
    // ...
    // ...
    // ...
```

This effectively make it possible for a unauthorized adversary to prevent authorized clients from dispersing blobs.

Recommendation:

1. Move the authorization check in `(s *DispersalServer) getAccountRate()` towards the top of the `DisperseBlobAuthenticated` and `DisperseBlob` request handlers to prevent authorized clients from having as much reachable attack surface as possible.

2. Enact origin-based rate limiting at this same location to prevent abuse of the challenge/response flow in `DisperseBlobAuthenticated` API calls.

5.1.4 `(s *DispersalServer) DisperseBlobAuthenticated()` vulnerable to stream exhaustion DOS

Severity: Critical Risk

Context: disperser/apiserver/server.go#L132, disperser/cmd/apiserver/main.go#L126

Description: The Disperser Server does not include a timeout in the authentication challenge flow. This makes it possible for a malicious actor to open up multiple streams that will tie up resources. It may be possible to hold the default max file descriptor count of a process (1024 in linux based systems) and prevent the disperser from being able to respond to new API requests.

In disperser/apiserver/server.go#L132 a grpc stream is used to communicate handle `DisperseBlobAuthenticated` requests. This stream appears to be used to enable bidirectional and stateful communication enabling a cryptographic challenge and public key verification. The issue is that second `Recv()` call on the stream will wait on the response from stream without a timeout.

```
challenge := rand.Uint32()
err = stream.Send(&pb.AuthenticatedReply{Payload: &pb.AuthenticatedReply_BlobAuthHeader{
        BlobAuthHeader: &pb.BlobAuthHeader{
                ChallengeParameter: challenge,
        },
}})
if err != nil {
        return err
}


// Receive challenge_reply
in, err = stream.Recv()
if err != nil {
        return fmt.Errorf("error receiving next message: %v", err)
}
```

Since the stream.Recv() method in gRPC Go does not have a built-in timeout, it requires specifically setting one using a context with a deadline eg. However, the context passed to this grpc server does not use a context with a timeout:

```
func RunDisperserServer(ctx *cli.Context) error {
    // ...
    // ...
    // ...
   return server.Start(context.Background())
}
```

Recommendation: Create a timeout for the grpc.stream for all disperser API handlers. This appears to be done correctly in the disperser client streams eg.:

```
disperserClient := disperser_rpc.NewDisperserClient(conn)
ctxTimeout, cancel := context.WithTimeout(ctx, c.config.Timeout)
defer cancel()
// ...
// ...
// ...
reply, err := disperserClient.DisperseBlob(ctxTimeout, request)
if err != nil {
    return nil, nil, err
}
```

5.1.5 `(s *DispersalServer) validateBlobRequest` contains externally triggerable data race

Severity: Critical Risk

Context: disperser/apiserver/server.go#L181

Description: In `(s *DispersalServer) validateBlobRequest` there are multiple reads of `s.quorumCount` without a lock. This is externally reachable via the DispersalServer API so a data race could bring down the disperser.

`s.quorumCount` is guarded by a mutex inside `s.updateQuorumCount()` which is called between these reads. Due to this the locks cannot wrapped around this function without causing a deadlock. Either those locks should be moved up into this function to broaden the critical section or a thread-unsafe version of "s.updateQuorumCount()' needs to be used here.

Recommendation: Refactor this function to lock on the reads and releases in question (before calling `s.updateQuorumCount()`) or make some other variation of this that guards the `s.quorumCount` shared data appropriately.

5.2 High Risk

5.2.1 `GetBlobHeaderFromProto` allows invalid points

Severity: High Risk

Context: node/grpc/utils.go#L65-L66, node/grpc/utils.go#L73-L76, node/grpc/utils.go#L79-L82

Description: In `GetBlobHeaderFromProto`, the components/limbs (`X`, `Y`) of the points (`Commitment`, `LengthCommitment`, and `LengthProof`) are manually set. Doing it this way will not check if the point is valid (on the curve, in the right subgroup, etc). This may actually impact pairing checks. Eventually, this point is casted to a `bn254.G1Affine` which won't check if the point is valid either.

Also, just a little nit, `h.GetLengthCommitment() != nil` and `h.GetLengthProof() != nil` checks are unnecessary as protobuf getter functions are guaranteed not to return nil.

Recommendation: Either of the following:

1. In `GetBlobHeaderFromProto`, cast the values to `bn254.G*Affine` and call `p.IsOnCurve()` and `p.IsInSubGroup()`.
2. Use compressed serialization and `bn254.G*Affine#SetBytes` which performs these checks.

5.2.2 `getBlobFromRequest()` vulnerable to resource consumption related DOS (CPU/memory/disk)

Severity: High Risk

Context: disperser/apiserver/server.go#L630

Description: `getBlobFromRequest()` makes allocations in a loop before checking the size of `Blob.data` or the number of `SecurityParams`. This info is checked later on as it is included in the various checks in `(s *DispersalServer) validateBlobRequest()` but this is after a second set of allocations is made for the data. Since there is no length check prior to here it is possible for a malicious node to force allocation of large amounts of blob data or a large number of `SecurityParams`, opening up a possible resource consumption related DOS vector.

The worst case resource usage is likely the number of `SecurityParams` that can fit in a 300MB protobuf message (server.go#L588), which will cause this function to iterate a `for` loop a significant # of times (~40MM at 8 bytes per `SecurityParam`):

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

This will also log the entirety of the security params values to disk in `(s *DispersalServer) disperseBlob()` right before the final rejection of the message in `validateBlobRequest()`:

```
s.logger.Debug("received a new disperse blob request", "origin", origin, "securityParams",
↪  strings.Join(securityParamsStrings, ", "))

err = s.validateBlobRequest(ctx, blob)
```

An attacker will not be rate limited here because their blob will be rejected after the damage is done but before the rate limit functionality is reached. They can likely fill a disk in a reasonable amount of time (eg. in less than a day you can easily fill a 1TB disk with the 300MB protobuff message limits.)

Assuming rate limits are enforced properly and `Debug` logging is off this may be handled decently by a well resourced machine, but the CPU usage and memory allocations can be entirely prevented with a `len(req.SecurityParams) <= 256` check here.

Recommendation: Add size checks for `Blob.data <= 2MB` and # `SecurityParams <= 256` before making their allocations or logging their values.

5.2.3 `validateChurnRequest` allows limitless, duplicate `QuorumIds`

Severity: High Risk

Context: operators/churner/server.go#L147-L162

Description: This function does not enforce a length limit and seems to allow duplicates, as long as they are valid. When logged in server.go#L69, it is possible to log ~300 MiB (the message size limit) worth of `uint32` values to the file/console. As a string, this could be very large and use up a lot of memory & disk space.

Recommendation: Check if the length of `QuorumIds` is greater than the max quorum count and check for duplicates.

5.2.4 Possible index out of range in `GetBlobMessages`

Severity: High Risk

Context: node/grpc/utils.go#L44

Description: If there are more `Bundles` than `QuorumHeaders`, this will panic with an index out of range error. This is only a high because it's part of the node, not the disperser.

Recommendation: Check that the length of these fields are equal before the loop.

5.2.5 Possible `nil` pointer dereference in `GetBatchHeader`

Severity: High Risk

Context: node/grpc/utils.go#L24

Description: In `GetBatchHeader`, when `in.BatchHeader.ReferenceBlockNumber` is accessed, `BatchHeader` could be nil.

Recommendation: Use protobuf getter functions, like `in.GetBatchHeader().GetReferenceBlockNumber()`.

5.2.6 Insufficiently random challenge in `BatchVerifyCommitEquivalence`

Severity: High Risk

Context: encoding/kzg/verifier/batch_commit_equivalence.go#L28-L47

Description: When generating `randomFr`, it is not enough to only hash the commitments. You must include everything that the verifier uses and the prover can influence.

Recommendation: Use `crypto/rand` for randomness instead of Fiat-Shamir.

5.2.7 Insufficiently random challenge in `UniversalVerify`

Severity: High Risk

Context: encoding/kzg/verifier/multiframe.go#L36-L48

Description: When generating `randomFr`, it is not enough to only hash the sample commitments. You must include everything that the verifier uses and the prover can influence. This includes the chunk length, sample data, commitments, proofs, indices, etc.

Recommendation: Use `crypto/rand` for randomness instead of Fiat-Shamir.

5.3 Medium Risk

5.3.1 `StoreChunksRequest` does not contain a signature field or other authentication root

Severity: Medium Risk

Context: node/grpc/server.go#L131

Description: `StoreChunksRequest` does not contain a signature and there is no way to attribute requests to the disperser, batcher, or any other node. This allows for servicing of invalid chunk storage requests. This can ultimately lead to a DOS of the DA node.

Recommendation: Add a disperser signature field to `StoreChunksRequest` messages.

5.3.2 `GetBlobHeader` is not rate limited allowing for DA Node DoS vector

Severity: Medium Risk

Context: node/grpc/server.go#L268

Description: The `node.GetBlobHeader` request is not rate limited in any way and contains multiple resource intensive operations:

- `(s *Server) getBlobHeader` retrieves blob headers from the node store database.
- `GetBlobHeaderHash()` creates `Keccak256` hashes (called multiple times in the request flow).
- `(s *Server) rebuildMerkleTree()` reads batch headers from the node store db and builds a merkle tree of the batch header hashes.
- `GenerateProof()` generates a merkle proof of the blob headers inclusion.

There is no origin based rate limiting and no caching of the requested values. This allows for infinite resource grieving of the DA node by unauthenticated adversaries.

Recommendation: Implement global and origin based rate limiting. When adding global rate limiting be careful that it cannot be triggered by a malicious peer to prevent valid requests from being serviced. Implement a `BlobHeader` cache so that the hashing, db lookups, merkle tree, and merkle proof generation do not have to happen on each request.

5.3.3 `InplaceFFT` will panic if given a zero-length slice

Severity: Medium Risk

Context: encoding/fft/fft_fr.go#L117, encoding/fft/fft_fr.go#L128

Description: If `InplaceFFT` is given a zero-length slice, it will panic with a divide by zero error. For example:

```
func TestInplaceFFT(t *testing.T) {
    fs := NewFFTSettings(4)
    err := fs.InplaceFFT([]fr.Element{}, []fr.Element{}, true)
    if err != nil {
        return
    }
}
```

```
--- FAIL: TestInplaceFFT (0.00s)
panic: runtime error: integer divide by zero [recovered]
    panic: runtime error: integer divide by zero
```

This is because `IsPowerOfTwo` returns `true` for 0. Technically, 0 is not a power of zero.

```
func IsPowerOfTwo(v uint64) bool {
    return v&(v-1) == 0
}
```

When decoding chunks, `GetInterpolationPolyEval` uses `InplaceFFT` which means it will panic if given a frame with zero coefficients. I'm unsure if it's possible to provide such a frame, but this should be fixed just in case.

```
func TestDecodeEmptyFrames(t *testing.T) {
    _, testEncodingParams := getTestData()
    var chunksData [32]*encoding.Frame
    for i := 0; i < 32; i++ {
        // Here len(frame.Coeffs) is 0
        chunksData[i] = new(encoding.Frame)
    }
    indices := []encoding.ChunkNumber{0, 1, 2, 3, 4, 5, 6, 7}
    maxInputSize := uint64(len(gettysburgAddressBytes)) + 10
    testProver.Decode(chunksData[:], indices, testEncodingParams, maxInputSize)
}
```

```
--- FAIL: TestDecodeEmptyFrames (0.00s)
panic: runtime error: integer divide by zero [recovered]
    panic: runtime error: integer divide by zero
```

Recommendation: Fix `IsPowerOfTwo` so that it returns false for 0.

5.3.4 `fp.Element#SetBytes` is used which allows values greater than the modulus

Severity: Medium Risk

Context: attestation.go#L111-L112, attestation.go#L139-L142, encoding/kzg/kzg.go#L65, node/grpc/utils.go#L65-L66, node/grpc/utils.go#L73-L76, node/grpc/utils.go#L79-L82

Description: `fp.Element#SetBytes` does not ensure the given value is less that or equal to the modulus (though it will if there are 33+ bytes though, but that's irrelevant). To check for this, you should use `SetBytesCanonical` instead. Here `q` is the modulus.

```go
// SetBytesCanonical interprets e as the bytes of a big-endian 32-byte integer.
// If e is not a 32-byte slice or encodes a value higher than q,
// SetBytesCanonical returns an error.
func (z *Element) SetBytesCanonical(e []byte) error {
    if len(e) != Bytes {
        return errors.New( text: "invalid fp.Element encoding")
    }
    v, err := BigEndian.Element((*[Bytes]byte)(e))
    if err != nil {
        return err
    }
    *z = v
    return nil
}
```

Note: the usage in `ToFrArray` is safe because it limits the number of bytes to 31.

This is classified as a medium risk because the point deserialization functions already allow invalid points and the other instances, like in `MapToCurve`, are currently unused.

Recommendation: Where necessary, use `fp.Element#SetBytesCanonical` instead.

5.3.5 `GetBlobHeaderFromProto` will cast and accept large quorum info values

Severity: Medium Risk

Context: node/grpc/utils.go#L89-L91, node/store.go#L229, node/grpc/server.go#L232

Description: `server.StoreChunks` will accept blobs where `BlobHeader.BlobQuorumInfo` values (`QuorumID`, `AdversaryThreshold`, and `ConfirmationThreshold`) are greater than `uint8::max` and the casted values pass the right checks. The protobuf-encoded header is what is written to the database. And in `server.GetBlobHeader`, `protoBlobHeader` is returned. If there's ever another implementation of a retrieval client or the max QourumID value changes, this could cause problems.

Recommendation: Return an error in `GetBlobHeaderFromProto` if these values are greater than `uint8::max`, or change these fields' types from `uint32` to `uint8`.

5.3.6 `DisperseBlobAuthenticated` Authentication flow is not rate limited

Severity: Medium Risk

Context: core/auth/authenticator.go#L31, core/auth/signer.go#L36

Description: The `DisperseBlobAuthenticated` API request authentication challenge response does not have origin based rate limiting, opening up a potential DOS vector.

`(s *DispersalServer) DisperseBlobAuthenticated()` challenges an unauthorized client by issuing a `rand.Uint32()` to hash and sign with their private key. The issuance of the random challenge, the hashing of the challenge, the verification of the returned signature, and the lookup of the various rate limit values have non-negligible resource usage. It is currently possible for a malicious actor to continuously call this API without being rate limited. The only rate limiting that happens is upon successful dispersion. Rate limiting this call by origin and immediately at the attempt of the call would keep malicious actors from using this as a DOS vector.

Implementing some type of scoring to timeout bad clients (eg. more than 3 signature verification failures) would also add value.

Recommendation: Add origin based rate limiting to this API call, incremented even when there is a signature verification failure.

5.3.7 No rate limiting for `(s *DispersalServer) RetrieveBlob()` API requests

Severity: Medium Risk

Context: disperser/apiserver/server.go#L530

Description: `(s *DispersalServer) RetrieveBlob()` does not enforce rate limiting. This request holds mutexes on blob metadata, will cause error logging when requests fail, and makes database lookups. Grieving these shared resources could be detrimental to proper disperser operation.

Recommendation: Implement rate limiting for the `RetrieveBlob` API call.

5.3.8 No rate limiting for `(s *DispersalServer) GetBlobStatus` API requests

Severity: Medium Risk

Context: disperser/apiserver/server.go#L530

Description: `(s *DispersalServer) GetBlobStatus` does not enforce rate limiting. This request holds mutexes on blob metadata and will cause error logging when requests fail, filling up disk space. Grieving these shared resources could be detrimental to proper disperser operation.

Recommendation: Implement rate limiting for the `RetrieveBlob` API call.

5.3.9 `(e *encodedBlobStore) DeleteEncodingRequest()` needs a write lock

Severity: Medium Risk

Context: disperser/batcher/encoded_blob_store.go#L83

Description: `(e *encodedBlobStore) DeleteEncodingRequest()` holds a read lock on `e.mu` which is is guarding `e.requested`. This is not sufficient because `e.requested` is modified with a `delete()` in this function. This is called regularly by `(e *EncodingStreamer) ProcessEncodedBlobs` when the `EncodingResultOrStatus chan` receives encoding requests. A data race here threatens the liveness of the disperser.

Recommendation: Change this functions `Rlock()` to a `Lock()`.

5.3.10 `CalculateRequestHash` does not include `QuorumIDs`

Severity: Medium Risk

Context: operators/churner/churner.go#L312-L318

Description: This function hashes everything except `churnRequest.QuorumIDs`. If it's possible to intercept this somehow, an intercepted request could provide a different `QuorumIDs` value.

Recommendation: Include the `QuorumIDs` field in the hash.

5.3.11 Resource leak due to `defer` called in loop in `monitorTransaction`

Severity: Medium Risk

Context: disperser/batcher/txn_manager.go#L165-L167

Description: If this loop runs many iterations before returning, you'll end up stacking many `defer` calls, which will only be executed when the function exits. This can lead to a temporary increase in memory usage since the contexts created by `context.WithTimeout` won't be cancelled (and thus, cleaned up) until the function returns.

This is a medium because the maximum number of iterations could be large.

Recommendation: Reimplement function so that it doesn't need to `defer` in a loop.

5.3.12 Possible integer overflow allows invalid `SecurityParams`

Severity: Medium Risk

Context: core/data.go#L80

Description: When checking `QuorumThreshold` and `AdversaryThreshold`, there's an unsafe addition which could overflow and be abused. These are both `uint8` types, so it is easy to overflow. For example, if `QuorumThreshold` is 5 and `AdversaryThreshold` is 255, this will pass the check and an invalid situation will be allowed.

Recommendation: Add another check to see if the overflow is possible.

5.4 Low Risk

5.4.1 `roundUpDivide()` will panic with `panic: runtime error: integer divide by zero` when `b == 0`

Severity: Low Risk

Context: encoding/utils.go#L30

Description/Recommendation: `roundUpDivide()` will panic with `panic: runtime error: integer divide by zero` when `b == 0`:

```
func roundUpDivide[T constraints.Integer](a, b T) T {
    return (a + b - 1) / b
}
```

This function is reachable via `GetEncodedBlobLength()` when `quorumThreshold-advThreshold == 0`.

5.4.2 `bitmapToBytesArray()` will panic on malformed input

Severity: Low Risk

Context: core/eth/tx.go#L795

Description: This is a similar issue to the finding in `BitmapToQuorumIds()`. `bitmapToBytesArray()` will `panic: underflow` when given `-0` as the `big.Int` to return a bitmap for. This happens when `bitmap.Bit()` is called:

```
func bitmapToBytesArray(bitmap *big.Int) []byte {
    // initialize an empty uint64 to be used as a bitmask inside the loop
    var (
        bytesArray []byte
    )
    // loop through each index in the bitmap to construct the array
    for i := 0; i < maxNumberOfQuorums; i++ {
        // check if the i-th bit is flipped in the bitmap
        if bitmap.Bit(i) == 1 {
            // if the i-th bit is flipped, then add a byte encoding the value 'i' to the `bytesArray`
            bytesArray = append(bytesArray, byte(uint8(i)))
        }
    }
    return bytesArray
}
```

The panic happens in /usr/local/go/src/math/big/nat.go in `(z nat) sub()`. The issue here is that there is no way to create a valid `-0 big.Int` in regular practice, but deserialization from protobuf or `github.com/ethereum/go-ethereum/accounts/abi.ConvertType()` takes in raw byte data and assigns values to underlying struct fields. This allows someone craft a malicious `Big.Int` like this:

```
// here is what a big.Int looks like in its struct definition
type Int struct {
    neg bool // sign
    abs nat  // absolute value of the integer
}

// here is a malicious big.Int that is set to -0
// an attacker would have to make special code to set these
// fields directly before sending or would need to send
// a raw byte array with this payload
var malicious_bitmap big.Int
bitmap.neg = true
bitmap.abs = 0

// this call would trigger the panic: underflow
bitmap.bit(1)
```

Recommendation: There is no way to check the subfields directly since they are private so it is best to either add `recover()` handlers to handle this situation or add an explicit check like this:

```
// it is impossible for a big int to be less than 0 AND greater than -1
if bitmap.Cmp(big.NewInt(0)) == -1 && bitmap.Cmp(big.NewInt(-1)) == 1 {
    fmt.Println("Invalid bitmap")
    // error in some way
}
```

5.4.3 `WriteBatch` should check that `keys` and `values` are the same length

Severity: Low Risk

Context: node/leveldb/leveldb.go#L57

Description: As the name suggests, this function write the keys/values to the database. It iterates over the keys and indexes into the values. If there are more keys than values, this will panic.

Recommendation: The caller (`StoreBatch`) handles this properly and this situation should never happen, but out of an abundance of caution, we should add a simple check which returns an error if they aren't the same length.

5.4.4 Incorrect usage of `break` in `decodeChunks`

Severity: Low Risk

Context: node/store.go#L364-L370

Description: Rather than `break`, we should `return nil, err` here. If there are extra bytes at the end, this would be a problem.

Recommendation: Replace `break` with `return nil, err`.

5.4.5 Resource leak due to `defer` called in loop in `getLatestFinalizedBlock`

Severity: Low Risk

Context: disperser/batcher/finalizer.go#L238-L240

Description: If this loop runs many iterations before returning, you'll end up stacking many `defer` calls, which will only be executed when the function exits. This can lead to a temporary increase in memory usage since the contexts created by `context.WithTimeout` won't be cancelled (and thus, cleaned up) until the function returns.

This is a low because the maximum number of iterations is small.

Recommendation: Reimplement function so that it doesn't need to `defer` in a loop.

5.4.6 Resource leak due to `defer` called in loop in `getTransactionBlockNumber`

Severity: Low Risk

Context: disperser/batcher/finalizer.go#L209-L211

Description: If this loop runs many iterations before returning, you'll end up stacking many `defer` calls, which will only be executed when the function exits. This can lead to a temporary increase in memory usage since the contexts created by `context.WithTimeout` won't be cancelled (and thus, cleaned up) until the function returns.

This is a low because the maximum number of iterations is small.

Recommendation: Reimplement function so that it doesn't need to `defer` in a loop.

5.4.7 Missing `json` tag in `BlobHeader` structure

Severity: Low Risk

Context: core/data.go#L101-L108

Description: The `QuorumInfos` field is missing a `json` tag.

Recommendation: Add a `json` tag to `QuorumInfos` with the snake case name.

5.4.8 Incorrect bounds check in `Read*Point` functions

Severity: Low Risk

Context: encoding/kzg/pointsIO.go#L43

Description: If `n` is an index, it should reject values equal to `g.SRSOrder`.

Recommendation: Change > to >= in condition.

5.4.9 Inconsistent use of `Length` and `Degree`

Severity: Low Risk

Context: encoding/data.go#L26-L28

Description: There is mixed terminology in the codebase concerning `Length` and `Degree`. We found it confusing that `VerifyBlobLength` is actually verifying the degree, or number of elements in the blob there are. For example, some files refer to this as the `lowDegreeProof` and other files refer to it as `LengthProof`. We suspect others might be confused too and accidentally make a mistake somewhere.

Recommendation: Be consistent and rename fields to `DegreeCommitment`, `DegreeProof`, and `Degree`.

5.4.10 Ignored error in `PreloadAllEncoders`

Severity: Low Risk

Context: encoding/kzg/prover/prover.go#L116

Description: If for some reason `GetAllPrecomputedSrsMap` fails, it will return `nil` instead of the error.

Recommendation: Return `err` instead of `nil`.

5.5 Optimizations

5.5.1 `AuthenticateBlobRequest()` signature checks should come before other logic

Severity: Optimization

Context: core/auth/authenticator.go#L48

Description: `(*authenticator) AuthenticateBlobRequest()` has simple checks for `nil` and incorrect length signatures. These checks should be moved to the top of the function to for optimization reasons and to prevent as much unauthenticated attack surface as possible. Eg. hashing the response, decoding the `AccountID` and `Nonce`, and Unmarshaling the publicKey bytes are operations that are not needed if the signature is `nil` or incorrectly sized

Recommendation: Move the fastest sanity checks to the top of the function.

5.5.2 Use fixed-base MSM for faster proof generation

Severity: Optimization

Context: encoding/kzg/prover/parametrized_prover.go#L175

Description: Here it is possible to use a fixed-base MSM rather than a variable-base MSM. It would require a precomputed table be stored in memory, but it could potentially reduce the time it takes to generate proofs by 2x. This will be an expensive operation, so any performance gains here would be important.

Recommendation: Investigate if gnark provides a function for this. If they do not, ask them to make one.

5.5.3 Calculating `leadingDs` is overly verbose and inefficient

Severity: Optimization

Context: encoding/kzg/verifier/multiframe.go#L148-L158

Description: This block of code, which raises `h` to the power of `D`, is overly verbose and inefficient.

Recommendation: Use the following implementation instead which uses gnark's `Exp` function.

```
h := ks.ExpandedRootsOfUnity[samples[k].X]
leadingDs[k].Exp(h, big.NewInt(int64(D)))
```

5.5.4 `ToFrArray` will always use the slow path

Severity: Optimization

Context: encoding/rs/utils.go#L24-L27

Description: Because `BYTES_PER_COEFFICIENT` is 31, calls to gnark's `Element.SetBytes` will always use the slow path.

Recommendation: Use 32-byte slice where possible.

5.6 Informational

5.6.1 Verifier wrapper functions should send result directly to channel

Severity: Informational

Context: core/validator.go#L184-L203

Description: These two functions can be simplified by directly sending the result to the channel. Also, I don't believe `VerifyBlobLengthWorker` needs to be exported.

Recommendation: Make the following changes:

```
  func (v *shardValidator) universalVerifyWorker(params encoding.EncodingParams, subBatch
  ↪  *encoding.SubBatch, out chan error) {
-
-      err := v.verifier.UniversalVerifySubBatch(params, subBatch.Samples, subBatch.NumBlobs)
-      if err != nil {
-          out <- err
-          return
-      }
-
-      out <- nil
+      out <- v.verifier.UniversalVerifySubBatch(params, subBatch.Samples, subBatch.NumBlobs)
  }

  func (v *shardValidator) VerifyBlobLengthWorker(blobCommitments encoding.BlobCommitments, out chan
  ↪  error) {
-     err := v.verifier.VerifyBlobLength(blobCommitments)
-     if err != nil {
-         out <- err
-         return
-     }
-
-     out <- nil
+     out <- v.verifier.VerifyBlobLength(blobCommitments)
  }
```

And change `VerifyBlobLengthWorker` to `verifyBlobLengthWorker`.

5.6.2 Incorrect function names in comments

Severity: Informational

Context: encoding/encoding.go#L10, encoding/encoding.go#L19, encoding/encoding.go#L22, encoding/encoding.go#L28, (*and probably a lot of other places*)

Description: The function name in the comment does not match the actual function name. It was most likely renamed and not updated.

Recommendation: Update these comments.

5.6.3 In `StoreBatch`, `size` is computed but not used

Severity: Informational

Context: node/store.go#L256

Description: The sum of chunk bytes (`size`) is computed, but it's never used anywhere.

Recommendation: Either remove this computation or share it via a debug log.

5.6.4 Should use `%w` instead of `%v` to print errors with `fmt.Errorf`

Severity: Informational

Context: node/config.go#L106, node/config.go#L123

Description: There are two instances of non-wrapping errors. It's good practice to wrap these with `%w`.

Recommendation: Make the following changes respective to the context:

```
- return nil, fmt.Errorf("could not read ECDSA key file: %v", err)
+ return nil, fmt.Errorf("could not read ECDSA key file: %w", err)
```

```
- return nil, fmt.Errorf("could not read or decrypt the BLS private key: %v", err)
+ return nil, fmt.Errorf("could not read or decrypt the BLS private key: %w", err)
```

5.6.5 Some arrays can be unsliced

Severity: Informational

Context: node/operator.go#L78, node/operator.go#L160, node/operator.go#L180, node/utils.go#L74

Description: There are a few instances of arrays which are sliced (like `arr[:]`) which do not require the `[:]` part.

Recommendation: Make the following changes respectively to the context:

```
- salt := crypto.Keccak256([]byte("churn"), []byte(time.Now().String()), QuorumIDs[:], privateKeyBytes)
+ salt := crypto.Keccak256([]byte("churn"), []byte(time.Now().String()), QuorumIDs, privateKeyBytes)
```

```
- copy(salt[:], crypto.Keccak256([]byte("churn"), []byte(time.Now().String()), operator.QuorumIDs[:],
↪  privateKeyBytes))
+ copy(salt[:], crypto.Keccak256([]byte("churn"), []byte(time.Now().String()), operator.QuorumIDs,
↪  privateKeyBytes))
```

```
- Salt:                      salt[:],
+ Salt:                      salt,
```

```
- buf := bytes.NewBuffer(append(prefix, ts[:]// ...))
+ buf := bytes.NewBuffer(append(prefix, ts// ...))
```

5.6.6 Function parameter names are capitalized

Severity: Informational

Context: node/operator.go#L98, node/operator.go#L156

Description: Two instances of `KeyPair *core.KeyPair` are capitalized when they should be lowercase.

Recommendation: Rename `KeyPair` to `keyPair`.

5.6.7 `*server.getBlob()` missing lock

Severity: Informational

Context: disperser/dataapi/blobs_handlers.go#L11

Description: *NOTE: This has been reduced to informational since `disperser/common/inmem` is deprecated and `disperser/common/blobstore` is being used instead.* `*server.getBlob()` calls `s.blobstore.GetBlobMetadata()` without a lock on `s.blobstore.mu`. This appears to be reachable externally via a http request. A nicely timed query here can cause a data race and a panic. Even if panics are handled gracefully with recover() race conditions are one of the few ways to cause memory corruption in Go. This can be abused as a remote DOS to the disperser.

Recommendation: Either use a lock when calling `s.blobstore.GetBlobMetadata()` or make a thread safe accessor for the function.

5.6.8 `"crypto/elliptic".Unmarshal()` is deprecated

Severity: Informational

Context: core/auth/authenticator.go#L42, disperser/apiserver/server.go#L113

Description: `"crypto/elliptic"` has deprecated `Unmarshal()` with the note:

> for ECDH, use the `crypto/ecdh` package. This function accepts an encoding equivalent to that of the `NewPublicKey` methods in `crypto/ecdh`.

The Geth team has been notified to update this but I am leaving a note here to make sure the updates are applied to EigenDA as well.

Recommendation: Either manually fix or wait for the next geth release and merge in their changes.

5.6.9 `(s *DispersalServer) disperseBlob()` ambiguous log message

Severity: Informational

Context: disperser/apiserver/server.go#L253

Description: `(s *DispersalServer) disperseBlob()` contains the error message: "received a new blob request". This could be confused in the logs with other similar API calls. It might be better to log "received a new disperse blob request."

Recommendation: Add some clarifying text to the logging in this function.

5.6.10 `BlobAuthHeader` explanation clarification

Severity: Informational

Context: api/docs/disperser.md#L163, api/grpc/disperser/disperser.pb.go#L253, api/proto/disperser/disperser.proto#L56

Description: There is a typo in the `BlobAuthHeader` explanation. It reads:

> Once payments are enabled, the BlobAuthHeader the KZG commitment to the blob, which the client will verify and sign.

It should probably read something like:

> Once payments are enabled, the BlobAuthHeader *will contain* the KZG commitment to the blob, which the client will verify and sign.

Recommendation: Add in the correct information.

5.6.11 Exported functions with unexported return types

Severity: Informational

Context: retriever/eth/chain_client.go#L24

Description: There are several instances of this, but the return type of some `New*` functions return an unexported type.

Recommendation: For example, make this change:

```
- func NewChainClient(ethClient common.EthClient, logger common.Logger) *chainClient {
+ func NewChainClient(ethClient common.EthClient, logger common.Logger) ChainClient {
```

5.6.12 `logger.Fatalf` with `%w` in `serveRetrieval`

Severity: Informational

Context: node/grpc/server.go#L109

Description: The `%w` verb is specifically designed for use with `fmt.Errorf` to create a new error that wraps another error.

Recommendation: Replace `%w` with `%v` in the format string.

5.6.13 `logger.Fatalf` with `%w` in `serveDispersal`

Severity: Informational

Context: node/grpc/server.go#L85

Description: The `%w` verb is specifically designed for use with `fmt.Errorf` to create a new error that wraps another error.

Recommendation: Replace `%w` with `%v` in the format string.

5.6.14 Reimplementation of serialization functions

Severity: Informational

Context: core/bn254/attestation.go#L96-L107

Description: These functions are unnecessarily complex. They can be replaced with function gnark provides.

Recommendation: Use `p.RawBytes()` to simplify the implementation.

5.6.15 Unnecessary `nil` check

Severity: Informational

Context: core/auth/authenticator.go#L49

Description: Because this is a slice, the `nil` check is unnecessary. The length check is sufficient.

Recommendation: Remove the `nil` check.