Layr Labs

EigenDA Proxy

Security Assessment Report

Version: 2.0

January, 2025

Contents

Introduction

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

Disclaimer

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

Document Structure

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

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

Overview

EigenDA is a Data Availability (DA) protocol that can be used to temporarily store data. This review focuses on the EigenDA Proxy, a HTTP sidecar server which aids actors in the EigenDA network to communicate more effectively.

The EigenDA Proxy is intended to sit between the Sequencer, Verifier Nodes and EigenDA Disperser. It then performs validation tasks to eliminate trust assumptions on the EigenDA Disperser as well as providing read fallback services and optional performance optimizations such as caching blobs.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to files at commit 9e1b746.

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

Approach

For the Golang libraries and modules, the review focused on internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime. The manual review explored known Golang antipatterns such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks and a multitude of panics including but not limited to `nil` pointer deferences, index out of bounds and calls to `panic()`.

To support this 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`

Furthermore, the testing team leveraged fuzz testing techniques (i.e. fuzzing), which is a process allowing the identification of bugs by providing randomised and unexpected data inputs to software with the purpose of causing crashes (in Golang, *panics*) and other unexpected behaviours (e.g. broken invariants, memory exhaustion, infinite / extended loops).

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 7 issues during this assessment. Categorised by their severity:

- Low: 3 issues.
- Informational: 4 issues.

Detailed Findings

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

A number of additional properties of the contracts, which do not cause direct security concerns, 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
EDAP-01	Private Key And Data Stored Unencrypted	Low	Closed
EDAP-02	No Limit To Input Blob Size	Low	Resolved
EDAP-03	Missing `nil` Checks On Parameters Of Incoming Requests	Low	Resolved
EDAP-04	Lack Of Transport Layer Encryption	Informational	Closed
EDAP-05	Missing `IsOnCurve` & `IsInSubgroup` Checks For Elliptic Curve Points	Informational	Resolved
EDAP-06	TODOs In Code	Informational	Closed
EDAP-07	Miscellaneous General Comments	Informational	Resolved

EDAP-01	Private Key And Data Stored Unencrypted		
Asset	`cmd/server/entrypoint.go, server/load_store.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The Proxy signer's private key, API endpoint key and Amazon S3 Endpoint key are stored in cleartext.

The `entrypoint.go::StartProxySvr()` function loads the private key and API endpoint key from a file before sanitising them for logging purposes. There is no functionality decrypting these keys after loading them, suggesting that the proxy signer private key and API endpoint key are stored in cleartext.

```
cfg := server.ReadCLIConfig(cliCtx)
  if cfg.EigenDAConfig.EdaClientConfig.SignerPrivateKeyHex != "" {
    cfg.EigenDAConfig.EdaClientConfig.SignerPrivateKeyHex = "*****" // marshaling defined in client config
  }
  if cfg.EigenDAConfig.EdaClientConfig.EthRpcUrl != "" {
    cfg.EigenDAConfig.EdaClientConfig.EthRpcUrl = "*****" // hiding as RPC providers typically use sensitive API keys within
  }
```

Likewise, in `load_store.go::LoadStoreManager()`, the function loads the Amazon S3 and Redis service endpoint keys from a file and there is no functionality decrypting these keys after loading them. This suggests the API endpoints are stored in cleartext.

```
if cfg.EigenDAConfig.StorageConfig.S3Config.Bucket != "" && cfg.EigenDAConfig.StorageConfig.S3Config.Endpoint != "" {
    log.Info("Using S3 backend")
    s3Store, err = s3.NewStore(cfg.EigenDAConfig.StorageConfig.S3Config)
    if err != nil {
      return nil, fmt.Errorf("failed to create S3 store: %w", err)
    }
  }

  if cfg.EigenDAConfig.StorageConfig.RedisConfig.Endpoint != "" {
    log.Info("Using Redis backend")
    // create Redis backend store
    redisStore, err = redis.NewStore(&cfg.EigenDAConfig.StorageConfig.RedisConfig)
    if err != nil {
      return nil, fmt.Errorf("failed to create Redis store: %w", err)
    }
  }
```

Recommendations

Avoid storing sensitive information, such as private keys, in cleartext. Where possible implement password encryption for sensitive data at rest.

Additionally, ensure all files have strict permissions that restrict reading of the file. In Linux that would be `700` permissions.

Resolution

The development team have opted not to resolve the issue, due to the low likelihood of the issue that an attacker is required to have access to the machine in order to exploit this.

EDAP-02	No Limit To Input Blob Size		
Asset	`server/handlers.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

An unbounded read occurs when handling `PUT` requests, leading to a resource exhaustion denial of service (DoS) condition.

When the user submits blob data using `Put()`, there is no limit placed on the size of the HTTP body, this data is read via `io.ReadAll()` and is then decoded via `e.client.GetCodec().EncodeBlob(value)`.

```
func (svr *Server) handlePostShared(w http.ResponseWriter, r *http.Request, comm []byte, meta commitments.CommitmentMeta) error {
    svr.log.Info("Processing POST request", "commitment", hex.EncodeToString(comm), "commitmentMeta", meta)
    input, err := io.ReadAll(r.Body) // @audit unbounded read
    if err != nil {
        err = MetaError{
            Err:  fmt.Errorf("failed to read request body: %w", err),
            Meta: meta,
        }
        http.Error(w, err.Error(), http.StatusBadRequest)
        return err
    }
```

The likelihood of this issue is rated as low, as exploiting unbounded reads to exhaust resources is non-trivial to exploit. Numerous factors come into play such as network width, CPU and memory capabilities which reduce the ease of exploitation of this specific vulnerability.

Recommendations

EigenDA compliant blobs are recorded as having a maximum length in accordance with the following equation, `MaxAllowedBlobSize = uint64(MaxSRSPoints * BytesPerSymbol / MaxCodingRatio)` from `verify/cli.go`. Therefore, check that user supplied blobs do not exceed this, allowing for a buffer for encoding inefficiencies.

This may be achieved by using a bounded read as seen in the following code snippet.

```
    r.Body = http.MaxBytesReader(w, r.Body, MaxAllowedEncodedBlobSize) // @audit determine max size when encoded

    body, err := io.ReadAll(r.Body)
    if err != nil {
        http.Error(w, "Request body too large", http.StatusRequestEntityTooLarge)
        return
    }
    defer r.Body.Close()
}
```

Resolution

The recommendation has been implemented in PR #277.

EDAP-03	Missing `nil` Checks On Parameters Of Incoming Requests		
Asset	`store/generated_key/eigenda/eigenda.go, verify/verifier.go, verify/cert.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

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

The following is a list of objects which do not validate that pointers are non `nil`. If fields of these objects are accessed when they are `nil`, the EigenDA proxy will crash.

In the following code from `eigenda.go` in `Get()`, line [62] will result in `nil` pointer dereference if `cert.blobVerificationProof` is `nil`. Furthermore, the issue also occurs if other fields of `cert` are `nil`. This `nil` pointer panic is reachable for the HTTP `GET` requests allowing any client to cause the server to crash.

store/generated_key/eigenda/eigenda.go

```
53 // Get fetches a blob from DA using certificate fields and verifies blob
   // against commitment to ensure data is valid and non-tampered.
55 func (e Store) Get(ctx context.Context, key []byte) ([]byte, error) {
     var cert verify.Certificate
57   err := rlp.DecodeBytes(key, &cert)
     if err != nil {
59     return nil, fmt.Errorf("failed to decode DA cert to RLP format: %w", err)
     }
61
     decodedBlob, err := e.client.GetBlob(ctx, cert.BlobVerificationProof.BatchMetadata.BatchHeaderHash,
         ↪  cert.BlobVerificationProof.BlobIndex) // @audit this could nil reference if BlobVerificationProof is nil
```

Also in `eigenda.go`, for the `Put()` function, if `blobInfo` is `nil` then `cert` will also be `nil` and attempting to access its fields on line [122] will panic.

store/generated_key/eigenda/eigenda.go

```
// Put disperses a blob for some pre-image and returns the associated RLP encoded certificate commit.
func (e Store) Put(ctx context.Context, value []byte) ([]byte, error) {
  encodedBlob, err := e.client.GetCodec().EncodeBlob(value)

  // ... snipped for brevity

  // We attempt to disperse the blob to EigenDA up to 3 times, unless we get a 400 error on any attempt.
  blobInfo, err := retry.DoWithData(
    func() (*disperser.BlobInfo, error) {
      return e.client.PutBlob(ctx, value)
    },

    // ... snipped for brevity

  }
  cert := (*verify.Certificate)(blobInfo)

  err = e.verifier.VerifyCommitment(cert.BlobHeader.Commitment, encodedBlob) // @audit nil pointer if cert.BlobHeader is nil
```

In `verifier.go`, if the parameter `batchHeader` for function `verifySecurityParams()` is `nil`, then [153-155] could cause a panic.

verify/verifier.go

```
148 // verifySecurityParams ensures that returned security parameters are valid
    func (v *Verifier) verifySecurityParams(blobHeader BlobHeader, batchHeader *disperser.BatchHeader) error {
150   confirmedQuorums := make(map[uint8]bool)

152   // require that the security param in each blob is met
      for i := 0; i < len(blobHeader.QuorumBlobParams); i++ {
154     if batchHeader.QuorumNumbers[i] != blobHeader.QuorumBlobParams[i].QuorumNumber {
          return fmt.Errorf("quorum number mismatch, expected: %d, got: %d", batchHeader.QuorumNumbers[i],
              ↪ blobHeader.QuorumBlobParams[i].QuorumNumber)
156     }
```

Note that there are numerous similar panics caused throughout this function for fields of `batchHeader`.

Furthermore, in `verifier.go`, the function `verifyCert()` may panic if `cert.Proof` or `batchMetadata` is `nil` on line [82] and line [94]. Additionally, if either `cert` or `cert.Proof()` are `nil` when calling `VerifyCert()` then line [82], line [88] or [94-95] will cause a panic. The testing team notes that no valid route to product a `nil` `cert` was found.

verify/verifier.go

```
75  // verifies V0 eigenda certificate type
    func (v *Verifier) VerifyCert(ctx context.Context, cert *Certificate) error {
77    if !v.verifyCerts {
        return nil
79    }

81    // 1 - verify batch in the cert is confirmed onchain
      err := v.cv.verifyBatchConfirmedOnChain(ctx, cert.Proof().GetBatchId(), cert.Proof().GetBatchMetadata()) // @audit if
          ↪ batchMetadata is nil then verifyBatchConfirmedOnChain() will panic
83    if err != nil {
        return fmt.Errorf("failed to verify batch: %w", err)
85    }

87    // 2 - verify merkle inclusion proof
      err = v.cv.verifyMerkleProof(cert.Proof().GetInclusionProof(), cert.BatchHeaderRoot(), cert.Proof().GetBlobIndex(),
          ↪ cert.ReadBlobHeader())
89    if err != nil {
        return fmt.Errorf("failed to verify merkle proof: %w", err)
91    }

93    // 3 - verify security parameters
      batchHeader := cert.Proof().GetBatchMetadata().GetBatchHeader() // @audit if batchMetadata is nil then GetBatchHeader() will
          ↪ panic
95    err = v.verifySecurityParams(cert.ReadBlobHeader(), batchHeader)
      if err != nil {
97      return fmt.Errorf("failed to verify security parameters: %w", err)
      }
99
      return nil
101 }
```

In `verify/cert.go` on line [93] if `GetBatchRoot()` returns `nil`, this will also cause a panic when cast into a byte array.

verify/cert.go

```
91  // 3. Compute the hash of the batch metadata received as argument.
    header := &binding.IEigenDAServiceManagerBatchHeader{
93    BlobHeadersRoot:       [32]byte(batchMetadata.GetBatchHeader().GetBatchRoot()), // @audit GetBatchRoot() can return nil and would
          ↪ cause a panic
      QuorumNumbers:         batchMetadata.GetBatchHeader().GetQuorumNumbers(),
95    ReferenceBlockNumber:  batchMetadata.GetBatchHeader().GetReferenceBlockNumber(),
      SignedStakeForQuorums: batchMetadata.GetBatchHeader().GetQuorumSignedPercentages(),
97  }
```

The overall impact is rated as low as a `nil` pointer exception would trigger a panic, however the HTTP Server will recover from the panic. Therefore, the server will continue to operate and handle future requests.

The likelihood of this occurring is rated as medium as any client with access to the `/get` endpoints may trigger this panic.

Recommendations

Prevent the associated pointer types from being `nil`. It is recommended to check for `nil` either immediately after decoding or during the decoding process, along with any functions which take untrusted input.

Furthermore, for all GRPC generated types, do not directly deference pointers using the `.` syntax. Instead, call the associated getter functions, such as `GetBatchHeader()`, which perform `nil` pointer checks. Noting that these functions may still return `nil`.

Resolution

The issue has been resolved in PR #231. Checks have been added such that each certificate is validated that all fields are non `nil` after decoding.

EDAP-04	Lack Of Transport Layer Encryption
Asset	`client/client.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The default client for interacting with the EigenDA Proxy does not utilise any transport layer encryption such as TLS, exposing sensitive data transmitted between client and the proxy server to interception and manipulation. Without TLS, communications occur over plaintext protocols, leaving them susceptible to eavesdropping and man-in-the-middle attacks.

The client is used for performing `Get()` and `Put()` requests meaning that, without authentication, anyone could intercept or alter data requests to and from the EigenDA proxy. This could lead to degraded performance of client systems reliant on the EigenDA proxy for publishing blobs to the Ethereum mainnet. This client implementation is, however, only a reference client, and it is noted that Optimism makes use of their own client and users can alter the client code themselves.

These endpoints are not intended to be publicly exposed and so have a low likelihood of being targeted by malicious actors.

In addition to this, the EigenDA proxy server has a config option to disable TLS. While TLS is enabled by default, it is worth noting that it can be disabled using this option and doing so would widen the risk of attacks due to unencrypted traffic.

Recommendations

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

Determine if TLS should be mandatory for the EigenDA proxy and if so, remove the configuration option.

Resolution

The development team have noted the TLS encryption is not required on the proxy server.

EDAP-05	Missing `IsOnCurve` & `IsInSubgroup` Checks For Elliptic Curve Points
Asset	`verify/verifier.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When the client puts a new blob to the EigenDA proxy, the expected commit elliptic curve point is not verified to be on the curve BN254 nor in the subgroup G1.

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

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

```
/ Verify regenerates a commitment from the blob and asserts equivalence
// to the commitment in the certificate
// TODO: Optimize implementation by opening a point on the commitment instead
func (v *Verifier) VerifyCommitment(expectedCommit *common.G1Commitment, blob []byte) error {
  actualCommit, err := v.Commit(blob) //@audit due to calculation actualCommit will exist in G1
  if err != nil {
    return err
  }

  expectedX := &fp.Element{}
  expectedX.Unmarshal(expectedCommit.X) //@audit bytes are converted to field elements but no checks are performed on the resulting
      ↪  point
  expectedY := &fp.Element{}
  expectedY.Unmarshal(expectedCommit.Y) //@audit bytes are converted to field elements but no checks are performed on the resulting
      ↪  point

  errMsg := ""
  if !actualCommit.X.Equal(expectedX) || !actualCommit.Y.Equal(expectedY) {
    errMsg += fmt.Sprintf("field elements do not match, x actual commit: %x, x expected commit: %x, ", actualCommit.X.Marshal(),
        ↪  expectedX.Marshal())
    errMsg += fmt.Sprintf("y actual commit: %x, y expected commit: %x", actualCommit.Y.Marshal(), expectedY.Marshal())
    return fmt.Errorf("%s", errMsg)
  }

  return nil
}
```

It is to be noted that the testing team did not find any directly exploitable vulnerabilities stemming from this issue due to an included check against the blob's actual generated commitment point which must exist in G1 and therefore currently enforces the expected point to also exist in G1.

However, due to planned changes to the verification method, it is advisable to harden the codebase to prevent future issues.

Recommendations

It is recommended to perform checks on each elliptic curve point to ensure:

- Each point is on the curve.
- Each point is in the correct subgroup.

This can be achieved by using the `G1Affine.IsInSubgroup()` and `G1Affine.InOnCurve()` functions from `gnark-crypto` which perform the required checks.

Resolution

The recommendation has been implemented in PR #229.

EDAP-06	TODOs In Code
Asset	`Server/config.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

Multiple areas have "TODO" comments left in the codebase or commented out code with discussion comments surrounding it.

A non-exhaustive list of instances is:

- `flags/flags.go` line [41]
- `flags/eigendaflags/cli.go` line [14] and line [157]
- `cmd/server/entrypoint.go` line [75]
- `server/load_store.go` line [19]
- `server/errors.go` line [53]
- `server/routing.go` line [64]
- `server/server.go` line [121] and line [126]
- `server/handlers.go` line [50], line [126], line [133] and line [161]
- `server/config.go` line [65]
- `server/middleware.go` line [17] and line [75]
- `metrics/metrics.go` line [106]
- `verify/verifier.go` line [43] and line [126]
- `verify/cli.go` line [21] and line [85]
- `verify/cert.go` line [64]
- `verify/deprecated_flags.go` line [10] and line [115]
- `store/precomputed_key/s3/s3.go` line [52]
- `store/precomputed_key/redis/redis.go` line [23]
- `store/generated_key/eigenda/eigenda.go` line [76] and line [116]

Recommendations

These comments should be addressed and resolved prior to using the codebase.

Resolution

The development team have acknowledged the issue.

EDAP-07	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. **Incorrect Variable Comment**
   Related Asset(s): verify/cli.go
   The comment on line [23] incorrectly states the constant as being 2^{28} when it is 2^{32}.
   The comments should be checked and corrected where wrong to prevent future errors arising.

Recommendations

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

Resolution

The issue has been resolved in PR #228.

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.

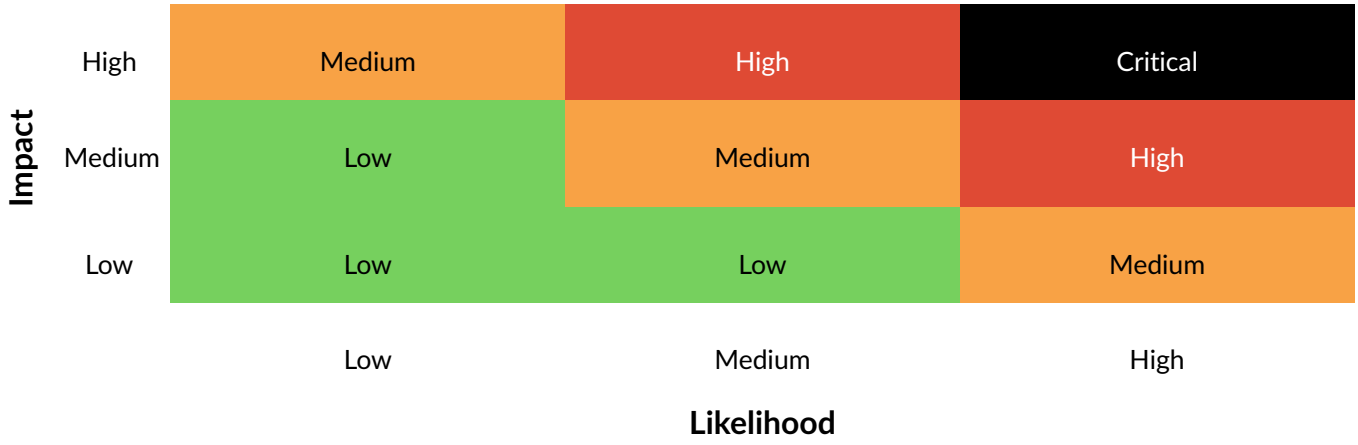

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References