OpenZeppelin | security

Mantle OP-Geth Audit

MANTLE

March 15, 2024

Table of Contents

Summary

Type	Layer 2
Timeline	From 2024-02-09 To 2024-02-29
Languages	Go

Total Issues	18 (7 resolved, 1 partially resolved)
Critical Severity Issues	3 (3 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	5 (2 resolved, 1 partially resolved)
Notes & Additional Information	10 (2 resolved)

Scope

We audited the mantlenetworkio/op-geth repository at head commit 4d05b2c. All files were diff-audited against the base commit 0a77db9. Any code outside of this diff was *not* audited. As such, we do not guarantee the correctness of the unchanged code.

In scope were the following files:

```
op-geth
├── accounts/abi/bind/backends/simulated.go
├── beacon/engine
│   ├── gen_blockparams.go
│   └── types.go
├── cmd
│   ├── evm/internal/t8ntool/execution.go
│   └── utils/flags.go
├── consensus/misc/eip1559.go
├── core
│   ├── genesis.go
│   ├── mantle_upgrade.go
│   ├── state_prefetcher.go
│   ├── state_processor.go
│   ├── state_transition.go
│   ├── txpool/txpool.go
│   ├── types
│   │   ├── deposit_tx.go
│   │   ├── gen_receipt_json.go
│   │   ├── meta_transaction.go
│   │   ├── rollup_l1_cost.go
│   │   ├── transaction.go
│   │   └── transaction_marshalling.go
│   └── vm
│       ├── interpreter.go
│       ├── jump_table.go
│       └── runtime.go
├── eth
│   ├── api_backend.go
│   ├── backend.go
│   ├── catalyst/api.go
│   ├── ethconfig/config.go
│   ├── state_accessor.go
│   └── tracers
│       ├── api.go
│       └── internal/tracetest/calltrace_test.go
├── graphql
│   └── graphql.go
├── internal
│   └── ethapi
```

```
│       ├── api.go
│       └── transaction_args.go
├── les/state_accessor.go
├── light/txpool.go
├── miner
│   ├── miner.go
│   ├── payload_building.go
│   └── worker.go
├── params
│   ├── config.go
│   └── protocol_params.go
└── rpc/http.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. This audit particularly focuses on its L2 execution client which has been forked from Optimism's op-geth repository. While changes have been made in order to customize the client as per Mantle's needs, its name still remains op-geth. Therefore, in this report, the term "op-geth" will refer to Mantle's version of op-geth, rather than Optimism's.

Mantle V1 has been running on Optimism's OVM so far. Hence, adopting the OP Stack Bedrock version was a big transition. Other changes include EIP-1559 support, removal of redundant components, stable block time, and block state tagging. This sets the starting point (base commit) of the audit. While the above changes reflect Mantle's compatibility with OP Stack Bedrock, other changes have been introduced that set Mantle apart. These distinctions include using Mantle DA as a data availability layer, using MNT as the native L2 token instead of ETH, native meta transactions, and the fee optimization strategy. This report specifically focuses on the last three distinctions which are changes introduced in the head commit.

Gas on L2

Due to MNT being the native token on L2, the way in which gas works can result in a slightly different behavior than on L1. For all L2s, there are two costs that the user must pay for: the L2 execution cost and the L1 data availability cost. Therefore, the gas fee charged to the user must cover both of these. However, the latter cost is in terms of ETH, and because MNT is the native token on L2, users must pay their gas fees in MNT. Mantle's solution to this is to use a `tokenRatio` in order to scale units of ETH gas to units of MNT gas. This `tokenRatio` is defined as the price of ETH divided by the price of MNT. Thus, depending on the price of these two assets, the amount of L2 gas the user must pay for their transaction to succeed will vary.

This contrasts with the behavior on L1, where the same transaction executed on the same state should cost the same amount of gas. Therefore, it is possible that a user executes a transaction with a certain gas limit on L2 and, depending on the `tokenRatio`, that transaction may succeed or may revert due to running out of gas. To try and prevent this from

happening, Mantle provides a way for users to perform gas estimation and returns a 20% buffer in its calculation to allow more room for fluctuation. Nevertheless, if ever there is a drastic price change in one of the two assets, this scenario could occur.

Security Model and Privileged Roles

The op-geth execution client relies on the appropriate configurations to be set correctly. Some of these parameters are hardcoded into the client code itself, whereas others need to be passed in and read. The party which feeds in these parameters is a trusted entity. In addition, the correct configuration also includes good access control for the L1 contracts upon which the system rests. The client also relies on information from the op-node. For example, deposit transactions (which come from L1) are first processed by the op-node before being sent over to op-geth. The assumption is that the op-node is processing these deposit transactions correctly. At the moment, Mantle is a centralized system and therefore inherits all the potential risks from being one. Furthermore, op-geth relies on some specific smart contracts on L2 for important information such as the `tokenRatio`, the L1 `basefee`, other parameters for fee calculation, and even the ERC-20 contract which is L2 ETH itself. Op-geth depends on the fact that the smart contracts are working as intended.

In terms of bridging, Mantle then relies on the correctness of the implementation of its L1 and L2 bridging contracts. What the op-geth expects needs to be aligned with the behavior of the smart contracts. For example, if a user bridges through the L1 messenger smart contract into an L2 account and the L2 transaction reverts, there is a chance for the funds to be stuck. However, at the moment, this is unlikely to happen as the values are hardcoded correctly on L1 such that this would not be allowed to occur. This interplay between the L1 smart contracts and op-geth is crucially important in order for the system to function properly. The security of any L2 is dependent on the security of the underlying L1 blockchain. Since Mantle is an optimistic rollup, the security of the system depends on the ability to send and execute fraud proofs in a manner that disincentivizes malicious behavior. While Mantle seeks to be as EVM compatible as possible, there are some differences with Ethereum which can be viewed in their documentation.

Critical Severity

C-01 Attacker Can Get Infinite `BVM_ETH` Tokens to Drain the Protocol

The process of depositing MNT and ETH from L1 to L2 starts in the `depositTransaction` function of the `OptimismPortal` contract. While this contract is used by the `L1CrossDomainMessenger` contract, it can also be called by users directly. It allows anyone to specify the values to mint and/or transfer MNT and/or ETH on L2. The values of minted MNT and minted ETH are determined by pulling the MNT token from the user and `msg.value`. However, the transaction values of MNT and ETH are just forwarded from the user input to the `TransactionDeposited` event. The node listens to this event, parses it, and includes it in a block to execute it in the client.

When the client processes the deposit transaction, it is checked that the L2 user has enough balance available for the given MNT transfer value. If not, the execution reverts. However, this check is never performed for the ETH transaction value. In fact, the ETH balance transfer is performed by reading the `from` and `to` balance from their contract storage slot, applying the difference in value, and setting the new state for these slots. **This happens without any over-/underflow check.** Furthermore, the `common.BigToHash` function that is used to set the state, changes a **negative** `big.Int` value to a **positive** hexadecimal representation.

This means that an attacker with zero ETH balance on L2 can initiate a deposit transaction to transfer 100 ETH to their second controlled address. Then, in the `transferBVMETH` function, their `from` balance is calculated as -100 ETH but written to state as +100 ETH, while their second account also gets another +100 ETH, totaling a gain of 200 ETH on L2 without any L1 investment (besides gas). This ETH value could simply be withdrawn to L1 to drain all of the locked ETH.

Consider applying stronger balance and overflow checks when directly manipulating the state of an asset.

Update: *Resolved in pull request #42 at commit 0cf00ba.*

C-02 Wrong Cost Accounting

The `Cost` function of the `transaction.go` file supposedly determines the L2 transaction cost that a user is charged. This is calculated as `GasPrice * GasLimit` for the maximum gas cost, adding the transaction `Value`, and optionally adding the blob transaction costs as `BlobGas * BlobGasFeeCap` (which is not yet supported). When the function was patched to add the optional blob costs, the team introduced a copy-paste mistake by removing the `Value` cost addition. As such, while the return value was interpreted to include the `Value`, it actually did not. This has a bad impact on the transaction validation of the transaction pools [1, 2] where it is checked whether the user's balance covers the transaction costs.

However, as the cost does not include the transaction value, a transaction the user could not afford by value would still be added to the pool as valid. Then, when the transaction is processed during `state_transition.go` to buy gas, the balance check is performed again, this time with the `Value`, which causes the transaction to revert without charging the user. This leads to a DoS attack vector, where an attacker can spam the network with lucrative transactions that would be prioritized by the node, but never get executed at no cost while also preventing other transactions from being added to the transaction pool, effectively causing the blockchain to stop working.

Consider adding the transaction `Value` to the transaction cost. Also, consider whether the code redundancy of cost calculation can be better managed with one or two methods.

Update: *Resolved in pull request #41 at commit 44c9a41.*

C-03 Potential Insufficient Balance for Sponsorship

The purpose of the `validateMetaTxList` function of `core/txpool/txpool.go` is to check if the sponsor has enough balance to cover the gas fees of the transactions that it is sponsoring. This is used by the `validateTx` function, the `promoteExecutables` function, and the `demoteUnexecutables` function, in order to determine which transactions are valid.

The `validateMetaTxList` function iterates over all the transactions in a list, and checks if the sponsor has enough to fund each transaction individually. However, there can be scenarios in which a sponsor has enough balance to fund each transaction individually, but not all of the transactions combined. The node then views all of these transactions as valid and keeps them in its `txpool`. It is not until the node builds a block and begins to process these transactions that it realizes that some of these transactions may fail. As such, a malicious attacker could

perform a DoS attack on a node by submitting many sponsored transactions to sponsor as much as its balance, and while the node views all of the transactions as valid and thus stores them in its `txpool`, only one of them can actually be valid. This could stall the node from processing other users' transactions, effectively causing the L2 to stop functioning.

Consider updating the `validateMetaTxList` function to check if the balance of the sponsor can pay for the `sponsorCostSum`, which is the total of the sponsored amounts.

Update: *Resolved in pull request #43 at commit 2619376.*

Low Severity

L-01 Incorrect Error Message

In the `validateTx` function of `light/txpool.go`, if the sponsor does not contain sufficient funds in order to pay for the `sponsorAmount`, the error returned is `core.ErrInsufficientFunds`. However, for clarity and consistency with `txpool/txpool.go`, consider returning the `types.ErrSponsorBalanceNotEnough` error instead.

Update: *Resolved in pull request #50 at commit 65ab3a1.*

L-02 Use of Non-Granular Value for `tokenRatio`

The value of `tokenRatio` is stored in the `GasPriceOracle.sol` contract on L2 as a `uint256`. This `tokenRatio` is intended to take on the quotient value of ETH price divided by MNT price, and it is used to calculate the gas on L2. Currently, when this value is updated, all decimals are truncated. Given the current market price of ETH and MNT, the truncation would only result in a slight error in the gas calculation. However, if the value of MNT grows relative to the price of ETH, the error will continue to grow.

Therefore, consider scaling up the `tokenRatio` value and, upon using this value in op-geth, scale it back down to its proper scale at that point in order to improve precision.

Update: *Acknowledged, will resolve.*

L-03 The Light `txpool` Implementation Does Not Account for L1 Costs

The light client implementation of `txpool` validates added transactions just as the core implementation does. This includes checking whether the user and the meta transaction sponsor have enough balance to cover the costs.

However, the user balance is only checked against the L2 cost [1, 2] without the L1 fee, even though it is necessary to account for the rollup transaction cost to L1. This implies that the underestimated cost could lead to the transaction reverting due to insufficient funds once it is sent to a full node.

Despite the light node not being in use yet, consider correcting the validation to account for the L1 costs in order to be prepared for when the network goes decentralized with light nodes.

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

> *The light node is not yet in use, we will fix it later.*

L-04 Fragilely Shared Pointer

In the `buyGas` function of the `state_transition.go` file, the value of `mgval` is intended to be copied into `balanceCheck`. However, instead of the value, a pointer to the value is copied. Hence, with this assignment, the variables share the same memory. Luckily, the `balanceCheck` variable then gets another pointer assigned due to `new(big.Int).SetUint64()`.

However, if in future revisions this code were to be changed to `balanceCheck.SetUint64()`, no new memory would be allocated for this value, implying an overwrite of `mgval` with it. In this code context, this means that the transaction value is additionally considered as gas cost, leading to a double spending of the value or additional spending for the transaction sponsor.

Consider avoiding a shared pointer altogether by always allocating a new value through `new(big.Int).SetUint64()`. While this is not an issue in the given function, it is better to minimize the risk before it escalates in the future if not treated carefully.

Update: *Resolved in pull request #52 at commit 75b60cb.*

L-05 Misleading Documentation

Throughout the codebase, there are instances of misleading documentation:

- In line 80 of `transaction.go`, the comment currently says "This is implemented by DynamicFeeTx, LegacyTx and AccessListTx", but should be updated to include the `BlobTx` and `DepositTx` types.
- In line 377 of `transaction.go`, the comment currently says "gas * gasPrice + value", but should say "(gas * gasPrice) + (blobGas * blobGasPrice) + value" instead.
- In line 655 of `state_transition.go`, the comment currently says "Return ETH for remaining gas", but should say "MNT" instead.

Furthermore, there is misleading documentation in the developer documentation as well:

- The *"BASEFEE Adjustment Mechanism"* section "Application of EIP-1559 in Mantle v2" is outdated. With the Mantle `BaseFee` upgrade, the `BaseFee` is set by a config contract (or remains as the last `BaseFee`).

Consider fixing the reported documentation instances to improve the overall readability of the codebase.

Update: *Partially resolved in pull request #41 at commit 44c9a41 and pull request #52 at commit 2527a58. While the comments on line 377 in `transaction.go` and line 655 in `state_transition.go` have been resolved, the comment on line 80 in `transaction.go` as well as the developer documentation remain unchanged.*

Notes & Additional Information

N-01 Code Redundancy

Code redundancy can lead to code bloat and an increased error surface when updating the code in the future. Throughout the codebase, there are several instances of code redundancy:

- The `newPUSH0InstructionSet` function contains the same code as the `newShanghaiInstructionSet` function.

- This section of code in the `DoEstimateGas` function which is to estimate the gas cap for a meta transaction is redundant given the `calculateGasWithAllowance` function.
- In line 87 of `meta_transaction.go`, the `len(MetaTxPrefix)` is the same as the existing constant `MetaTxPrefixLength`.
- In lines 73 to 76 of `rollup_l1_cost.go`, the contract state is fetched directly instead of using the `DeriveL1GasInfo` function.
- In `worker.go`, the way the `header.BaseFee` can be overwritten twice is redundant.

Consider removing any instances of code redundancy or explicitly documenting why they are necessary.

Update: *Acknowledged, will resolve.*

N-02 Incorrect Module Name

The Mantle codebase is currently using `github.com/ethereum/go-ethereum` as its module name, as seen in the `go.mod` file. The usage of this name, which is from a GitHub repo that it does not own, prevents others from importing Mantle's code and makes it more challenging for testing.

Consider using Mantle's own path for its module name.

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

> *No issue. In mantle-v2, we will replace `github.com/ethereum/go-ethereum v1.11.6` => `github.com/mantlenetworkio/op-geth...` and so on.*

N-03 Struct Field and Tag Mismatch

In the `txJSON` struct, the `Data` field is tagged and referred to as "input". In the `go-ethereum` reference implementation, this field is called `Input` as well.

To prevent confusion about the parameters and to close the gap between this and the `go-ethereum` repository, consider renaming the `Data` field to `Input`.

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

> *Will not fix now, we will upgrade to the latest `go-ethereum` later.*

N-04 Block Number Expiry Potentially Confusing

The validity of a meta transaction can be limited by the block number. When this block number is exceeded, the transaction will not be accepted as valid. However, from a user experience perspective, it is less intuitive for a sponsor to set a block number as opposed to a timestamp.

To enhance the user experience, consider changing the unit from block number to block timestamp.

Update: *Acknowledged, not resolved. The Mantle team stated:*

> *The blockheight and blocktime of the mantle-v2 blocks correspond one-to-one, and this is only for user experience. If we modify it, it will introduce significant modifications. Will not fix.*

N-05 Unused Variables

As the codebase grows and matures, unused variables can cause code bloat, naming collisions, and confusion when reading the code. Throughout the codebase, there are instances of unused variables:

- The `OptimismL1FeeRecipient` variable is not used since the L1 fee is already included in the fee that goes towards the `OptimismBaseFeeRecipient`.
- The `OverrideMantleBaseFee` variable is unused and from the comment it should be removed after the fork.
- The `OverrideShanghai` variable is unused and from the comment it should be removed after the fork.

Consider removing these unused variables to improve code clarity.

Update: *Acknowledged, will resolve.*

N-06 Inexplicit Struct Declaration

It is considered best practice to create a struct using the keys explicitly when instantiating it. This helps improve the readability and maintainability of the codebase. Struct usage with inexplicit declarations reduces code readability and is more error-prone. One instance of an inexplicit struct declaration is `EthAPIBackend` in `backend.go`.

Consider using the key-value syntax for explicitness.

Update: *Resolved in pull request #55 at commit 79bab65.*

N-07 Unclear Calldata Byte Counting

In Ethereum, zero and non-zero bytes in the `calldata` are taxed differently. Hence, when a transaction buys gas in the gas estimation (without balance check) run mode, it counts the zero and non-zero bytes of the RLP-encoded transaction to approximate the L1 fee cost. For these specific run modes, a heuristic value of 80 is added to the number of `Ones` (non-zero bytes). This is to cover transaction fields that are unknown at the time of estimation but will carry non-zero bytes data during execution. Furthermore, these byte counts are used in the `DataGas` function to apply the different gas cost per each zero and non-zero byte. However, before the Regolith update, a magic value of 68 is added to the `Ones`.

Consider using a `const` value at the top of the file along with some context information instead of magic numbers. This will ensure the maintenance of the value as the protocol progresses. Moreover, consider renaming the `Ones` field of the `RollupGasData` struct to `NonZero` in order to better reflect its meaning.

Update: *Acknowledged, will resolve.*

N-08 Todo Comments in the Code

During development, having well-described TODO comments will make the process of tracking and solving them easier. Without such information, these comments might age and important information for the security of the system might be forgotten by the time it is released to production. These comments should be tracked in the project's issue backlog and resolved before the system deployment.

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO comment to the corresponding issues backlog entry.

Update: *Acknowledged, will resolve.*

N-09 Unnecessary Double Check of `SponsorPercent`

Meta transactions are transactions where the another party other than the `tx.origin` can agree to sponsor the gas fees or a percentage of them on behalf of the user. The percent that a sponsor can sponsor up to should have an upper bound of 100%. In `meta_transaction.go`, this upper bound is checked in lines 100 to 102 in the `DecodeMetaTxParams` function.

In the `DecodeAndVerifyMetaTxParams` function, this `DecodeMetaTxParams` function is called and then subsequently the upper bound of 100% is checked once again. Consider removing this second check as it has already been performed.

Update: *Resolved in pull request #57 at commit ff8f850.*

N-10 Inconsistent Naming of File

The `deposit_tx.go` file implements the `DepositTx` type. Implementations of other types of transactions are located in the `tx_access_list.go`, `tx_blob.go`, `tx_dynamic_fee.go`, and `tx_legacy.go` files.

Consider renaming the `deposit_tx.go` file to `tx_deposit.go` for consistency.

Update: *Acknowledged, not resolved. The Mantle team stated:*

> *No issues. This file is related to the `depositTx` data struct.*

Conclusion

Mantle's op-geth upgrade to version 2 has introduced new capabilities to the rollup. These include native meta-transactions, MNT as a native token, and gas fee optimizations. The audit uncovered three issues of critical severity, in addition to several other issues of a lower-severity. We strongly recommend that Mantle implements more extensive QA and testing before going live to prevent potentially undiscovered vulnerabilities from being exploited. This is especially crucial in areas relating to the DOS-ing of the node and deposit transactions. We really appreciated working with the Mantle team as they were very supportive throughout the audit period and answered questions in a timely manner.