OpenZeppelin | security

Mantle Op-geth & Op-stack Diff Audit

February 12, 2025

Table of Contents

Summary

Type	Layer 2
Timeline	From 2024-01-06 To 2024-01-27
Languages	Go

Total Issues	10 (5 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	6 (3 resolved)
Notes & Additional Information	3 (1 resolved)

Scope

This audit is divided into three parts:

1. A diff audit of the mantlenetworkio/mantle-v2 repo, comparing commit c77cb6e with commit fd99099, and the following files being in scope:

```
mantle-v2
├── op-batcher
│   ├── batcher
│   │   ├── channel_builder.go
│   │   ├── channel_manager.go
│   │   ├── config.go
│   │   ├── driver.go
│   │   └── driver_da.go
│   ├── cmd
│   │   └── main.go
│   ├── flags
│   │   └── flags.go
│   └── metrics
│       ├── metrics.go
│       └── noop.go
├── op-e2e
│   └── actions
│       └── l2_verifier.go
├── op-node
│   ├── flags
│   │   └── flags.go
│   ├── metrics
│   │   └── metrics.go
│   ├── node
│   │   ├── client.go
│   │   ├── config.go
│   │   └── node.go
│   ├── rollup
│   │   ├── da
│   │   │   ├── datastore.go
│   │   │   └── interfaceRetrieverServer/server.pb.go
│   │   ├── derive
│   │   │   ├── calldata_source.go
│   │   │   ├── l1_retrieval.go
│   │   │   └── pipeline.go
│   │   ├── driver
│   │   │   └── driver.go
│   │   └── types.go
│   └── service.go
├── op-program/client/driver/driver.go
└── op-service
    ├── client
```

```
    │   └── http.go
    ├── eigenda
    │   ├── cli.go
    │   ├── codec.go
    │   ├── config.go
    │   ├── da.go
    │   ├── da_proxy.go
    │   ├── derivation.go
    │   └── metrics.go
    ├── eth
    │   ├── blob.go
    │   ├── blobs_api.go
    │   ├── ether.go
    │   ├── id.go
    │   └── types.go
    ├── proto
    │   ├── gen/op_service/v1/calldata.pb.go
    │   └── src/op_service/v1/calldata.proto
    ├── retry
    │   ├── operation.go
    │   └── strategies.go
    ├── sources/l1_beacon_client.go
    ├── txmgr
    │   ├── cli.go
    │   └── txmgr.go
    └── upgrade
        └── mantle_upgrade.go
```

1. A diff audit of the mantlenetworkio/op-geth repo, comparing commit 4a20fa6 with commit 87c2ac2, and the following files being in scope:

```
op-geth
├── consensus/misc/eip1559.go
├── core
│   ├── genesis.go
│   ├── mantle_upgrade.go
│   ├── state_transition.go
│   ├── txpool/txpool.go
│   ├── types
│   │   ├── deposit_tx.go
│   │   ├── meta_transaction.go
│   │   ├── transaction.go
│   │   ├── transaction_signing.go
│   │   ├── tx_access_list.go
│   │   ├── tx_dynamic_fee.go
│   │   └── tx_legacy.go
├── internal
│   └── ethapi
│       ├── api.go
│       └── transaction_args.go
```

```
├── light/txpool.go
└── params/confighttp.go
```

1. An audit of two pull requests in the mantlenetworkio/op-geth repo:

   - Pull request #93 and the following files being in scope:

```
op-geth
├── core
│   ├── genesis.go
│   ├── mantle_upgrade.go
│   ├── txpool/txpool.go
│   └── vm
│       ├── contracts.go
│       └── evm.go
├── crypto
│   └── secp256r1
│       ├── publickey.go
│       └── verifier.go
└── params
      ├── config.go
      └──protocol_params.go
```

   - Pull request #95 and the following files being in scope:

```
op-geth
├── core
│   ├── mantle_upgrade.go
│   ├── state_transition.go
│   ├── txpool/txpool.go
│   └── types/meta_transaction.go
└── light/txpool.go
```

System Overview

Mantle V2 is a layer 2 (L2) scaling solution for Ethereum that uses fraud proofs instead of validity proofs for its security. The protocol aims to provide low transaction fees and high throughput while maintaining full EVM compatibility. Mantle V2 is built on top of Ethereum using the OP Stack and therefore shares many similarities with Optimism. Further details about Mantle can be found in our previous audit reports here, here, here, and here. In this diff audit, the reviewed changes can be categorized into three groups:

Op-stack Updates

The updates to the op-stack focus on enhancing scalability and reliability by addressing data availability (DA) using EigenDA. These can be categorized into three main upgrades:

- **Batcher**: The batcher aggregates transactions and disperses them to EigenDA for storage, posting commitments to L1. If EigenDA is unavailable, Ethereum blobs are used as a fallback.
- **Rollup**: The rollup uses data from EigenDA (or Ethereum blobs) and the commitment (dispersed by the batcher) on Ethereum to reconstruct the L2 blockchain.
- **EigenDA Service**: A new client, `NewEigenDAClient`, facilitates communication with the EigenDA proxy, enabling the retrieval of blobs and their associated commitments, and verifying them when necessary.

Other notable updates include:

- A Mantle update configuration that specifies when each chain should upgrade to EigenDA based on block height.
- Retry handling for operations to ensure failed operations are retried with an appropriate backoff strategy.
- Enhanced tracking of the DA layer with metrics to monitor transaction submissions, handle retries, and manage fallback options in case of issues.

Op-geth Updates

Several changes have been made to the op-geth implementation. These can be categorized into three main categories: updates related to Blob transactions, updates related to meta transactions, and bug fixes.

- **Blob Transactions**: A new file, `eip4844.go`, has been added. Its functions handle header verification for blob transactions as defined by EIP-4844 and compute the base fee for such transactions. In addition, support for signing such transactions has been introduced by implementing the Cancun signer. Blob transactions are designed to be included in L1 blocks and are not relevant to the L2 execution layer. As a result, the functions in this file are not utilized elsewhere in the op-geth code.
- **Meta Transactions**: Meta transactions have been disabled, as the Mantle team plans to implement EIP-7702 as a replacement in a future upgrade.
- **Gas Bug Fix**: A bug related to the computation of the gas for transactions with a sender having pending transactions has been fixed.
- **Proxy Admin Owner Update**: The proxy admin owner for the proxy contract of Mantle's system contracts had been previously set incorrectly. Now, this issue has been addressed by implementing an update of the owner address at the node level.

New Precompile for Signatures on `sec256r1`

Mantle has implemented a new precompile (`p256Verify`) for verifying ECDSA signatures on the `sec256r1` curve, following RIP-7212. The `sec256r1` curve is widely adopted and supported by many modern devices, making the addition of this precompile highly convenient.

`p256Verify` is similar to `ecrecover` and the main differences lie in the choice of the elliptic curve and the expected arguments. `ecrecover` verifies signatures on the `sec256k1` curve. Furthermore, while `ecrecover` takes as arguments the message hash along with the three fields of the signature: `v`, `r`, and `s`, and the public key of the signer can be recovered from them, `p256verify` expects the hash of the message, the signature `(r,s)`, but also the public key of the signer `(x,y)` to be provided explicitly.

When invalid arguments are provided, such as if the `r` or `s` are out of range or if the public key does not represent a valid point of the curve, the precompile does not revert but instead returns `false`. Moreover, as stated in RIP-7212, the precompile permits malleable signatures, and it is left to the wrapper libraries to add a malleability check, if needed. The gas cost for

calling this new precompile has been set to 3450, slightly higher than that of `ecrecover`. To implement input validation and signature verification, the precompile relies on standard Go libraries, as `crypto/ecdsa` and `crypto/elliptic`. These libraries are widely used and were considered safe for the purposes of this audit.

Security Model and Privileged Roles

This audit focused on reviewing changes across various components of the Mantle chain. Since this was a diff audit, we only assessed the modified code and, thus, cannot guarantee the correctness of the unchanged parts.

In the `op-stack`, we assume that the EigenDA proxy server, hosted by Mantle, is functioning as expected, verifying both the DA certificate and KZG commitments. We also assume that the parameters and servers are correctly configured to ensure seamless communication between the EigenDA service and EigenDA, that the EigenDA dispersal process is actively monitored, and that in the event of failure, the system automatically adjusts the `SkipEigenDARpc` toggle for a quicker fallback to Ethereum blobs.

In the `op-geth` part, we identified several hardcoded values, primarily related to timestamps for scheduled upgrades and rules changes. During the audit, many of these values had not been set yet and were marked as TODO. We assume these values will be properly configured before deployment. In addition, a new proxy admin owner will be assigned to the proxy contract that governs most of Mantle's system contracts. This address will have the authority to update system contracts at will, particularly those controlling the bridging mechanism with L1. The designated admin address is expected to be controlled by a multisig and is assumed to be trusted to ensure the proper functioning of the system.

Medium Severity

M-01 Inefficient Error Handling and Timeout in `loopEigenDa` Loop

The `loopEigenDa` function of the `BatchSubmitter` attempts to call `disperseEigenDaData` within a retry loop that continues for either a fixed number of retries (`EigenRPCRetryNum`) or until a timeout (`DisperseBlobTimeout`) is reached. However, the implementation neglects to properly handle errors returned by `disperseEigenDaData`. Instead of classifying or acting on the errors, the loop merely logs them and retries, effectively waiting for the entire `DisperseBlobTimeout` duration, even in cases where the operation cannot succeed due to permanent or critical errors.

This oversight can lead to inefficient resource utilization and unnecessary delays. In scenarios where a critical error occurs (e.g., invalid input data or persistent configuration issues), the loop will still continue retrying until the timeout expires. This results in wasted computation time, potential bottlenecks in the rollup process, and delayed submission of transaction data. Furthermore, this behavior can obscure the root cause of errors in the logs as the same error is repeatedly logged without actionable resolution.

To address this issue, consider enhancing the error-handling logic to distinguish between transient errors (e.g., network issues) and permanent ones (e.g., invalid data). For transient errors, the loop can continue with retries, potentially implementing exponential back-off to reduce retry frequency over time. For permanent errors, the loop should exit immediately to avoid unnecessary delays. Moreover, the timeout check should be performed at the beginning of each retry iteration to ensure that retries do not continue beyond the configured timeout.

Update: *Resolved in pull request #192. The Mantle team implemented a more fine-grained distinction between transient and permanent errors. Transient errors will be retried, while permanent errors such as EigenDA invocation errors will exit immediately, requiring manual investigation.*

Low Severity

L-01 Duplication of RLP Encoding

In the `loopEigenDa` function, the `txAggregatorForEigenDa` function performs the RLP encoding of transaction data to validate its size against `RollupMaxSize`. However, the method only returns the raw, non-encoded transaction data (`txsData`). Subsequently, in `loopEigenDa`, the same data is passed to `disperseEigenDaData`, where it is RLP-encoded again. This duplication of the RLP encoding process is unnecessary and inefficient, as the encoded data from `txAggregatorForEigenDa` could have been reused directly.

This redundancy introduces computational overhead, especially when handling large datasets, as RLP encoding can be resource-intensive. Repeating the encoding process effectively doubles the time and CPU cycles required for this operation, which could lead to performance degradation. In addition, the current implementation introduces unnecessary complexity and inconsistency, as the RLP-encoded data is discarded instead of being reused, making the data flow harder to follow and maintain.

To resolve this issue, consider updating the `txAggregatorForEigenDa` function to return both the RLP-encoded data (`transactionByte`) and the raw transaction data (`txsData`). The `loopEigenDa` function can then directly use the encoded data in `disperseEigenDaData`, eliminating the need for a second encoding pass. This change reduces computational overhead, simplifies the logic, and ensures consistent handling of the encoded data. By avoiding redundant operations, the overall performance and maintainability of the code are significantly improved.

Update: *Resolved, not an issue. The Mantle team stated:*

> *We tested the performance of `rlp.EncodeToBytes` for data of 4MiB (4 times the size of a typical Mantle blob) in the following code. The test results showed an average time consumption of 0.43ms per EncodeToBytes operation.*

```
bash pkg: github.com/ethereum-optimism/optimism/op-batcher/batcher
BenchmarkRLPEncoding
BenchmarkRLPEncoding-8
2767        438315 ns/op     5434873 B/op          2 allocs/op
```

> *Compared to the approximately 1-minute duration for a single loopEigenDa operation, the impact on performance is negligible. Therefore, we believe this will not cause any performance issues.*

```
go func BenchmarkRLPEncoding(b *testing.B) { // Generate 40 random 128KB chunks data := make([][]byte, 40) for i := range data { data[i] = make([]byte, 128*1024) _, err := rand.Read(data[i]) if err != nil { b.Fatalf("Failed to generate random data: %v", err) } } b.ResetTimer() for i := 0; i < b.N; i++ { _, err := rlp.EncodeToBytes(data) if err != nil { b.Fatalf("RLP encoding failed: %v", err) } } }
```

L-02 Inadequate Handling of Single Large Data Frames in `blobTxCandidates`

In the `blobTxCandidates` function, the code assumes that data exceeding the size limit (`se.MaxBlobDataSize * MaxblobNum`) results from the aggregation of multiple frames. However, it does not account for the possibility that a single frame (`frameData`) could individually exceed the maximum size. When this occurs, the function appends the oversized frame to `dataInTx` and proceeds without any special handling, which may lead to unintended behavior during blob creation.

If a single frame exceeds the maximum blob size, the function does not split it into smaller chunks or handle the error explicitly. This can result in the generation of transaction candidates with empty blobs or transaction candidates with an amount of blobs that exceeds the configured amount of maximum blobs (`MaxblobNum`), which is currently set to 4.

Consider introducing explicit handling for single frames that exceed the maximum size. Before appending a frame to `dataInTx`, check whether the frame itself exceeds the size limit. If it does, log an error and return an appropriate error message. Alternatively, consider implementing logic to split oversized frames into smaller chunks before proceeding. Another solution could be to ensure that a single frame will always fit in a single blob transaction by enforcing that the configured `MaxFrameSize` is smaller than the maximum size (`se.MaxBlobDataSize * MaxblobNum`). Adding this check will ensure robust handling of all edge cases.

Update: *Resolved in pull request #194. The Mantle team stated:*

> *A check has been introduced to ensure a single frame is not larger than `MaxBlobDataSize * MaxblobNum`.*

L-03 Redundant Encoding and Decoding of Commitments in `disperseEigenDaData`

In the `disperseEigenDaData` function, the commitment retrieved from `EigenDA.DisperseBlob` is first decoded in `DisperseBlob` using `DecodeCommitment` and then is returned to `disperseEigenDaData`, where it is subsequently re-encoded using `EncodeCommitment`. This results in unnecessary computational overhead, as the commitment undergoes an extra encoding-decoding cycle that does not add value to the process. This redundancy increases computational costs and adds inefficiency to the blob dispersal process. The extra encoding and decoding operations introduce unnecessary processing time, which could impact the overall performance of batch submission.

To optimize the process, consider having `DisperseBlob` return both the encoded and decoded versions of the commitment, eliminating the need for `disperseEigenDaData` to re-encode it while still being able to extract information from the decoded commitment and adding the encoded version to the calldata frame.

Update: *Resolved, not an issue. The Mantle team stated:*

> *In our tests, the average time elapsed for each decode is 0.0013ms, and for each encode is 0.0004ms. Moreover, `DecodeCommitment` and `EncodeCommitment` are not high-frequency operations, so their impact on performance is very minimal. We believe it's not a issue.*

L-04 Direct Communication with EigenDA Disperser in `RetrieveBlob`

The rollup service leverages `RetrieveBlob` to retrieve blobs from EigenDA in order to reconstruct the L2 state. This `RetrieveBlob` function calls the `RetrieveBlob` function of the EigenDA client when the length of the commitment is 0. This function directly communicates with the EigenDA disperser over gRPC to retrieve the blob based on the `BatchHeaderHash` and `BlobIndex` instead of using the EigenDA proxy used by the `RetrieveBlobWithCommitment` function.

Using the EigenDA proxy instead of the EigenDA disperser directly to retrieve blobs comes with several benefits such as KZG verification ensuring data correctness and DA certificate verification ensuring data represented by bad DA certificates does not become part of the canonical chain. By using the disperser directly, these verification checks are not performed,

which could lead to an inability to reconstruct the proper L2 state when using EigenDA to retrieve blobs. Do note that this only happens when the commitment length is equal to zero.

While this generally should not happen as the batcher will not post empty commitments to L1, consider adding a check to ensure that the `RetrieveBlob` function in the rollup service is not invoked with a commitment length of 0.

Update: *Acknowledged, will resolve. The Mantle team stated:*

> *The Direct Communication with EigenDA Disperser in RetrieveBlob is retained compatibility code for a historical EigenDA integration version on the sepolia testnet. It will not be executed on the mainnet. We will remove these codes in the next protocol upgrade.*

L-05 Inconsistent Signer Selection in Transaction Signing

In `transaction_signing.go`, there are three functions responsible for returning the appropriate Signer type for the chain: `LatestSignerForChainID`, `LatestSigner`, and `MakeSigner`. While these functions differ in their inputs, they are expected to return the same Signer types.

- `LatestSignerForChainID`: Only accepts the chain ID as an argument.
- `LatestSigner`: Accepts the chain configuration.
- `MakeSigner`: Accepts both the chain configuration and the block number.

However, there is an inconsistency. The `LatestSignerForChainID` function returns `cancunSigner`, reflecting that the Cancun upgrade is expected to be the most recent op-geth update adopted by Mantle. In contrast, the other two functions, even if the chain has been upgraded to Cancun, return `londonSigner` instead of `cancunSigner`.

The only difference between `londonSigner` and `cancunSigner` is that the latter supports Blob transactions. However, Blob transactions are meant to be posted on L1 and are not intended for inclusion in L2 blocks. In fact, such transactions are discarded from the L2 `txpool`. Despite this, since `LatestSignerForChainID` returns `cancunSigner`, it allows users to create Blob transactions for the L2 by calling `NewKeyedTransactionWithChainID` and providing a Blob transaction, even though that transaction will be rejected.

To ensure uniform behavior of the three functions and to prevent attempts to submit Blob transactions to the L2 chain, consider modifying `LatestSignerForChainID` to return `londonSigner`. This change would align the behavior of all three functions and maintain the intended handling of transactions on L2.

Update: *Acknowledged, will resolve. The Mantle team stated:*

> *Acknowledged. It will be fixed in the next version.*

L-06 Error-Prone Proxy Admin Owner Upgrade Logic

Most Mantle system contracts are implemented behind a proxy contract. This proxy contract is controlled by a `ProxyAdmin` contract. However, as part of the previous upgrade, an invalid address has been inadvertently set as the owner of the `ProxyAdmin` contract. In this upgrade, the issue is remedied by setting the correct address as the proxy admin owner.

The owner upgrade is handled in the `TransitionDb` function, which updates the actual state of the `Proxy` contract. More specifically, if the block's timestamp matches the predefined `ProxyAdminUpgradeTime` (this value was not yet set in the version of the code provided for this audit), the proxy owner is updated to the correct address.

This approach is error-prone as it relies on the following assumptions:

- If the majority of nodes update before the target `ProxyAdminUpgradeTime`, the other minority of nodes has to update and resync the chain in order to have the correct view of the state. Similarly, If a minority of nodes updates before the target `ProxyAdminUpgradeTime`, the owner state update will not happen and a new `ProxyAdminUpgradeTime` has to be scheduled.
- The predefined `ProxyAdminUpgradeTime` must exactly match a future block's timestamp for the owner upgrade to be executed successfully. If the block times are not fixed, this is an impossible task.

While in Mantle a fixed block time of 2 seconds is configured, which allows for more accurate predictions of future block times, this reliance on precise timing remains risky.

Consider replacing the current timestamp-matching logic with a more flexible and robust approach. For example, the owner upgrade could be executed if the block timestamp is greater than or equal to `ProxyAdminUpgradeTime` and the upgrade has not been already executed.

Update: *Acknowledged, not resolved. The Mantle team has decided to keep the simple* `ProxyAdmin` *owner upgrade logic. The team will announce the upgrade well in advance, giving nodes sufficient time to upgrade before the* `ProxyAdminUpgradeTime`*. The team told us that they expect only a small number of nodes to fail to to upgrade on time, and these can resync to align with the correct system view.*

Notes & Additional Information

N-01 Missing Signature Malleability Check

ECDSA signatures are known to be malleable due to the fact that both `(r,s)` and `(r,-s)` are valid signatures for the same message `m`. To remove this undesirable property, implementations typically enforce `s <= N/2`, where `N` is the order of the elliptic curve group (`N := curve.Params().N`). This naturally rejects the second valid signature `(r,-s)` for which `-s=N-s > N/2` (since `s <= N/2`).

The implementation of p256Verify does not apply the mentioned malleability check on the value of `s`. This is intentional and is motivated by the RIP-7212 standard, which states the following:

> "[...] the NIST FIPS 186-5 specification does not include a malleability check. We've matched that here [in RIP-7212] in order to maximize compatibility with the large existing NIST P-256 ecosystem. Wrapper libraries SHOULD add a malleability check by default, with functions wrapping the raw precompile call (exact NIST FIPS 186-5 spec, without malleability check) clearly identified. For example, `P256.verifySignature` and `P256.verifySignatureWithoutMalleabilityCheck`. Adding the malleability check is straightforward and costs minimal gas."

Consider explicitly documenting that the implementation of `p256Verify` intentionally does not have a malleability check and referring to the relevant part of the RIP-7212 standard for the motivation.

Update: *Resolved. The Mantle team has explicit documentation stating that the new* `p256Verify` *precompile does not perform any malleability checks.*

N-02 Missing Documentation

Throughout the parts of the codebase relevant to PR #93, multiple instances of missing comments and opportunities for improving documentation were identified:

- There is a missing comment in `contracts.go` describing the `PrecompiledContractsMantleEverest` map similar to the other maps (e.g., the comment to the Berlin release).
- The `newPublicKey` function returns nil in two cases: first, when at least one of the coordinates is invalid (i.e., `x` or `y` or both are `nil`), and second, when the point is not on the curve.
- The `Verify` function does not explicitly check that the `r,s` inputs are in the correct range `1<= r,s < N`, where `N` is the curve order. Instead, it delegates the actual verification to the Go standard library's `ecdsa.Verify`, which performs the check implicitly.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. In addition, for enhanced readability and maintainability consider providing clear documentation within the codebase for cases where the behavior is not explicit or self-explanatory.

Update: *Acknowledged, will resolve. The Mantle team stated:*

> *Acknowledged. It will be fixed in the next version.*

N-03 Improved Efficiency of Input Validation

The `newPublicKey` function correctly validates the `x,y` input, ensuring that:

1. `x,y` is not an invalid input `nil,nil`.
2. The point `x,y` lies on the P256 curve by calling `IsOnCurve(x, y)`.

The function does not explicitly check that `x,y` is not the point-at-infinity, which is commonly represented as `0,0` (in affine coordinates) (e.g., as noted in RIP-7212). However, the point `0,0` does not lie on the P256 curve, so the call to `IsOnCurve` will reject it. Therefore, this check is implicit. While the employed checks are appropriate, the function `IsOnCurve`, which performs elliptic curve arithmetic, is computationally expensive.

To improve efficiency, consider implementing a (more) efficient preliminary range check on `x,y` before making the expensive call to `IsOnCurve`. This check is similar to what is done in ecrecover and validates that `x,y` have the expected byte length and that their values are in the expected range determined by the curve order `N := curve.Params().N`.

Note that the mentioned preliminary range check on `x,y` is also performed by `IsOnCurve` and so will be done twice for valid points. As such, the decision to implement it or not depends on how frequently `IsOnCurve` is expected to be called. Alternatively, one may modify the `IsOnCurve` implementation in the Go standard library to remove the extra range check.

Consider implementing a preliminary range check on `x,y` keeping in mind the mentioned efficiency trade-offs. In addition, consider explicitly rejecting the case `x,y=0,0` or at least documenting that the check is implicit.

Update: *Acknowledged, will resolve. The Mantle team stated:*

> *Acknowledged. It will be fixed in the next version.*

Conclusion

In this update, Mantle introduced several new features and improvements over the previous version. Notably, EigenDA has been implemented as the primary data availability layer, while Ethereum blobs are being used as the fallback. Additionally, a new precompile for verifying signatures on `sec256r1` have been introduced and meta transactions are being disabled. This update also includes some minor improvements and bug fixes which increases the overall robustness of the chain.

The audit identified one medium-severity and a few low- and note-severity issues. Although some of the changes were adopted from Optimism, these changes have been audited independently. The Mantle team was highly responsive and provided detailed answers to our questions.