Mantle Network

Mantle L2 Rollup V2

Security Assessment Report

Version: 2.0

April, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Mantle Network's Version 2 smart contracts and related Golang software changes. For both areas, the review focused solely on the security aspects of the implementation, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. 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 Mantle Version 2 upgrade 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*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Mantle Network smart contracts.

Overview

Mantle is an EVM-compatible Layer 2 (L2) Optimistic Rollup network designed to scale Ethereum, utilising its strong security guarantees while reducing transaction cost and latency for users. Building on the framework designed by Optimism, another Layer 2 Optimistic Rollup, Mantle seeks to reduce transaction costs further by integrating the use of EigenLayer for Data Availability (DA).

The focus of this security assessment was on Mantle version 2, in which Mantle has transitioned to the Optimism's "Bedrock" system, reducing transaction costs and improving the upgradability of sections of Mantle's layer 2, reducing friction of future upgrades.

Additionally, Mantle has also fully integrated the use of their MNT native token as well as expanded native token bridging to support ERC-721 compliant NFTs.

Security Assessment Summary

Scope

The initial scope of this time-boxed review focused on the following:

- For smart contracts, commit 9d2f150 was reviewed with the scope strictly limited to files in `packages/contracts-bedrock/contracts/L1/` and `packages/contracts-bedrock/contracts/L2/` directories
- For the `op-geth` client, the scope was limited to changes made in PR#20
- For `op-node`, `op-batcher` and `op-proposer` the scope was limited to changes made in PR#72

During testing, the Mantle team has requested to further extend the scope of the engagement to include the following:

- For Mantle Network V2 repository:
  - PR#89
  - PR#95
  - PR#98
- For Mantle Network's OP Geth Client:
  - PR#31
  - PR#37

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

Approach

For smart contracts, the review was conducted on the files hosted in the Mantle Network V2 repository at commit 9d2f150.

For off-chain software, the initial review of changes made in PR#20 and PR#72 was performed on files hosted in Mantle Network's OP Geth Client and Mantle Network V2 repositories.

Review of the extended scope items was made at op-geth @ 4d05b2c and mantle-v2 @ e29d360 commits.

The manual code review section of the report is focused on identifying issues/vulnerabilities associated with the business logic implementation of the components in scope.

For smart contracts this includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

For the Golang libraries and modules, this includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime and use of the Ethereum protocol.

To support this review, the testing team used the following automated testing tools:

For Solidity based smart contracts:

- Mythril: `https://github.com/ConsenSys/mythril`
- Surya: `https://github.com/ConsenSys/surya`

For Golang code:

- `golangci-lint`: `https://github.com/golangci/golangci-lint`
- `semgrep-go`: `https://github.com/dgryski/semgrep-go`
- `go-geiger`: `https://github.com/jlauinger/go-geiger`
- native go fuzzing: `https://go.dev/doc/fuzz/`

Output for these automated tools is available upon request.

Coverage Limitations

Due to a 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 21 issues during this assessment. Categorised by their severity:

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

Detailed Findings

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

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

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

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

Summary of Findings

ID	Description	Severity	Status
MNTV2-01	Lack Of Balance Checks During BVM ETH Token Transfers	Critical	Resolved
MNTV2-02	`op-node` Fails To Handle Multiple Datastore Commitments In A Single Block	Critical	Closed
MNTV2-03	Token Theft Through Crafted Message on Unsafe `_target`	Critical	Resolved
MNTV2-04	Bridged MNT Tokens Can Be Stolen	Critical	Resolved
MNTV2-05	Use of Out of Support Golang Version With Known DoS Vulnerabilities	High	Resolved
MNTV2-06	Known DoS Vulnerability Present That is Fixed in Upstream Code	High	Closed
MNTV2-07	Use of Dependencies With Known Vulnerabilities	High	Closed
MNTV2-08	Unlimited Retries May Lead To DoS	High	Resolved
MNTV2-09	Insufficient Validation of Stored and Retrieved Data	Medium	Closed
MNTV2-10	Insecure Connections to MantleDA/EigenDA Remote Disperser and Retriever Nodes	Medium	Closed
MNTV2-11	Incorrect Logic on Conditional Statement	Medium	Resolved
MNTV2-12	`op-node` Retrieves Unrelated Datastores From MantleDA, Risk of DoS	Medium	Closed
MNTV2-13	Insufficient Validation of MantleDA Dispersal `ReferenceBlockNumber`	Low	Closed
MNTV2-14	Unrecoverable `MNT` Token	Low	Closed
MNTV2-15	Cross-Domain Message Gas Chosen By Relayer	Low	Closed
MNTV2-16	`DepositTx.effectiveGasPrice()` Always Returns Zero	Low	Closed
MNTV2-17	`sponsorAddress` Not Included In The Signed Data	Low	Closed
MNTV2-18	No Upgradeability Feature on `StandardBridge`	Informational	Closed
MNTV2-19	Deprecated `selfdestruct`	Informational	Resolved
MNTV2-20	Overwriting Previous Value	Informational	Resolved
MNTV2-21	Miscellaneous General Comments	Informational	Closed

MNTV2-01	Lack Of Balance Checks During BVM ETH Token Transfers		
Asset	`op-geth: core/state_transition.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Function `transferBVMETH()` does not check if `fromBalance` is sufficient before initiating the transfer.

Consider the following:

```go
func (st *StateTransition) transferBVMETH(ethValue *big.Int, rules params.Rules) {
	if !rules.IsMantleBVMETHMintUpgrade {
		return
	}
	var ethRecipient common.Address
	if st.msg.To != nil {
		ethRecipient = *st.msg.To
	} else {
		ethRecipient = crypto.CreateAddress(st.msg.From, st.evm.StateDB.GetNonce(st.msg.From))
	}
	if ethRecipient == st.msg.From {
		return
	}

	fromKey := getBVMETHBalanceKey(st.msg.From)
	toKey := getBVMETHBalanceKey(ethRecipient)

	fromBalanceValue := st.state.GetState(BVM_ETH_ADDR, fromKey)
	toBalanceValue := st.state.GetState(BVM_ETH_ADDR, toKey)

	fromBalance := fromBalanceValue.Big()   // @audit fromBalance converted into *big.Int - can be negative
	toBalance := toBalanceValue.Big()

	fromBalance = new(big.Int).Sub(fromBalance, ethValue)   // @audit subtracting ethValue from fromBalance - can go into
	↪  negative, no checks to see if the balance is sufficient
	toBalance = new(big.Int).Add(toBalance, ethValue)

	st.state.SetState(BVM_ETH_ADDR, fromKey, common.BigToHash(fromBalance))     // @audit BigToHash() returns an absolute
	↪  value, adjusting fromBalance and not reflecting deduction of funds
	st.state.SetState(BVM_ETH_ADDR, toKey, common.BigToHash(toBalance))

	st.generateBVMETHTransferEvent(st.msg.From, ethRecipient, ethValue)
}
```

Due to the `fromBalanceValue` being a `*big.Int`, it can go into a negative value. The `BigToHash()` call that follows, ignores the sign and returns an absolute value.

As such, balance transfer of any amount, regardless of the available `fromBalance`, will be successful, resulting in significant loss of funds. Subsequently, the `fromBalance` will also be changed to a positive value.

This allows a malicious actor to obtain an arbitrarily large BVM ETH balance and, if the exploit is unnoticed in order to pause withdrawals, drain all ETH from the L1 bridge.

Recommendations

Implement balance checks to ensure `fromBalance` is sufficient before initiating transfer of funds.

Resolution

The finding has been resolved in Pull#42.

MNTV2-02	op-node Fails To Handle Multiple Datastore Commitments In A Single Block		
Asset	mantle-v2: op-node/rollup/derive/calldata_source.go		
Status	**Closed:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`dataFromMantleDa()` only ever returns frames from at most one datastore/blob per L1 block. It fails to handle multiple datastore commitments in a single block.

While the data itself is kept offchain by EigenDA/MantleDA nodes, there are onchain commitments of each datastore/blob which the node uses to know what `dataStoreId` to request from EigenDA network. The node monitors the events `event ConfirmDataStore(uint32 dataStoreId, bytes32 headerHash)` from `datalayr/contracts/datalayr-contracts/src/contracts/core/DataLayrServiceManager.sol:87`.

The `dataFromMantleDa()` function of `op-node/rollup/derive/calldata_source.go`, which is supposed to loop through all logs emitted in a block, returns as soon as it gets a valid log, resulting in any later logs being ignored — see line [**236**] from the snippet code below.

```
205  for _, receipt := range receipts {
         for _, rLog := range receipt.Logs {
207        if strings.ToLower(rLog.Address.String()) != strings.ToLower(config.DataLayrServiceManagerAddr) {
             continue
209        }
           if rLog.Topics[0] != ConfirmDataStoreEventABIHash {
211          continue
           }
213        if len(rLog.Data) > 0 {
             err := confirmDataStoreArgs.UnpackIntoMap(dataStoreData, rLog.Data)
215          if err != nil {
               log.Error("Unpack data into map fail", "err", err)
217            continue
             }
219          if dataStoreData != nil {
               dataStoreId := dataStoreData["dataStoreId"].(uint32)
221            log.Info("Parse confirmed dataStoreId success", "dataStoreId", dataStoreId, "address", rLog.Address.String())
               daFrames, err := syncer.RetrievalFramesFromDa(dataStoreId - 1)
223            if err != nil {
                 log.Error("Retrieval frames from mantleDa error", "dataStoreId", dataStoreId, "err", err)
225              continue
               }
227            log.Info("Retrieval frames from mantle da success", "daFrames length", len(daFrames), "dataStoreId", dataStoreId)
               err = rlp.DecodeBytes(daFrames, &out)
229            if err != nil {
                 log.Error("Decode retrieval frames in error", "err", err)
231              continue
               }
233            metrics.RecordParseDataStoreId(dataStoreId)
               log.Info("Decode bytes success", "out length", len(out), "dataStoreId", dataStoreId)
235          }
             return out
237        }
         }
239    }
       return out
```

The issue could potentially occur if a few datastore commitment transactions end up in the same block. The nodes may then never be able to retrieve that data from the data-availability layer. Because the existing design expects all EigenDA consumers (e.g. other rollups and dapps) to submit confirmations to the `DataLayrServiceManager` contract, it is highly likely that this will be eventually triggered in a production environment, without requiring malicious activity.

For the verifiers/validators this means they either build a chain that is different to the sequencer (if those disappeared transactions "never happened") or there are blocks that never go from "unsafe" to "safe". Sequencer would also miss out on confirming those own blocks as "safe".

Recommendations

Modify the implementation to append all transaction data to `out` variable, e.g. `out = append(out, tx.Data())` and return it at the end of the function, when the loop is complete.

Resolution

The finding has been closed with the following comment from the development team:

> *"Only one commitment in a block (expected)."*

MNTV2-03	Token Theft Through Crafted Message on Unsafe `_target`		
Asset	`contracts: L1/L1CrossDomainMessenger.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Function `relayMessage()` utilises a low-level call to `_target` address with a crafted `_message` data on line [**254**].

```
bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
```

Consider cases where a malicious actor utilises this feature by calling an unsafe target, where the call will have `L1CrossDomainMessenger` as the `msg.sender`.

One of the most potent issues would be calling `IERC20(L1_MNT_ADDRESS).transfer()` to transfer the contract's token balance to an address controlled by the malicious actor. Since the contract holds token balance and the call is made by the token owner, then the transaction will succeed.

Recommendations

Consider utilising function `_isUnsafeTarget()` to filter out unsafe `_target` addresses by allowing the contract administrator to add blacklisted addresses dynamically.

This will allow for a flexible prevention should a future attack is identified.

Resolution

The changes made in Pull#98 resolve this issue by disallowing direct calls to `L1_MNT_ADDRESS` on `OptimismPortal.sol` and `L2ToL1MessagePasser.sol`.

MNTV2-04	Bridged MNT Tokens Can Be Stolen		
Asset	`contracts: L1/OptimismPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Note: This issue was disclosed by the Mantle team during the review.

When withdrawing from Mantle's network, it is possible to make a contract call on Ethereum Layer 1 as part of the withdrawal, because of this a user can drain the MNT tokens held by the bridging contract that belong to other users.

This is possible because the bridging process is handled by the `OptimismPortal` contract which holds any layer 1 Mantle tokens it receives for bridging. Then when making a withdrawal, there is an arbitrary call that can be specified to any contract and so it is possible to use this arbitrary call to call `L1_MNT_ADDRESS.transfer()`, moving the tokens accrued in the contract from other user interactions.

Recommendations

Modify the arbitrary contract call to disallow calls to the `L1_MNT_ADDRESS` contract address.

Resolution

The development team has fixed this issue in Pull#98 by ensuring the arbitrary call made on finalizing withdrawals cannot be made to `L1_MNT_ADDRESS`.

MNTV2-05	Use of Out of Support Golang Version With Known DoS Vulnerabilities		
Asset	`mantle-v2: go.mod, op-*/Dockerfile, op-geth: go.mod, Dockerfile`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The offchain node software is built using Go compiler versions that are no longer in support and which contain known, relevant and high-severity vulnerabilities. This leaves Internet-accessible node software vulnerable to denial-of-service (DoS) attacks.

The offchain software under review is compiled using `go1.19.9` and `go1.20.x` (as defined in relevant Dockerfile and CI files for `op-batcher` , `op-proposer` , `op-node` , `op-geth`).

As described in the Go release policy, *"each major Go release is supported until there are two newer major releases"*. `go1.22.0` is the latest major release at the time of writing, meaning releases `1.20` and older no longer receive critical security updates.

As the Mantle's `op-node` and `op-geth` software is configured by default to communicate via P2P, DoS conditions are a real concern for individual verifier nodes. The risk to the sequencer appears to be limited, as it is surrounded by replica nodes and not directly accessible by untrusted connections.

The following vulnerabilities are associated with the compiler versions used and affect the node software under review:

- CVE-2023-39325 a.k.a. GO-2023-2102 - HTTP/2 rapid reset can cause excessive work in net/http
  - CVSS High severity
  - Severity is deemed accurate for MantleV2
  - Refer to `https://github.com/golang/go/issues/63417` for more technical details.
- CVE-2023-39326 a.k.a. GO-2023-2382 - Denial of service via chunk extensions in net/http:
  - CVSS Medium severity
  - Severity is deemed accurate for MantleV2
  - Refer to `https://go-review.googlesource.com/c/go/+/547335` for more technical details.
- CVE-2023-45287 a.k.a. GO-2023-2375 - Before Go 1.20, the RSA based key exchange methods in crypto/tls may exhibit a timing side channel
  - CVSS High severity
  - Severity is deemed lower for MantleV2, as most information sent over the Internet does not need to remain secret.

The versions used have been out of support and with known vulnerabilities for some time without being rectified, (`go1.19` has been out of support since August 2023), indicating inefficiencies in Mantle's internal maintenance and vulnerability monitoring processes.

Recommendations

Ensure software is compiled using the latest patch release of a currently supported Go compiler version, and that these updates are promptly applied to critical infrastructure.

Monitor advisory for relevant security patches and compiler releases. Re-examine any procedures used to apply these patches to critical centralised infrastructure and communicate security updates to the node operator community.

Introduce automated monitoring to alert when relevant security vulnerabilities are reported. Solutions could involve dependabot and govulncheck.

Resolution

The finding has been resolved in Pull#71 and Pull#148.

MNTV2-06	Known DoS Vulnerability Present That is Fixed in Upstream Code		
Asset	`op-geth: p2p/peer.go`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Mantle v2 `op-geth` software is a modified fork of Optimism op-geth, which is itself a fork of Go-Ethereum. Mantle v2 `op-geth` contains a P2P denial-of-service (DoS) vulnerability that has been publicly disclosed and fixed in the upstream Go-Ethereum.

This vulnerability is labelled CVE-2023-40591 and GHSA-ppjg-v974-84cm. The code changes made to fix this, detailed in go-ethereum#27887, have not been applied to Mantle op-geth.

This vulnerability was publicly disclosed some months ago in September 2023, indicating flaws with internal procedures to incorporate upstream fixes.

As the Mantle `op-node` and `op-geth` software is configured by default to communicate via P2P, DoS conditions are a real concern for individual verifier nodes. Fortunately, risk to the sequencer is mitigated, as it is surrounded by replica nodes and not directly accessible by untrusted connections. However disruption of the replicas can cause problems for Dapps, as can disruption of specific nodes that the dapps rely on.

In addition, exploitation of this vulnerability is reasonably straightforward because the relevant details have been publicly disclosed.

Recommendations

Ensure changes functionally equivalent to go-ethereum#27887 are applied to Mantle `op-geth`.

Improve monitoring of upstream releases for relevant security patches. Reexamine any procedures used to apply these patches to critical centralised infrastructure and communicate security releases to the node operator community.

Introduce automated monitoring to alert when relevant security vulnerabilities are disclosed in upstream code. Go-ethereum publishes their security vulnerabilities at `https://geth.ethereum.org/docs/vulnerabilities/vulnerabilities.json` and `https://github.com/ethereum/go-ethereum/security/advisories?state=published`. As of this writing, the Optimism Foundation does not appear to have a dedicated vulnerability disclosure feed, and previous disclosures have been announced on their blog.

Consider implementing alerting for any pauses to Optimism chains, which would indicate a critical issue that may also apply to Mantle. This is especially useful if Mantle is not included in private disclosures. Optimism has specified a `SuperchainConfig` contract, described in this blog post, that can provide a global signal to pause.

Given upstream policies to introduce security patches silently,[1] it can be safer to more closely follow the upstream fork and incorporate seemingly innocuous changes.

If possible, consider collaborating with the Optimism and Go-Ethereum teams so that Mantle is notified of vulnerabilities prior to public disclosure.

[1]Detailed in `https://geth.ethereum.org/docs/developers/geth-developer/disclosures#why-silent-patches` and `https://github.com/ethereum-optimism/.github/blob/master/SECURITY.md`

Resolution

The issue has been closed with the following comment from the development team:

> *"No p2p connections between op-geth nodes (are supported) for now, we will fix it later."*

MNTV2-07	Use of Dependencies With Known Vulnerabilities		
Asset	`mantle-v2: go.mod`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The offchain node software is built using dependencies that contain known, relevant and high-severity vulnerabilities. This leaves internet-accessible node software vulnerable to denial-of-service (DoS) attacks.

As the Mantle `op-node` and `op-geth` software is configured by default to communicate via P2P, DoS conditions are a real concern for individual verifier nodes. The risk to the sequencer appears to be limited, as it is surrounded by replica nodes and not directly accessible by untrusted connections. However, disruption of the replicas may cause problems for Dapps which rely on them.

The following vulnerabilities are associated with dependency versions in use and affect the node software under review:

- CVE-2023-40583 a.k.a. GO-2023-2024 - `libp2p` nodes vulnerable to OOM (out of memory) DoS attack
  - CVSS High severity
  - Severity is deemed accurate for MantleV2
  - Refer to `https://github.com/advisories/GHSA-gcq9-qqwx-rgj3` for more technical details.
  - Advisory first published on 25 Aug, 2023. Dependency in mantle-v2: `github.com/libp2p/go-libp2p v0.25.1`
- CVE-2023-39533 a.k.a. GO-2023-2000 libp2p nodes vulnerable to resource exhaustion attack using large RSA keys
  - CVSS High Severity
  - Severity is deemed accurate for MantleV2
  - Refer to `https://github.com/libp2p/go-libp2p/security/advisories/GHSA-876p-8259-xjgg` for more technical details.
    Also note the following advice - *"To protect your application, it's necessary to update to these patch releases AND to use the updated Go compiler (1.20.7 or 1.19.12, respectively)"*.
  - Advisory first published on 9 Aug, 2023. Dependency in mantle-v2: `github.com/libp2p/go-libp2p v0.25.1`

Recommendations

Update affected dependencies to their latest versions that do not contain known, publicly disclosed vulnerabilities.

Ensure software is compiled using recent releases of critical dependencies that do not contain known vulnerabilities, and that these updates are promptly applied to critical infrastructure. The testing team notes that, at the time of reporting, the upstream OP stack uses a recent release of `go-libp2p v0.32.0`.

Monitor advisory for relevant security patches and compiler releases. Re-examine any procedures used to apply these patches to critical centralised infrastructure and communicate security releases to the node operator community.

Introduce automated monitoring to alert when relevant security vulnerabilities are reported. Solutions could involve dependabot and govulncheck.

Resolution

The issue has been acknowledged by the development team and closed with the following comment:

> *"Will fix it (in the future) using go-libp2p v0.32.0"*.

MNTV2-08	Unlimited Retries May Lead To DoS	
Asset	`mantle-v2: op-batcher/batcher/driver_da.go`	
Status	**Resolved:** See Resolution	
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `isRetry()` function always returns `true`. As this function is used by `loopRollupDa()`, which is fetching data from the channel's cache of pending transactions and rollup to MantleDA, a specific sequence of error conditions may result in an infinite loop, as the number of retries will not stop after predefined 10 attempts.

Consider the following:

```go
func (l *BatchSubmitter) isRetry(retry *int32) bool {
  *retry = *retry + 1
  l.metr.RecordRollupRetry(*retry)
  if *retry > DaLoopRetryNum {
    l.log.Error("rollup failed by 10 attempts, need to re-store data to mantle da")
    *retry = 0
    l.state.params = nil
    l.state.initStoreDataReceipt = nil
    l.metr.RecordDaRetry(1)
    return true   // @audit should be 'false' - otherwise `loopRollupDa()` will never return 'false' either, as `isRetry()`
      ↪ technically always succeeds.
  }
  time.Sleep(5 * time.Second)
  return true
}
```

And the following snippet of `loopRollupDa()`:

```go
for {
  // ...snip...

  err := sendTx()
  if err != nil {
    if l.isRetry(&retry) {
      continue
    }
    return false, err
  }
}
```

The above, on error condition, will loop indefinitely since `isRetry()` always returns `true`.

Recommendations

Change implementation of `isRetry()` to return `false` when max number of attempts, specified by `DaLoopRetryNum`, has been reached.

Resolution

The finding has been resolved in Pull#52.

MNTV2-09	Insufficient Validation of Stored and Retrieved Data		
Asset	`mantle-v2: op-batcher/batcher/driver_da.go, op-node/rollup/derive/calldata_source.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `op-batcher` and `op-node` do not appear to verify that the data dispersed (stored) by EigenDA is as expected, or that the data retrieved from the platform matches that which was stored.

As the current data availability implementation involves centralised points of trust (e.g. a hosted disperser) and fraud proofs are not currently active, trustless data availability is not *yet* a critical priority.

Identifiers for the rollup data are retrieved from events published by the L1 EigenDA `DataLayrServiceManager` contract at `op-node/rollup/derive/calldata_source.go:199`. This `confirmDataStore` event consists of an enumerable `dataStoreID` identifying a unique data store and a `headerHash`, which is a hash fingerprint of the following structure:

```
// @param header is the summary of the data that is being asserted into DataLayr,
type DataStoreHeader struct {
    KzgCommit      [64]byte
    Degree         uint32
    NumSys         uint32
    NumPar         uint32
    OrigDataSize   uint32
    Disperser      [20]byte
    LowDegreeProof [64]byte
}
```

The `op-node` never uses the `headerHash` when validating the retrieved frame data. The current implementation trusts that EigenDA and the retriever have not modified the data, and are presenting the same data to all entities. A malicious or compromised retriever can present different data to different nodes, resulting in an inconsistent view of the state — a consensus failure.

When fraud proofs become available, this can be used to have verifiers make a costly challenge against a state that is valid, or have the sequencer unwittingly publish state roots that cannot be reliably reconstructed.

Recommendations

Retrieved data should be verified against its KZG Commitment and the trust-path followed back to the value stored on the L1 chain. With the current implementation, this could be checked against the `DataStoreHeader` hash-root stored on-chain. This is emitted as the `headerHash` argument in the `confirmDataStore` event.

The sequencer should ensure data required to reconstruct the state is properly confirmed and available before publishing the associated state root onto L1.

The testing team notes that the EigenDA interface has apparently changed in more recent iterations. So validation using the existing interface may be less applicable.

Resolution

The finding has been closed with the following comment from the development team:

> *"Acknowledged, but currently there is no immediate fix. Because MantleDA is currently an exclusive component of Mantle, and the validity of the data is ensured by the mechanism of MantleDA, other components unconditionally trust the correctness of the data."*

MNTV2-10	Insecure Connections to MantleDA/EigenDA Remote Disperser and Retriever Nodes		
Asset	mantle-v2: `op-node/rollup/da/datastore.go`, `op-batcher/batcher/driver_da.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `op-node` and `op-batcher` connect to remote gRPC endpoints over an insecure connection, with TLS disabled. This allows a properly positioned malicious actor to perform MITM (man-in-the-middle) attacks to manipulate the content without detection, and corrupt the data being stored or retrieved from MantleDA/EigenDA.

These connections are defined at `op-node/rollup/da/datastore.go`:

```
conn, err := grpc.Dial(mda.Cfg.RetrieverSocket, grpc.WithTransportCredentials(insecure.NewCredentials()))
```

```
conn, err := grpc.Dial(mda.Cfg.MantleDaIndexerSocket, grpc.WithTransportCredentials(insecure.NewCredentials()))
```

And similarly for the `op-batcher` at `op-batcher/batcher/driver_da.go`:

```
conn, err := grpc.Dial(l.DisperserSocket, grpc.WithTransportCredentials(insecure.NewCredentials()))
```

A lack of transport layer security also allows an observer to read the message contents but, in this case, the communication contains no data that is important to keep confidential. Integrity of the data is instead critical.

In combination with MNTV2-09, this can allow an attacker to act as a malicious disperser and make data dispersal fail in a manner that may be difficult to pinpoint. It also makes exploits described in MNTV2-09 possible without the requirement that the retriever or disperser be compromised.

Recommendations

Ensure all remote connections occur over TLS (or some other form of transport-layer security), particularly for important centrally controlled infrastructure like the disperser node.

If insecure connections are important for testing, consider allowing it only via an obvious configuration input, like a CLI flag named `--unsafe-allow-insecure-grpc`.

Resolution

The finding has been closed with the following comment from the development team:

> *"Acknowledged, and there is no immediate fix. Repair will be considered in the future. MantleDA is only used within the internal network, and its security can be ensured.".*

MNTV2-11	Incorrect Logic on Conditional Statement		
Asset	`contracts: L1/OptimismPortal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The following code has an incorrect logic on the first two conditions `success && l1mntSuccess == false`, where the code will revert only on one variation, that is when `success == true` and `l1mntSuccess == false`.

```
if (success && l1mntSuccess == false && tx.origin == Constants.ESTIMATION_ADDRESS) {
```

The code should revert on all conditions, except when both `success` and `l1mntSuccess` are `true`.

Recommendations

Replace `success && l1mntSuccess == false` with `!success || !mntSuccess`.

Resolution

The issue has been fixed in Pull#95.

MNTV2-12	`op-node` Retrieves Unrelated Datastores From MantleDA, Risk of DoS		
Asset	`mantle-v2: op-node/rollup/derive/calldata_source.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `op-node` retrieves all DataStores whose roots are published to the `DataLayrServiceManager` contract. In production, EigenDA is intended for use by many platforms (not just Mantle). The `DataLayrServiceManager` records *all* data stores held by EigenDA/MantleDA. The current implementation retrieves *all* data, including that held for unrelated platforms.

This makes it possible to overload `op-node` instances resources and bandwidth, as they are forced to download much more data than expected. This is particularly impactful for any resource-constrained rollup nodes. It could be possible to intentionally attack the Mantle network by storing unrelated data on EigenDA.

Consider the following excerpt of `dataFromMantleDa()`:

```
205   for _, receipt := range receipts {
        for _, rLog := range receipt.Logs {
207       if strings.ToLower(rLog.Address.String()) != strings.ToLower(config.DataLayrServiceManagerAddr) {
            continue
209       }
          if rLog.Topics[0] != ConfirmDataStoreEventABIHash {
211         continue
          }
213       if len(rLog.Data) > 0 {
            err := confirmDataStoreArgs.UnpackIntoMap(dataStoreData, rLog.Data)
215         if err != nil {
              log.Error("Unpack data into map fail", "err", err)
217           continue
            }
219         if dataStoreData != nil {
              dataStoreId := dataStoreData["dataStoreId"].(uint32)
221           log.Info("Parse confirmed dataStoreId success", "dataStoreId", dataStoreId, "address", rLog.Address.String())
              daFrames, err := syncer.RetrievalFramesFromDa(dataStoreId - 1)
223           if err != nil {
                log.Error("Retrieval frames from mantleDa error", "dataStoreId", dataStoreId, "err", err)
225             continue
              }
227           // ...
```

The function loops through all logs emitted in a block, to find all `ConfirmDataStore` events emitted by the EigenDA `DataLayrServiceManager` contract. At line [**222**], any `dataStoreId` is used to request the corresponding data from the EigenDA network. There is no prior logic to first identify whether that `dataStoreId` is relevant to the Mantle network.

While later logic will ensure malformed data or invalid frames are ignored, this data must be first retrieved from EigenDA.

Recommendations

Any solution would also involve the `op-batcher` saving relevant `dataStoreIDs` where they can be obtained by `op-node` instances, so the `op-node` can determine which data stores are relevant without first needing to retrieve the whole data store content. A naive solution could involve submitting this onto L1 within `handleConfirmDataStoreReceipt()` at `op-batcher/batcher/driver_da.go:476`.

The sequencer should confirm that the data is available before publishing the associated state root.

Review the current EigenDA rollup integration documentation for a more in-depth exploration of possible designs.

Resolution

The finding has been closed with the following comment from the development team:

> *"MantleDA is an exclusive component of Mantle and only op-batcher has been authorised to submit data from one address (an EOA)."*

MNTV2-13	Insufficient Validation of MantleDA Dispersal `ReferenceBlockNumber`		
Asset	`mantle-v2: op-batcher/batcher/driver_da.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

At line [305], the `op-batcher` does not include a block number in its request to the remote EigenDA disperser, to encode the rollup data. The returned `ReferenceBlockNumber` is stored and used without further validation.

The `op-batcher` is responsible for submitting the data store roots onchain in an `initDataStore()` and `confirmDataStore()`. If this value is too old or corresponds to a block number that is in the future, data store submission transactions revert.

The corresponding onchain validation occurs in the `DataLayrServiceManager` contract, seen below:

```
require(
    referenceBlockNumber <= block.number, "DataLayrServiceManager.initDataStore: specified referenceBlockNumber is in future"
);

require(
    (referenceBlockNumber + BLOCK_STALE_MEASURE) >= uint32(block.number),
    "DataLayrServiceManager.initDataStore: specified referenceBlockNumber is too far in past"
);
```

As the current trust model involves trusting the centrally controlled EigenDA disperser for liveness (data submission does not occur if it is unavailable), a malicious disperser is unlikely. However, this becomes more readily exploitable if messages can be manipulated in transit due to MNTV2-10.

Recommendations

When submitting a MantleDA/EigenDA dispersal request, consider supplying your own known `ReferenceBlockNumber` and validate that the one returned by the remote disperser is not outside the bounds.

The testing team notes that the onchain data store submission flow has changed in more recent EigenDA iterations, and this may no longer be relevant after updating.

Resolution

The finding has been closed with the following comment from the development team:

> *"MantleDA is an internal component and the availability and validity of MantleDA data is ensured by internal mechanisms within MantleDA. Op-node only retrieves data processing that has been successfully confirmed by confirmDataStore. "Confirmed" represents final confirmation of the data state and cannot be tampered with."*

MNTV2-14	Unrecoverable `MNT` Token		
Asset	`contracts: L1/L1CrossDomainMessenger.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Function `relayMessage()` allows for the `_target` address to claim `MNT` tokens if `_mntValue` is non-zero, as defined in the following:

```
250 if (_mntValue!=0){
      mntSuccess = IERC20(L1_MNT_ADDRESS).approve(_target, _mntValue);
252 }
    xDomainMsgSender = _sender;
254 bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
    xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
256 if (_mntValue!=0){
      mntSuccess = IERC20(L1_MNT_ADDRESS).approve(_target, 0);
258 }
```

In the code snippet above, the `_target` is responsible for taking the `MNT` token from `L1CrossDomainMessenger`.

If the `_target` fails to do so, the approval would then be set to zero (lines [**256-258**]), meaning that the tokens are then locked forever in `L1CrossDomainMessenger` contract.

Recommendations

Consider creating a token withdrawal function. The amount of safely withdrawn tokens can be identified by accumulating the unclaimed `_mntValue`.

Resolution

The finding has been closed with the following comment from the development team:

> *"Users can replay relayMessage() without causing MNT to be locked inside L1CrossDomainMessenger."*

MNTV2-15	Cross-Domain Message Gas Chosen By Relayer		
Asset	`contracts: L2/L2CrossDomainMessenger.sol, L1/L1CrossDomainMessenger.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When sending a cross-domain message, the gas attached to the message call is chosen by the `relayer` as the execution logic may be determined by the quantity of gas given to the call. This can give the `relayer` limited control over how the call executes.

Currently, so long as the gas attached to the call defined by `gasleft() - RELAY_RESERVED_GAS` exceeds the `_minGasLimit` specified by the message sender, the call will be invoked.

Recommendations

To prevent a possible exploitation by the `relayer`, the gas sent in the message call should be more predictable. This can be achieved by attaching the exact amount specified by the message sender.

Resolution

The finding has been closed with the following comment from the development team:

> *"Under common circumstance, the users themselves will be the relayers. If some attackers try to front run the relayer transactions and deliberately fail the messages, the users can just try to replay."*

MNTV2-16	`DepositTx.effectiveGasPrice()` Always Returns Zero		
Asset	`op-geth: types/deposit_tx.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The second argument `baseFee *big.Int` in `effectiveGasPrice()` function is unused. As such, the return value is always zero.

```go
func (tx *DepositTx) effectiveGasPrice(dst *big.Int, baseFee *big.Int) *big.Int {
  return dst.Set(new(big.Int))
}
```

Note, the functionality appears to be only used when reading receipts for metrics and logs, and is not used in any processing, hence lower severity rating.

Recommendations

Modify implementation to account for the provided `baseFee` parameter and perform calculations accordingly.

Resolution

The finding has been closed with the following comment from the development team:

> *"The transaction doesn't execute up to this point."*

MNTV2-17	`sponsorAddress` Not Included In The Signed Data		
Asset	`op-geth: core/types/meta_transaction.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`checkSponsorSignature()` does not include `GasFeeSponsor` as a part of the signed data, yet during the signature verification, the signer derived from the signature is compared against `GasFeeSponsor` provided in the transaction parameters.

Consider the following code snippet from `checkSponsorSignature()`:

```go
// ... snip ...

        metaTxSignData := &MetaTxSignData{
                ChainID:        tx.ChainId(),
                Nonce:          tx.Nonce(),
                GasTipCap:      tx.GasTipCap(),
                GasFeeCap:      tx.GasFeeCap(),
                Gas:            tx.Gas(),
                To:             tx.To(),
                Value:          tx.Value(),
                Data:           metaTxParams.Payload,
                AccessList:     tx.AccessList(),
                ExpireHeight:   metaTxParams.ExpireHeight,
                SponsorPercent: metaTxParams.SponsorPercent,
        }

        gasFeeSponsorSigner, err = recoverPlain(metaTxSignData.Hash(), metaTxParams.R, metaTxParams.S, metaTxParams.V, true)
        if err != nil {
                return ErrInvalidGasFeeSponsorSig
        }
}

if gasFeeSponsorSigner != metaTxParams.GasFeeSponsor {  // @audit comparing signer derived from signature with an address from tx
    ↪  params, which were not signed and could be forged
        return ErrGasFeeSponsorMismatch
}
```

As there is no data identifying the sponsor in the hash, when only having the hash, anyone can claim to be the sponsor by simply signing the message and providing own address and relevant `r`, `s`, `v` values in `metaTxParams`.

Otherwise, if the `GasFeeSponsor` address was part of the hash, then only the actual sponsor's signature would be able to recover it to the correct address.

Note, considering little to no benefit for a potential attacker, who would need to pay gas on behalf of a user to potentially receive a small fee in return, the issue is rated with a lower severity.

Recommendations

Include `metaTxParams.GasFeeSponsor` as a part of `MetaTxSignData` to be able to verify the signer.

Resolution

The development team has acknowledged the issue and decided not to take further action due to a low risk and an unlikely exploitation, which would not yield benefits to potential attackers.

MNTV2-18	No Upgradeability Feature on `StandardBridge`
Asset	`contracts: L1/L1StandardBridge.sol, L2/L2StandardBridge.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Both contracts `L1StandardBridge` and `L2StandardBridge` do not have upgradeability feature, unlike other contracts in `L1` or `L2` directories.

This is likely due to both contracts deriving non-upgradeable `StandardBridge`, however the lack of upgradeability feature prevents future improvement and bug fixes on the contracts.

Recommendations

Ensure this behaviour is intentional, understood and documented.

Resolution

The finding has been closed with the following comment from the development team:

> *"L1StandardBridge will use L1ChugSplashProxy.sol as the proxy and can upgrade. L2StandardBridge will use Proxy.sol as the proxy and can upgrade."*

MNTV2-19	Deprecated `selfdestruct`
Asset	`contracts: L2/L2ToL1MessagePasser.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Function `burn()` removes the contract's native token balance by sending the balance to a new contract, then self-destructs the new contract, locking the native token permanently on the destroyed contract.

To execute the functionality, the contract uses contract `Burner` in library `Burn`. The contract `Burner` uses `selfdestruct()` that will soon be deprecated as per EIP-4758.

Recommendations

Consider removing the use of `selfdestruct()` for future compatibility.

Resolution

The finding has been resolved in Pull#122.

MNTV2-20	Overwriting Previous Value
Asset	`contracts: L1/L1CrossDomainMessenger.sol, L2/L2CrossDomainMessenger.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The code on line [**257**] overwrites the value of `mntSuccess` assigned on line [**251**], while the previous value is never evaluated.

```
249  bool mntSuccess = true;
     if (_mntValue!=0){
251      mntSuccess = IERC20(L1_MNT_ADDRESS).approve(_target, _mntValue);
     }
253  xDomainMsgSender = _sender;
     bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
255  xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
     if (_mntValue!=0){
257      mntSuccess = IERC20(L1_MNT_ADDRESS).approve(_target, 0);
     }
259  if (success && mntSuccess) {
```

Assuming that `_mntValue` is nonzero, the code on line [**259**] only evaluates the new value of `mntSuccess` and not the initial one.

Note that the same issue also occurs on `L2CrossDomainMessenger` contract.

Recommendations

Consider storing both values on two different variables if they are meant for evaluation. Otherwise, the first value of `mntSuccess` does not need to be stored.

Resolution

The finding has been resolved in Pull#98.

MNTV2-21	Miscellaneous General Comments
Asset	All code in scope
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Misleading Natspec Comment**

   Related Asset(s): L2/L2StandardBridge.sol

   The Natspec comment relating to the `L2StandardBridge` contract's handling of ERC20 tokens is misleading. It states:

   "The L2StandardBridge is responsible for transfering ETH and ERC20 tokens between L1 and L2. In the case that an ERC20 token is native to L2, it will be escrowed within this contract. If the ERC20 token is native to L1, it will be burnt."

   This is incorrect, `MNT`, which can be considered an ERC20 on L1 and is handled by the `L2StandardBridge` is not escrowed within the `L2StandardBridge` contract. `MNT` sent from L2 to L1 are forwarded to the `CrossDomainMessenger` which then forwards it to the `L2ToL1MessagePasser` contract where it can be burnt at a later date. Likewise, any ETH (which is an ERC20 on Mantle L2) sent from L2 to L1 is also forwarded to the same contracts and is then immediately burnt. Finally, if an ERC20 is an `OptimismMintableERC20` then it is not transferred to the `L2StandardBridge` contract and is instead burnt.

   The Natspec comment should be updated to reflect the variety of handling behaviours.

2. **L2StandardBridge Event Backward Compatibility Limited**

   Related Asset(s): L2/L2StandardBridge.sol

   Care has been taken with designing the new `L2StandardBridge` contract to ensure it emits the same events on token transfers as the older contract design, however when a deposit fails there is no `DepositFailed` event emission that was historically emitted on failed ERC20 deposits.

   This event could be used by third parties to track deposits and should be emitted for consistency with the old bridge design.

   Consider adding the `DepositFailed` event back to the `L2StandardBridge` contract to maintain event continuity between versions.

   Alternatively highlight this missing event so that any integrating services can be aware this event is no longer emitted by the bridge.

3. **Magic Numbers**

   Related Asset(s): L1/OptimismPortal.sol, L2/GasPriceOracle.sol

   Some contracts include hardcoded values which are not always adequately explained. Observed instances were: `OptimismPortal` line [**208**], `GasPriceOracle` [**177-179**].

   It is recommended to store these values as named constants to improve readability.

4. **Ensure Bindings Bytecode Is Up-To-Date**

   Related Asset(s): L2/*

   Mantle's L2 relies upon predeployed contracts that are hardcoded into the system via the Mantle nodes. These predeployed addresses are updated via `op-bindings` on upgrading to Mantle V2, it is advised therefore to ensure that all binding bytecodes are checked to ensure they reflect the newest version available for each underlying contract.

5. **No Universal Source Of Predeployed Addresses**

   Related Asset(s): L2/L2ERC721Bridge.sol, L2/L1Block.sol

   Some contracts rely on the constructor arguments given during deployment to set predeployed addresses or constants set within the contract. Other contracts such as `L2StandardBridge` and `BVM_ETH` make use of the `Predeploys` library meaning there are different methods utilised to set key addresses, making errors more likely.

   Employing a universal source of predeployed addresses to ensure these values are consistent between different files would help prevent future deployment errors.

6. **`GasPriceOracle` Emits Wrong Event Argument**

   Related Asset(s): L2/GasPriceOracle.sol

   When setting the token ratio via `setTokenRatio()`, the event `TokenRatioUpdated` emits the new token ratio for both arguments due to a mistake on line [**85**] setting `previousTokenRatio` to the wrong value.

   Changing line [**85**] to `uint256 previousTokenRatio = tokenRatio;` will fix this issue.

7. **Arithmetic Over/Underflow**

   Related Asset(s): L1/L2OutputOracle.sol

   Function `latestOutputIndex()` returns `l2Outputs.length - 1`. This code will produce an `Arithmetic over/underflow` error if `l2Outputs.length` is zero.

   Consider reverting with a clearer error message when `l2Outputs.length` is zero.

8. **Typos**

   Related Asset(s): L1/OptimismPortal.sol

   On line [**32**]: *"whcih"* should be *"which"*

   On lines [**72-73**]: *"If the of this variable* should be *"If the value of this variable"*

9. **Duplicated Information on event `Paused` and `Unpaused`**

   Related Asset(s): L1/OptimismPortal.sol

   Functions `pause()` and `unpause()` are accessible only by an actor with `GUARDIAN` role (immutable) described in the contract constructor. Therefore, the field `account` in the event `Paused` and `Unpaused` is always `GUARDIAN`. Emitting `account` is redundant, because that information that can be inferred from the code.

   Consider removing `account` from event `Paused` and `Unpaused`.

10. **Address `TODO` comments**

    Related Asset(s): core/mantle_upgrade.go

    Revisit all `TODO` comments to ensure that all key design decisions have been made and there are no outstanding items that could be considered critical.

11. **Confusing parameter variable name**

    Related Asset(s): op-batcher/metrics/metrics.go

    Consider renaming the parameter variable name to `_blockNumber` in the following function:

```go
func (m *Metrics) RecordInitReferenceBlockNumber(dataStoreId uint32) {  // @audit INFO: confusing parameter name
  m.recordReferenceBlockNumber.Set(float64(dataStoreId))
}
```

12. **Insufficient Test Coverage**

    It was observed that complex parts of the system, particularly the offchain components, are missing comprehensive test coverage. It is strongly advised to improve code coverage and dedicate ample time to develop variety of tests covering both positive paths and edge case scenarios.

Recommendations

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

Resolution

The development team has acknowledged the above findings and advised that they will be addressed at a later date.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Forge` framework was used to perform these tests and the output is given below.

```
Running 1 test for test/Basic.t.sol:BasicTest
[PASS] test_VariablesWork() (gas: 2235)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.33ms

Running 3 tests for test/Counter2.t.sol:Counter2Test
[PASS] testIncrement() (gas: 12150)
[PASS] testSetNumber(uint256) (runs: 1000, u: 12075, ~: 12234)
[PASS] test_VariablesWork() (gas: 2235)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 76.62ms

Running 2 tests for test/L2/BVM_ETH.t.sol:BVM_ETH_Tests
[PASS] test_burn() (gas: 32597)
[PASS] test_mint() (gas: 23518)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 108.39ms

Running 2 tests for test/L2/L1Block.t.sol:L1Block_Tests
[PASS] test_setL1BlockValues_failure() (gas: 30973)
[PASS] test_setL1BlockValues_success() (gas: 197935)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 113.58ms

Running 2 tests for test/L2/CrossDomainOwnable.t.sol:CrossDomainOwnable2Test
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_set(uint256) (runs: 1000, u: 36079, ~: 36836)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 117.43ms

Running 3 tests for test/L2/CrossDomainOwnable.t.sol:CrossDomainOwnable3Test
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_set_local(uint256) (runs: 1000, u: 36984, ~: 37721)
[PASS] test_set_not_local(uint256) (runs: 1000, u: 45274, ~: 46210)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 222.25ms

Running 2 tests for test/L2/CrossDomainOwnable.t.sol:CrossDomainOwnableTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_set(uint256) (runs: 1000, u: 36444, ~: 37420)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 101.89ms

Running 3 tests for test/L2/L2ERC721Bridge.t.sol:L2ERC721BridgeTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_bridgeERC721(uint256,uint32,bytes) (runs: 1000, u: 221618, ~: 222508)
[PASS] test_finalizeBridgeERC721(uint256,bytes,uint32) (runs: 1000, u: 172448, ~: 173834)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 834.67ms

Running 4 tests for test/L2/BaseFeeVault.t.sol:BaseFeeVault_Tests
[PASS] test_receive_success() (gas: 20923)
[PASS] test_withdraw_balance_too_low() (gas: 25237)
[PASS] test_withdraw_fuzz(uint256,uint256) (runs: 1000, u: 205773, ~: 206236)
[PASS] test_withdraw_success() (gas: 192414)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 977.37ms

Running 4 tests for test/L2/L1FeeVault.t.sol:L1FeeVault_Tests
[PASS] test_receive_success() (gas: 20923)
[PASS] test_withdraw_balance_too_low() (gas: 25237)
[PASS] test_withdraw_fuzz(uint256,uint256) (runs: 1000, u: 205800, ~: 206236)
[PASS] test_withdraw_success() (gas: 192414)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.11s

Running 9 tests for test/L2/GasPriceOracle.t.sol:GasPriceOracle_Tests
[PASS] test_getL1Fee_fuzz(bytes,uint256,uint256) (runs: 1000, u: 209181, ~: 209734)
[PASS] test_getL1GasUsed_fuzz(bytes) (runs: 1000, u: 33933, ~: 31167)
[PASS] test_setOperator_success() (gas: 44879)
[PASS] test_setOperator_wrong_caller() (gas: 15755)
```

```
[FAIL. Reason: log != expected log] test_setTokenRatio_Vuln() (gas: 69970)
[PASS] test_setTokenRatio_wrong_caller() (gas: 43283)
[PASS] test_transferOwnership_success() (gas: 42861)
[PASS] test_transferOwnership_wrong_caller() (gas: 15735)
[PASS] test_transferOwnership_zero_address() (gas: 20347)
Test result: FAILED. 8 passed; 1 failed; 0 skipped; finished in 868.92ms

Running 4 tests for test/L2/SequencerFeeVault.t.sol:SequencerFeeVault_Tests
[PASS] test_receive_success() (gas: 20923)
[PASS] test_withdraw_balance_too_low() (gas: 24825)
[PASS] test_withdraw_fuzz(uint256,uint256) (runs: 1000, u: 205793, ~: 206236)
[PASS] test_withdraw_success() (gas: 192414)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.59s

Running 4 tests for test/L2/L2ToL1MessagePasser.t.sol:L2ToL1MessagePasserTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_burn(uint256) (runs: 1000, u: 106561, ~: 106561)
[PASS] test_initiateWithdrawal(uint256,uint256,address,uint256,bytes) (runs: 1000, u: 310969, ~: 310877)
[PASS] test_receive_burn(uint256) (runs: 1000, u: 112213, ~: 112213)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.86s

Running 6 tests for test/L1/SystemConfig.t.sol:SystemConfigTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_setBaseFee(uint256) (runs: 1000, u: 30312, ~: 30533)
[PASS] test_setBatcherHash(bytes32) (runs: 1000, u: 30509, ~: 30509)
[PASS] test_setGasConfig(uint256,uint256) (runs: 1000, u: 37603, ~: 37978)
[PASS] test_setGasLimit(uint64) (runs: 1000, u: 28168, ~: 22955)
[PASS] test_setUnsafeBlockSigner(address) (runs: 1000, u: 33520, ~: 33520)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 867.68ms

Running 28 tests for test/L2/L2StandardBridge.t.sol:L2StandardBridge_Tests
[PASS] test_bridgeERC20_mismatched_tokens() (gas: 33579)
[PASS] test_bridgeERC20_not_EOA() (gas: 23257)
[PASS] test_bridgeERC20_not_ETH() (gas: 42505)
[PASS] test_bridgeERC20_not_MNT() (gas: 43209)
[PASS] test_bridgeERC20_sender_has_no_tokens() (gas: 36693)
[PASS] test_bridgeERC20_success() (gas: 208799)
[PASS] test_bridgeETH_SC_call() (gas: 191258)
[PASS] test_bridgeETH_success() (gas: 210626)
[PASS] test_finalizeBridgeERC20_cross_domain_message_sender_not_set() (gas: 31847)
[PASS] test_finalizeBridgeERC20_native_token_success() (gas: 222675)
[PASS] test_finalizeBridgeERC20_non_mintable_token_failure() (gas: 155151)
[PASS] test_finalizeBridgeERC20_non_mintable_token_success() (gas: 385959)
[PASS] test_finalizeBridgeERC20_wrong_caller() (gas: 21178)
[PASS] test_finalizeBridgeETH_cross_domain_message_sender_not_set() (gas: 29425)
[PASS] test_finalizeBridgeETH_success() (gas: 314335)
[PASS] test_finalizeBridgeETH_wrong_caller() (gas: 18869)
[PASS] test_finalizeBridgeMNT_cross_domain_message_sender_not_set() (gas: 29491)
[PASS] test_finalizeBridgeMNT_success() (gas: 202668)
[PASS] test_finalizeBridgeMNT_wrong_caller() (gas: 18867)
[PASS] test_l1TokenBridge() (gas: 12689)
[PASS] test_withdrawTo_ERC20_non_zero_msgvalue() (gas: 82561)
[PASS] test_withdrawTo_ERC20_success() (gas: 209353)
[PASS] test_withdrawTo_ETH_success() (gas: 218135)
[PASS] test_withdrawTo_MNT_mismatch_amount() (gas: 21584)
[PASS] test_withdrawTo_MNT_success() (gas: 162394)
[PASS] test_withdrawTo_insufficient_allowance() (gas: 35476)
[PASS] test_withdrawTo_no_ETH() (gas: 66389)
[PASS] test_withdrawTo_non_zero_msgvalue() (gas: 65862)
Test result: ok. 28 passed; 0 failed; 0 skipped; finished in 89.09ms

Running 5 tests for test/WETH9.t.sol:WETH9Test
[PASS] test_VariablesWork() (gas: 2257)
[PASS] test_deposit(uint256) (runs: 1000, u: 42933, ~: 44556)
[PASS] test_deposit_withdraw(uint256,uint256) (runs: 1000, u: 54573, ~: 55655)
[PASS] test_transfer(uint256,uint256) (runs: 1000, u: 72622, ~: 75207)
[PASS] test_transferFrom(uint256,uint256) (runs: 1000, u: 88029, ~: 90065)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 1.41s

Running 4 tests for test/L2/L2CrossDomainMessenger.t.sol:L2CrossDomainMessengerTestL2
```

```
[PASS] test_relayMessage_failed_replayed(uint16,uint240,uint256,uint256) (runs: 1000, u: 387974, ~: 389968)
[PASS] test_relayMessage_success(uint16,uint240,uint256,uint256) (runs: 1000, u: 328434, ~: 329566)
[PASS] test_sendMessage_eth(uint256,address,bytes,uint32) (runs: 1000, u: 288629, ~: 288243)
[PASS] test_sendMessage_noEth(address,bytes,uint32) (runs: 1000, u: 112784, ~: 112121)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 7.97s

Running 5 tests for test/L1/OptimismPortal.t.sol:OptimismPortalTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_depositTransaction(uint256,uint256,uint32,bool,bytes) (runs: 1000, u: 2379586, ~: 519433)
[PASS] test_donateETH(uint256) (runs: 1000, u: 16907, ~: 16907)
[PASS] test_pause_unpause() (gas: 32054)
[PASS] test_received(uint256) (runs: 1000, u: 128254, ~: 128703)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 43.99s

Running 2 tests for test/L1/ResourceMetering.t.sol:ResourceMeteringTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_set(uint256,uint256,uint64) (runs: 1000, u: 386047, ~: 87373)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.82s

Running 9 tests for test/L1/L1CrossDomainMessenger.t.sol:L1CrossDomainMessengerTest
[PASS] test_VariablesWork() (gas: 2257)
[FAIL. Reason: <no data> Counterexample:calldata=0x9a5b 905b 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
      ↪  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
      ↪  0000 0000 0000 0000 0000 0000 0000 0000 0000 0001, args=[0, 0, 1]] test_relayMessage_L1MNT_Vuln(uint16,uint240,uint256)
      ↪  (runs: 0, u: 0, ~: 0)
[PASS] test_relayMessage_failed(uint16,uint240,uint256,uint256) (runs: 1000, u: 354891, ~: 356964)
[PASS] test_relayMessage_failed_modified(uint16,uint240,uint256,uint256) (runs: 1000, u: 274242, ~: 276258)
[PASS] test_relayMessage_failed_reenter(uint16,uint240,uint256,uint256) (runs: 1000, u: 596260, ~: 598788)
[PASS] test_relayMessage_success(uint16,uint240,uint256,uint256) (runs: 1000, u: 293590, ~: 294647)
[PASS] test_relayMessage_success_token_stuck(uint16,uint240,uint256,uint256) (runs: 1000, u: 263019, ~: 264004)
[PASS] test_sendMessage(uint256,address,bytes,uint32) (runs: 1000, u: 906105, ~: 534977)
[PASS] test_sendMessage_noToken(address,bytes,uint32) (runs: 1000, u: 750356, ~: 372199)
Test result: FAILED. 8 passed; 1 failed; 0 skipped; finished in 52.84s

Running 3 tests for test/L1/L1ERC721Bridge.t.sol:L1ERC721BridgeTest
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_bridgeERC721(uint256,uint32,bytes) (runs: 1000, u: 858171, ~: 472548)
[PASS] test_finalizeBridgeERC721(uint256,bytes,uint32) (runs: 1000, u: 808169, ~: 476393)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 12.34s

Running 20 tests for test/L1/L1StandardBridge.t.sol:L1StandardBridgeTest
[PASS] testFail_finalizeBridgeERC20_insufficientDeposits(uint256,bytes,uint32) (runs: 1000, u: 1034813, ~: 816712)
[PASS] test_VariablesWork() (gas: 2235)
[PASS] test_bridgeERC20(uint256,uint32,bytes) (runs: 1000, u: 929035, ~: 558851)
[PASS] test_bridgeERC20To(uint256,uint32,bytes) (runs: 1000, u: 984642, ~: 560382)
[PASS] test_bridgeETH(uint32,bytes,uint256) (runs: 1000, u: 851516, ~: 401931)
[PASS] test_bridgeETHTo(uint32,bytes,uint256) (runs: 1000, u: 904732, ~: 404188)
[PASS] test_bridgeMNT(uint256,uint32,bytes) (runs: 1000, u: 838811, ~: 591339)
[PASS] test_bridgeMNTTo(uint256,uint32,bytes) (runs: 1000, u: 894695, ~: 593076)
[PASS] test_depositERC20(uint256,uint32,bytes) (runs: 1000, u: 987078, ~: 558891)
[PASS] test_depositERC20To(uint256,uint32,bytes) (runs: 1000, u: 976268, ~: 559767)
[PASS] test_depositETH(uint32,bytes,uint256) (runs: 1000, u: 818009, ~: 401479)
[PASS] test_depositETHTo(uint32,bytes,uint256) (runs: 1000, u: 841254, ~: 404062)
[PASS] test_depositMNT(uint256,uint32,bytes) (runs: 1000, u: 930991, ~: 591458)
[PASS] test_depositMNTTo(uint256,uint32,bytes) (runs: 1000, u: 1097743, ~: 593128)
[PASS] test_finalizeBridgeERC20(uint256,bytes,uint32) (runs: 1000, u: 958249, ~: 685713)
[PASS] test_finalizeBridgeETH(uint32,bytes,uint256) (runs: 1000, u: 64000, ~: 64354)
[PASS] test_finalizeERC20Withdrawal(uint256,bytes,uint32) (runs: 1000, u: 1023439, ~: 686791)
[PASS] test_finalizeETHWithdrawal(uint256,bytes) (runs: 1000, u: 49633, ~: 49898)
[PASS] test_finalizeMantleWithdrawal(uint256,bytes) (runs: 1000, u: 241103, ~: 242762)
[PASS] test_receive(uint256) (runs: 1000, u: 583072, ~: 584500)
Test result: ok. 20 passed; 0 failed; 0 skipped; finished in 68.45s

Running 6 tests for test/L1/L2OutputOracle.t.sol:L2OutputOracleTest
[PASS] testFail_propose_delete() (gas: 437937345)
[PASS] test_VariablesWork() (gas: 2257)
[PASS] test_deleteL2Outputs(uint256) (runs: 1000, u: 7842322, ~: 7834722)
[PASS] test_deleteL2Outputs_finalised(uint256) (runs: 1000, u: 7817064, ~: 7817064)
[PASS] test_proposeL2Output(bytes32) (runs: 1000, u: 111183, ~: 111183)
[PASS] test_proposeL2Output_multi(uint256) (runs: 1000, u: 8521915, ~: 8521890)
```

```
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 67.73s

Ran 23 test suites: 129 tests passed, 2 failed, 0 skipped (131 total tests)

Failing tests:
Encountered 1 failing test in test/L1/L1CrossDomainMessenger.t.sol:L1CrossDomainMessengerTest
[FAIL. Reason: <no data> Counterexample:calldata=0x9a5b 905b 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
      ↪  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
      ↪  0000 0000 0000 0000 0000 0000 0000 0000 0000 0001, args=[0, 0, 1]] test_relayMessage_L1MNT_Vuln(uint16,uint240,uint256)
      ↪  (runs: 0, u: 0, ~: 0)

Encountered 1 failing test in test/L2/GasPriceOracle.t.sol:GasPriceOracle_Tests
[FAIL. Reason: log != expected log] test_setTokenRatio_Vuln() (gas: 69970)

Encountered a total of 2 failing tests, 129 tests succeeded
```

Appendix B 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.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

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

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].