OpenZeppelin | security

Mantle V2 Solidity Contracts Audit

MANTLE

March 15, 2024

Table of Contents

Table of Contents 2

Summary 4

Scope 5

System Overview 7

Bridges 8

Messengers 8

Portal and Message Passer 8

Privileged Roles 10

Security Model and Trust Assumptions 10

Critical Severity 12

C-01 BVM_ETH and MNT Deposited in Messengers Can Be Stolen 12

Medium Severity 13

M-01 Cross Domain Messengers Can Fail in Relaying a Message 13

M-02 Gas Estimation Can Fail in finalizeWithdrawalTransaction 14

M-03 Unnecessary Payable Function Definition 15

Low Severity 16

L-01 Assets Might Get Stuck in Contracts 16

L-02 Unusual Upgradeability Patterns Are Adopted 17

L-03 Wrong Value Emitted in Event 18

L-04 Incomplete Docstrings 18

L-05 Floating and Multiple Pragma Directives Are Being Used 19

L-06 Unsafe ABI Encoding 20

L-07 Missing Docstrings 20

Notes & Additional Information 21

N-01 Unnecessary Boolean Values 21

N-02 Misleading Docstrings 22

N-03 Duplicated Getter Function 22

N-04 public Functions Can Be Declared as external 23

N-05 Code Style Inconsistency 23

N-06 Typographical Errors 23

N-07 Variables Naming Does Not Follow Solidity Style Guide 24

N-08 Use of Magic Constants 24

N-09 Usage of Single Step Ownership Transfer 24

N-10 Lack of Indexed Event Parameters 25
N-11 Lack of Security Contact 25
N-12 Unnecessary Cast 25
N-13 Unused Code 26
N-14 Addresses of Predeploys Are Not Ordered 26
N-15 Address Is Being Removed Twice 26
N-16 Predeployed Contracts Missing Custom Documentation Tag 27
N-17 Unused Import 27
N-18 Duplicate Event Emission 27

Conclusion 28

Summary

Type	L2 Rollup
Timeline	From 2024-01-31 To 2024-03-01
Languages	Solidity

Total Issues	29 (12 resolved, 1 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (3 resolved)
Low Severity Issues	7 (1 resolved)
Notes & Additional Information	18 (7 resolved, 1 partially resolved)

Scope

We audited the mantlenetworkio/mantle-v2 repository at commit e29d360.

In scope were the following files:

```
packages/contracts-bedrock/contracts
├── L1
│   ├── L1CrossDomainMessenger.sol
│   ├── L1ERC721Bridge.sol
│   ├── L1StandardBridge.sol
│   ├── L2OutputOracle.sol
│   ├── OptimismPortal.sol
│   ├── ResourceMetering.sol
│   └── SystemConfig.sol
└── L2
│   ├── BaseFeeVault.sol
│   ├── BVM_ETH.sol
│   ├── CrossDomainOwnable.sol
│   ├── CrossDomainOwnable2.sol
│   ├── CrossDomainOwnable3.sol
│   ├── GasPriceOracle.sol
│   ├── L1Block.sol
│   ├── L1FeeVault.sol
│   ├── L2CrossDomainMessenger.sol
│   ├── L2ERC721Bridge.sol
│   ├── L2StandardBridge.sol
│   ├── L2ToL1MessagePasser.sol
│   └── SequencerFeeVault.sol
```

In addition, we also performed a limited review of the contracts listed below as they are dependencies of above in-scope contracts. As a result, some low- and note-level issues were raised. However, a full audit of these files was not performed.

```
├── universal
│   ├── ERC721Bridge.sol
│   ├── StandardBridge.sol
│   ├── CrossDomainMessenger.sol
│   ├── OptimismMintableERC20.sol
│   ├── Semver.sol
│   └── FeeVault.sol
```

Finally, the following files were audited exclusively in terms of differences with the base commit bb0ff70:

```
├── deployment
│   ├── PortalSender.sol
│   ├── SystemDictator.sol
├── legacy
│   └── LegacyERC20MNT.sol
├── libraries
│   ├── Burn.sol
│   ├── Encoding.sol
│   ├── Hashing.sol
│   ├── Predeploys.sol
│   ├── Types.sol
```

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. As far as the differences are concerned, the most important one is the native currency used in L2 being changed from ETH to the Mantle Token (MNT). MNT is an ERC-20 token on the Ethereum mainnet becaue of which the code has to be adapted. Prior to going into the details of all the specific changes, it is worth summarizing what the Mantle v2 stack is and how it works. The system is based on two main directories of contracts, namely the `L1` and `L2` directories.

As the name suggests, the `L1` directory will contain contracts that manage the following:

- ERC-20, ERC-721, MNT, and ETH deposit into the L1 contracts.
- Event emission for L2 bridging finalization.
- Additional features to prove and finalize a withdrawal transaction (the inverse action of depositing into L2).
- Auxiliary contracts like the `ResourceMetering` contract to handle gas unit measurements according to EIP-1559, the `L2OutputOracle` contract that hosts finalized state roots of L2 blocks, and the `SystemConfig` contract which serves merely to retrieve system configurations of different parameters.

On the other hand, the `L2` directory will contain contracts that manage the following:

- Initialization of ERC-20, ERC-721, MNT, and ETH withdrawals.
- Finalization of ERC-20, ERC-721, MNT, and ETH deposits.
- Fee vaults where base fees, sequencer fees, and the l1 portion of the transaction fees are accumulated.
- Auxiliary contracts like the `GasPriceOracle`, where the relation of price between MNT and ETH is managed by a `tokenRatio` parameter and L1 block information can be retrieved thanks to the information stored in the `L1Block` contract, and the ownable contracts that are meant to manage cross-domain ownership interactions.

The three main parts of both `L1` and `L2` are the following:

Bridges

On L1, ERC-20 assets, MNT tokens, and ETH are managed through the `L1StandardBridge` contract, while ERC-721 assets are managed through the `L1ERC721Bridge` contract. On L2, equivalent contracts are found, namely the `L2StandardBridge` and `L2ERC721Bridge` contracts.

Both bridges have functions in common to initialize asset transfers to the opposite domain like the `bridgeMNT`, `bridgeERC20`, `bridgeETH` or `bridgeERC721` functions, and to finalize bridging from the opposite domain like the `finalizeBridgeERC20`, `finalizeBridgeMNT`, `finalizeBridgeETH` or `finalizeBridgeERC721`. ERC-20 and ERC-721 assets are locked within the bridge contracts, whereas MNT and ETH are transferred to the `L1CrossDomainMessenger` / `L2CrossDomainMessenger` contracts when bridging to the other domain, or are transferred from them when finalizing a bridge to the current domain.

Messengers

The `L1CrossDomainMessenger` and `L2CrossDomainMessenger` contracts are merely intermediaries between bridges and the `OptimismPortal` / `L2toL1MessagePasser`. Messengers do not distinguish between assets and the only thing they do is pass arbitrary encoded messages together with MNT or ETH. When bridging over to the other domain, the `sendMessage` functions are triggered, whereas when finalizing a bridge from another domain, the `relayMessage` functions are called. The target of any `sendMessage` call is the `OptimismPortal` / `L2toL1MessagePasser`, whereas the target of a `relayMessage` execution is usually a bridge contract.

Portal and Message Passer

The `OptimismPortal` (portal) contract is the final point in the L1 execution when depositing assets from L1 to L2, whereas the `L2toL1MessagePasser` contract is the final point for withdrawing from L2 to L1. These contracts emit events whose parameters are the encoded messages of the assets being transferred. One notable difference between the two is that `OptimismPortal` hosts the mechanism to prove and finalize any withdrawal transaction from L2-to-L1, while the `L2toL1MessagePasser` merely passes messages from L2-to-L1. The verification of withdrawal transactions is where L2 state roots are used to ensure that the withdrawal transaction has been effectively triggered on L2 first. The `OptimismPortal` and the `L2toL1MessagePasser` contracts are the ones effectively holding MNT and ETH on both domains.

The reason behind this is that one can skip the entire flow of passing from bridges to messengers and target them directly, thereby saving gas. However, it is a less user-friendly flow to follow and is also error-prone. Moreover, users can also only skip the bridges and send arbitrary data through the messengers. Messengers will pass the execution to the portal or to the message passer on L2. The main difference introduced by Mantle v2 is changing the native currency on L2 from ETH to MNT. This means that ETH is converted into an ERC-20 asset on L2, specifically into an instance of an `OptimismMintableERC20` token. This comes with some nuances:

- A deposit transaction transfers ERC-20 MNT into the L1 system, whereas on the L2, it is the native currency. MNT, as the native currency, is minted at the protocol level before the L2 finalization executes, as would be the case for ETH in the original Optimism code.
- ETH is transferred into the L1 system which is accounted for by an ERC-20 WETH token mint into the L2 system. WETH is first minted and then transferred to the initiator of L2 deposit finalization. The minting process is performed at the protocol level.

When bridging from L2 to L1:

- The native MNT is collected into the L2 system and burned at a later stage through a contract that self-destructs at construction time, effectively removing the native currency from circulation. The corresponding amount of the ERC-20 Mantle token is then released on the L1 system.
- The WETH is first burned on L2 and then released on L1 as native currency. In contrast to the minting process, the burning process happens at the contract level.

Since the MNT-to-ETH exchange rate fluctuates, in order to correctly account for transaction fees, a `tokenRatio` value has been introduced in the `GasPriceOracle` contract. The `tokenRatio` represents the value of MNT compared to ETH which enables fee values to be correctly calculated. This token ratio is managed at the client level and is meant to be adjusted by a trusted operator every time the price relation changes.

Mantle v2 introduces many other features which are implemented at the protocol level and are deemed out of scope of the current smart contract list. We recommend taking a look at the following official page for a more in-depth analysis of the new features.

Privileged Roles

There are several privileged actors within the system:

- In the `L2OutputOracle`, only the `CHALLENGER` address can revert state roots which have been already pushed, whereas only the `PROPOSER` address can add new roots.
- In the `OptimismPortal`, the `GUARDIAN` address can `pause/unpause` withdrawal verification and finalization.
- The `owner` of the `SystemConfig` contract can change the unsafe block signer, the entire configuration, and the `batcherHash`, `overhead`, `scalar`, `gasLimit`, and `baseFee` parameters.
- The `owner` of the `GasPriceOracle` contract can change the operator address, while the `operator` can change the token ratio. There are no sanity checks present that can ensure that the token ratio set by the operator is correct.
- The `DEPOSITOR_ACCOUNT` address can set L1 block values in the `L1Block` contract.

At any moment, it is assumed that all these special actors have been properly configured and that none of their private keys are compromised.

Security Model and Trust Assumptions

Cross-chain messaging and the asset bridging built on top of are integral parts of every L2 solution. However, the correctness of these cross-chain communications depends on the underlying node and client operations. As such, we assume that the emitted data for cross-chain communication is correctly relayed from one layer to the other. Given the increased complexity in differentiating between MNT and ETH (both on L1 and L2), significant amount of logic is handled at the protocol level. This makes both system contracts and clients have a bigger dependent relation compared to the original Optimism code. More precisely, there is the assumption that a deposit transaction will be executed exactly as requested. For a counter-example of how funds can get stuck due to some incorrect node relaying or client executions, please see the Appendix.

Furthermore, the token ratio that establishes a price relation for MNT-to-ETH is assumed to be manipulation-resistant as claimed in the official documentation.

Critical Severity

C-01 `BVM_ETH` and `MNT` Deposited in Messengers Can Be Stolen

In the `L2CrossDomainMessenger` contract, the `relayMessage` function will perform an arbitrary external call to `_target`. At the same time, the same function is expected to fail in the external call and has logic to handle such a case. If an external call fails, the `failedMessages[versionedHash]` mapping will be set to `true` and, at that point, anyone can retry the execution of the transaction.

When transferring ETH from L1 to L2, if the user went through the `L1StandardBridge` logic, the `BVM_ETH` will be minted and transferred to the `L2CrossDomainMessenger` on L2 and the `relayMessage` execution is then triggered to move those `BVM_ETH` to their final destination. Similarly, if the user went through the `L2StandardBridge` contract when transferring MNT from L2 to L1, the `OptimismPortal` contract will transfer MNT to the `L1CrossDomainMessenger` and execute the `relayMessage` function to finalize the MNT withdrawals.

In both cases, the messengers are expected to potentially fail. If that happens, ETH will be sitting in the `L2CrossDomainMessenger` contract and MNT will be sitting in the `L1CrossDomainMessenger` contract, waiting for anyone to retry the failed execution. This is where a malicious actor can steal all of the ETH or MNT. The attacker can initiate a `depositTransaction` through the `L1CrossDomainMessenger` contract which will be passed to `relayMessage` with the `_target` as the `BVM_ETH` contract and the `_data` corresponding to an `approve` call from the `L2CrossDomainMessenger` to an EOA owned by the attacker.

The approval allows the attacker to steal any `BVM_ETH` sitting in the `L2CrossDomainMessenger` coming from a failed `relayMessage` execution and waiting to be retried. The same attack applies to L1 where anyone can become an allowed spender of MNT stored in `L1CrossDomainMessenger` and steal those too.

Consider declaring the `BVM_ETH` address an unsafe target in the `_isUnsafeTarget` function of `L2CrossDomainMessenger` and doing the same for the MNT address in the `L1CrossDomainMessenger` contract. Alternatively consider prohibiting setting the

`BVM_ETH` / `MNT` address as target when sending cross-chain messages that will trigger one of the messenger. Moreover, consider whether such change can restrict potential use cases that are allowed by the system.

Update: *Resolved in pull request #123 at commit e251c1b. No new unit tests have been added.*

Medium Severity

M-01 Cross Domain Messengers Can Fail in Relaying a Message

The `L1CrossDomainMessenger` contract extends `CrossDomainMessenger` and overrides the `relayMessage` function. One of the characteristics of the original `relayMessage` function is that the estimation of whether there is enough gas or not to proceed with the external call has a few operations performed in between. The `hasMinGas` function, responsible for the proper gas check, has clear docstrings that warn against the overhead gas provided. It states that 40000 units of gas are added as extra gas cost to account for a worst-case scenario of the `CALL` opcode called in the subsequent external call. The worst-case scenario includes:

- Access to cold storage that accounts for 2600 units of gas.
- Call to a non-existent target that accounts for 25000 units of gas.
- A positive `msg.value` in the call that will increase the cost by 9000 units of gas.

Also, note that the second argument of the `hasMinGas` function is the sum of the following two variables:

- `RELAY_RESERVED_GAS` which is set to 40000 units of gas. This is an estimation of how much gas is needed to continue with the `relayMessage` execution after the external call. This is unchanged from Optimism code.
- `RELAY_GAS_CHECK_BUFFER` which is set to 5000 units of gas and represents an amount that should be used in between the `hasMinGas` function and the external call. This is also unchanged from Optimism code.

The `hasMinGas` function contains the following formula:

```
[ gasLeft - (40000 + _reservedGas) ] * 63/64 >= _minGas
```

Here, `_reservedGas` is 45000 units of gas of which only 5000 are estimated to be a buffer before the external call. Taking into account all of this, between the gas estimation and the external call, there is a total buffer of 5000 plus the remainder of the 40000, removing the worst case scenario of 36600 units of gas, for a total of 8400 units of gas (and not 5700 as mentioned in the docs). After the external call, another 40000 units of gas are reserve to finish with the normal execution.

The `L1CrossDomainMessenger` override adds some extra instructions in the code: an `approve` call to the MNT token contract in case the message being relayed contains a movement of MNT tokens, and a second `approve` to set the allowance back to 0 which is repeated after the external call. Whether the second approval is needed or not depends on whether there might be circumstances in which given approvals are not consumed by the target of the external call.

An approval of an ERC-20 token can span from a few thousand up to 30000 or 40000 units of gas, exceeding the buffer of few thousands units provided by far. Some instances of an `OptmismMintableERC20` token might consume even more than 40000 units of gas for every `approve` call. `approve` call gas consumption definitely depends on whether values are being set from zero to positive values or the other way around, or from non-zero to non-zero values. Notice that a similar argument can be made for the `relayMessage` function of the `L2CrossDomainMessenger`.

In light of the above, consider revisiting the values for `RELAY_GAS_CHECK_BUFFER` and `RELAY_RESERVED_GAS`, and deciding whether the second approval is needed to avoid having unexpected gas failures due to extra instructions included from the original Optimism code that came with no changes to those default estimation values. Moreover, given the added logic from Optimism code, gas buffers should be adapted to ensure that enough overhead is added so that the transactions do not fail. It is worth noting that calls to `relayMessage` that can be engineered to fail can prevent the finalization of deposits and withdrawals, opening the doors for DoS attacks.

Update: *Resolved in pull request #114 at commit 67f0904 and at commit 92ebaf9.*

M-02 Gas Estimation Can Fail in `finalizeWithdrawalTransaction`

The `finalizeWithdrawalTransaction` function of the `OptimismPortal` contract has been slightly changed from the original Optimism's code to accommodate the changes required to bridge MNT, separately from the bridging of other assets. Like many other functions

within the contract, the function is supposed to revert whenever an external call fails and the `tx.origin` is the `ESTIMATION_ADDRESS`. Conversely, if the external call did not fail, the call should not revert even if the `tx.origin` is the `ESTIMATION_ADDRESS`.

As a result of the changes introduced, the original Optimism behavior is not maintained anymore. Now, even if the external call does not fail and the `tx.origin` is the `ESTIMATION_ADDRESS`, the `finalizeWithdrawalTransaction` execution can still fail if the only asset being bridged is ETH. This is because the boolean flagging of whether the MNT transfer was successful or not defaults to `false`, making the call revert. However, this should not happen if no MNT are being transferred. The correct fix would be to set the `l1mntSuccess` boolean by default to `true` so that the same Optimism behavior is maintained, but this can also render the boolean useless as mentioned in issue N01.

Consider making the gas estimation behavior consistent with what has been inherited from Optimism and left unchanged in other parts of the codebase. When doing so, consider the mentioned issue about useless boolean variables being used.

Update: *Resolved in pull request #105 at commit 6022c06.*

M-03 Unnecessary Payable Function Definition

The `proposeL2Output` function of the `L2OutputOracle` contract is defined as `payable` but it does not handle any `msg.value`. While the function is restricted to be called exclusively by the `PROPOSER`, if any `msg.value` is passed, funds can get stuck in the contract as there is no way to pull them out.

Consider whether the `proposeL2Output` has to be `payable` and document the reason. Alternatively, consider removing the `payable` attribute.

Update: *Resolved in pull request #138 at commit af0d029.*

Low Severity

L-01 Assets Might Get Stuck in Contracts

Across the codebase, there are several circumstances in which assets can get locked in the contracts due to external call failures. Two examples are:

- When bridging ETH from L2 to L1, the `finalizeWithdrawalTransaction` function of the `OptimismPortal` is called. This will attempt to call the `L1StandardBridge` at the `finalizeBridgeETH` function which performs an external call to the recipient of the ETH. However, if such a call fails, the `success` returned boolean will be false and the `finalizeWithdrawalTransaction` will revert exclusively if the `tx.origin` is the `ESTIMATION_ADDRESS`, but it will not revert if the caller is a normal user trying to finalize the bridge back to L1. Moreover, the `finalizedWithdrawals` mapping will be set to `true` for this specific withdrawal, preventing any future attempt to replay the transaction and make it work. The result is that ETH will be stuck in the `OptimismPortal`.

- When bridging an ERC-721 token from L2 to L1, the same `finalizeWithdrawalTransaction` will be called, but this time the `L1ERC721Bridge` will be called at the `finalizeERC721Bridge` function. This function will effectively perform a `safeTransferFrom` from the `L1ERC721Bridge` to the recipient of the ERC-721 token. The usage of the `safeTransferFrom` implies also triggering a `_checkOnERC721Received` hook which will revert if the recipient is a contract that does not implement the correct interface to receive the token. If the call reverts, the same situation as before will happen since the `finalizeWithdrawalTransaciton` will not revert and the withdrawal will be marked as finalized. The result here is that the ERC-721 token will be stuck in the bridge.

Consider either documenting such behaviours in the docstrings of the contracts or putting remediations in place.

Note that case in which ETH gets stuck in the `OptimismPortal` is inherited from the Optimism contracts and Optimism have already taken a position in which they delegate the responsibility of this to the user who should understand the risk. Quoting one of their issues:

> One of the quirks of the OptimismPortal is that there is no replaying of transactions. If a transaction fails, it will simply fail, and all ETH associated with it will remain in the

> OptimismPortal contract. Users have been warned of this and understand the risks, so Optimism takes no responsibility for user error.

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

> *Won't fix; we have already implemented transaction replay at the CrossDomainMessenger contract level.*

L-02 Unusual Upgradeability Patterns Are Adopted

In the codebase, some contracts are meant to be upgradeable and so may have a `__gap` variable defined. Upgradeable contracts are meant to be called via proxies, for which reason they usually have an `initialize` function that is called through the proxy. This function sets the initial variable values of the proxy storage slots. In order to prevent someone from initializing the implementation contract directly, in some circumstances, the `initialize` function is called within the constructor. However, this consumes unnecessary gas and is sub-optimal. All initializable contracts extend the OpenZeppelin `Initializable` contract which defines an internal function called `_disableInitializers` that performs the same task of disabling implementation initializations, but without wasting gas by setting the variables to unnecessary values.

Consider calling the `_disableInitializers` function instead of calling the `initialize` function within the constructor.

Moreover, some contracts seem to have misplaced and incorrectly-used `__gap` variables as is the case for the ERC721Bridge and L1ERC721Bridge contracts. The former has two immutable variables that do not take any slot and the `__gap` variable is set to have a size of 49 slots despite the canonical value being 50, while the latter has one slot occupied by the deposits mapping. This may be misleading as it might explain the 49 slot size instead of the 50 slot size for the `__gap` variable. However, the way in which storage layout works in the `contract L1ERC721Bridge is ERC721Bridge, Semver` definition is to have the `ERC721Bridge` slots defined first, then the `Semver` ones, and finally the `L1ERC721Bridge` as the last one.

Consider reviewing the codebase and always using upgradeability standard patterns in which the `__gap` variable's size reflects the amount of storage slots occupied by the current contextual contract.

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

> *Not fixing. It will not affect the main logic.*

L-03 Wrong Value Emitted in Event

The `TokenRatioUpdated` event is emitted in the `GasPriceOracle` contract everytime the `tokenRatio` variable is updated to a new value. Its parameters are the previous and the new value being set. However, the event incorrectly emits the new token ratio twice instead of emitting the previous token ratio followed by the new token ratio.

Consider assigning the `previousTokenRatio` variable to the current token ratio instead of the used input parameter.

Update: *Resolved in pull request #138 at commit 572600a.*

L-04 Incomplete Docstrings

Throughout the codebase, there are several parts that have a incomplete docstring:

- In the relayMessage function of the `CrossDomainMessenger` contract, the `_mntValue` parameter is not documented.
- The OwnershipTransferred event of the `CrossDomainOwnable3` contract does not document what the `previousOwner`, `newOwner`, and `isLocal` parameters are.
- The TokenRatioUpdated, `OwnershipTransferred`, and `OperatorUpdated` events in the `GasPriceOracle` contract do not document what their parameters are.
- In the transferOwnership function of the `GasPriceOracle` contract, the `_owner` parameter is not documented.
- In the depositMNT function of the `L1StandardBridge` contract, the `_amount` parameter is not documented.
- In the depositMNTTo function of the `L1StandardBridge` contract, the `_amount` parameter is not documented.
- In the bridgeETH function of the `L2StandardBridge` contract, the `_value` parameter is not documented.
- In the l1Token function of the `OptimismMintableERC20` contract, not all return values are documented. The same happens in the l2Bridge, remoteToken, and the bridge functions of the same contract.
- In the WithdrawalProven event of the `OptimismPortal` contract, the `from` and `to` parameters are not documented.

- In the initialize function of the `OptimismPortal` contract, the `_paused` parameter is not documented.
- In the minimumGasLimit function of the `OptimismPortal` contract, the `_byteCount` and the return value are not documented.
- In the initialize function of the `SystemConfig` contract, the `_baseFee` parameter is not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of any contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

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

> *Will fix later. it's not a logic issue.*

L-05 Floating and Multiple Pragma Directives Are Being Used

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled. Throughout the codebase, there are multiple floating pragma directives. The majority of contracts have a fixed version of `0.8.15` but there are some contracts that differ:

- The `BVM_ETH.sol` file has the `solidity ^0.8.9` floating pragma directive.
- The `CrossDomainOwnable.sol` file has the `solidity ^0.8.0` floating pragma directive.
- The `CrossDomainOwnable2.sol` file has the `solidity ^0.8.0` floating pragma directive.
- The `CrossDomainOwnable3.sol` file has the `solidity ^0.8.0` floating pragma directive.
- The `Semver.sol` file has the `solidity ^0.8.0` floating pragma directive.

Moreover, there are cases in which one contract has a pragma directive which differs from that of its imports:

- The `CrossDomainOwnable2.sol` file has the `pragma solidity ^0.8.0;` pragma directive and imports `L2CrossDomainMessenger.sol` which has a different pragma directive.

- The `CrossDomainOwnable3.sol` file has the `pragma solidity ^0.8.0;` pragma directive and imports `L2CrossDomainMessenger.sol` which has a different pragma directive.

Consider using a fixed pragma version which is consistent across all contracts.

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

> *No need to fix.*

L-06 Unsafe ABI Encoding

It is not an uncommon practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first option is not typo-safe and the second option is not type-safe. The results in both of these methods being error-prone and thus to be considered unsafe. Within `Encoding.sol`, there are several occurrences of unsafe ABI encodings:

- In line 92.
- In line 124.

Consider replacing all the occurrences of unsafe ABI encodings with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

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

> *There is no need to fix this.*

L-07 Missing Docstrings

Throughout the codebase, there are several parts that do not have docstrings. For instance:

- The `mint` function of the `BVM_ETH` contract is not documented.
- The variables, events, and modifiers of the `GasPriceOracle` contract are not documented.
- The `L1_MNT_ADDRESS` variable of the `L1CrossDomainMessenger` contract is missing documentation. The same variable lacks docstrings in the `L1StandardBridge` and in the `L2StandardBridge` contracts.
- The `bridgeMNTTo` function of the `L2StandardBridge` contract is not documented.

- In the `depositTransaction` function of the `OptimismPortal` contract it is possible to bridge simultaneously ETH and MNT at the same time. If this is the case the user should not use the normal flow of bridging through `L2CrossDomainMessenger` and `L2StandardBridge` since this supports bridging only one asset at time. Consider warning the user about it.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not `public`, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

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

> *No need to fix.*

Notes & Additional Information

N-01 Unnecessary Boolean Values

Throughout the codebase, there are instances of boolean values being defined but not being logically useful:

- The `success` value of the `approve` call within the `_initiateBridgeMNT` function of the `L1StandardBridge` contract. The Mantle token's `approve` function either reverts or returns true, there is no case in which its result value is false.
- The `l1mntSuccess` variable of the `finalizeWithdrawalTransaction` function of the `OptimismPortal` contract is either true or the `transfer` call reverted. It will never be false.
- The `ethSuccess` variable of the `relayMessage` function of the `L2CrossDomainMessenger` contract is either true or the `approve` function reverted. It will never be false whenever its value is evaluated.

Consider refactoring the code to avoid using unnecessary boolean values. When doing so, care should be taken in maintaining the same flow of execution, especially at places where the current unnecessary booleans are being evaluated.

Update: *Resolved in pull request #128 at commit d8efd33.*

N-02 Misleading Docstrings

Several instances of incorrect or misleading docstrings have been identified throughout the codebase:

`BVM_ETH.sol`:

- Line 12: the comment above the `BVM_ETH` definition is outdated and can be misleading

`LegacyERC20MNT.sol`:

- Lines 39, 47, 55, 63: "ETH" should be "MNT"

`Burn.sol`:

- Lines 34, 35: "ETH" should be "MNT"
- The docstrings in lines 19, 21 mention that the `gas` function "burns" a specific amount of gas. However, the amount of gas is not burnt but consumed

`Types`:

- Both `mntTxValue` and `ethTxValue` share identical documentation, yet they serve different purposes in `UserDepositTransaction`.
- In `WithdrawalTransaction`'s struct documentation, the docstring for the non-existent field `value` can be removed. Additionally, both `mntValue` and `ethValue` fields are not documented.

Consider updating the misleading instances of docstrings for improved clarity and readability.

Update: *Partially resolved in pull request #129 at commit fd4fc03. The outdated comment in `BVM_ETH.sol` is still present.*

N-03 Duplicated Getter Function

The `RECIPIENT` variable of the `SequencerFeeVault` contract is declared as `public`. However, it also has a specific getter defined.

Consider removing the duplicate getter and leaving only one instance to retrieve the value of the `RECIPIENT` variable from.

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

> *No need to fix.*

N-04 `public` Functions Can Be Declared as `external`

Throughout the codebase, there are multiple instances of contracts that define `public` functions. However, these functions can be defined as `external` instead.

To save gas and improve code clarity, consider reviewing the codebase and marking all functions that are not called within the code itself as `external`.

Update: *Resolved in pull request #130 at commit c6f2d81.*

N-05 Code Style Inconsistency

The `ERC721Bridge` contract has a specific `require` statement to make sure that the caller is an EOA and not a contract. However, other contracts have a specific modifier called `onlyEOA` for the same purpose.

Consider using the `onlyEOA` modifier consistently across the codebase to improve code readability and quality.

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

> *No need to fix.*

N-06 Typographical Errors

In the codebase, there are a few instances of docstrings containing typos:

- In line 32 of the `OptimismPortal` contract, "whcih" should be "which".
- In line 72 of the `OptimismPortal` contract, the first docstring line is missing "If the value" and thus does not logically connect with the second docstring line.

Consider reviewing the entire codebase and addressing typographical errors in order to improve code quality and readability.

Update: *Resolved in pull request #131 at commit 53fe7ce.*

N-07 Variables Naming Does Not Follow Solidity Style Guide

As per the Solidity Style Guide suggestions, private or internal variable identifiers should be prefixed with `_`. Throughout the codebase, there are multiple instances of variable naming that do not follow these guidelines.

Consider reviewing the codebase and fixing any instances of irregular variable naming, and adopting all the Solidity style guidelines in order to improve the overall code quality and readability.

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

> *No need to fix.*

N-08 Use of Magic Constants

In `L1CrossDomainMessenger`, magic constants are being used. In the linked instance, the check can be changed from `< 2` to `<= MESSAGE_VERSION`.

Consider always defining constants with explicit names for better readability and understandability of the codebase.

Update: *Resolved in pull request #132 at commit 75e7984. However, the same happens on* `L2CrossDomainMessenger` *but it hasn't been fixed there.*

N-09 Usage of Single Step Ownership Transfer

In the CrossDomainOwnable and GasPriceOracle contracts, ownership is transferred in a single step. This might be pose a risk since setting an incorrect address would mean that the ownership of the contracts is permanently lost, with no method of recovery.

Consider using a two-step ownership transfer process such as OpenZeppelin's Ownable2Step.

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

> *No need to fix.*

N-10 Lack of Indexed Event Parameters

Throughout the codebase, several events do not have their parameters indexed:

- The `Withdrawal` event of the `FeeVault` contract
- The `Paused` and `Unpaused` events of the `OptimismPortal` contract

Consider indexing event parameters to improve the ability of off-chain services to search and filter for specific events.

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

> *No need to fix.*

N-11 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in these, it becomes easier for the maintainers of those libraries to make contact with the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are many instances of contracts not having a security contact.

Consider adding a NatSpec comment containing a security contact above the contract definitions. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

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

> *There is no need to fix this.*

N-12 Unnecessary Cast

Within the `LegacyERC20MNT` contract, the `address(_who)` cast is unnecessary.

To improve the overall clarity, intent, and readability of the codebase, consider removing unnecessary casts.

Update: *Resolved in pull request #133 at commit 9032ff2.*

N-13 Unused Code

The `hashDepositTransaction` function of the `Hashing` library contract is never used within the codebase. In addition, the following code eventually remains unused as well, since it currently only assists the `hashDepositTransaction` function:

- function `encodeDepositTransaction` of the `Encoding` library
- function `hashDepositSource` of the `Hashing` library
- struct `UserDepositTransaction` of the `Types` library

To improve the overall clarity, intentionality, and readability of the codebase, consider removing any unused code.

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

> *There is no need to fix this.*

N-14 Addresses of Predeploys Are Not Ordered

The constant values of addresses in the `Predeploys` library are not ordered incrementally which is prone to errors when new addresses need to be added.

Consider ordering all addresses incrementally.

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

> *There is no need to fix this.*

N-15 Address Is Being Removed Twice

In the `SystemDictator` contract, the `step3` function is being called to remove deprecated addresses from the `AddressManager` contract. However, the `BVM_CanonicalTransactionChain` address is being removed twice, first on line 293 and then on line 300.

Consider only removing the deprecated address of `BVM_CanonicalTransactionChain` once.

Update: *Resolved in pull request #135 at commit 4ed9335.*

N-16 Predeployed Contracts Missing Custom Documentation Tag

Throughout the codebase, predeployed contracts listed in the `Predeploys` library include the custom tag `@custom:predeploy` in each contract's NatSpec documentation. However, the following contracts were found missing the custom `@custom:predeploy` tag: - `ProxyAdmin` - `OptimismMintableERC721Factory` - `L2ERC721Bridge` - `BVM_ETH`

To improve code clarity, consider adding the `@custom:predeploy` tag with the appropriate address to each contract's NatSpec documentation.

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

> *There is no need to fix this.*

N-17 Unused Import

The `L1StandardBridge.sol` contract imports `L1CrossDomainMessenger` but does not use it.

Consider removing any unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #136 at commit 2002a90.*

N-18 Duplicate Event Emission

The `OwnershipTransferred` event of the `CrossDomainOwnable3` contract is already emitted inside the internal `_transferOwnership` function.

Consider removing the duplicate event.

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

> *No need to fix.*

Conclusion

The audit yielded one critical and some medium-severity issues. While the code inherited from Optimism was well documented, the new code needs some documentation fixes as suggested in the reported issues. Given the issues raised, the test suite could be improved around the changes introduced to the Mantle codebase. As such, we strongly recommend that the Mantle team implements more extensive QA and testing before going live to prevent potentially undiscovered vulnerabilities from being exploited. When doing so, it should be ensured that a high branch coverage is achieved and that comprehensive end-to-end tests are performed. The Mantle team was very responsive in resolving doubts and answering questions during the course of the audit. The official documentation was also quite helpful in getting the right context to understand the changes introduced.

Update: *The team resolved the higher severity issues and some of the issues lower in severity. Many issues have been not addressed but acknowledged. Moreover, even if the changes introduced have been properly reviewed in addressing the issues found, there is no addition of proper unit tests around those. We recommend the Mantle team to improve the overall test suite in light of the new changes introduced.*

Appendix - Locked Funds Due to Failed Deposit Transaction Exploration

The code has been adapted to support changing ETH from the native token to an ERC-20 token on L2, as well as to change MNT from an ERC-20 token to the native token. When bridging from L1 to L2, the last L1 execution step is the `depositTransaction` function of the `OptimismPortal`. This function emits an event that is then listened to at the protocol level and processed. Then, the first L2 execution step is the `relayMessage` function of the `L2CrossDomainMessenger`. The L2 execution should carry over the parameters emitted in the L1 event into the L2 execution.

The details of the execution are as follows:

1) The native L1 ETH `ethValue` amount is locked into the `OptimismPortal` contract once the `depositTransaction` execution finishes. 2) At the protocol level, the `ethValue` amount emitted in the event is minted in the form of ERC-20 WETH to the `from` parameter emitted in the `TransactionDeposited` event. 3) A snapshot is taken. 4) The amount `ethTxValue` is then transferred to the `to` of the emitted L1 event. 5) The L2 `relayMessage` function is now executed.

The reason for this flow resides in two main considerations:

- The normal user execution flow to bridge ETH from L1 to L2 would require the user to first trigger the `_initiateBridgeETH` function on the `L1StandardBridge`. This would call the `sendMessage` function of the `L1CrossDomainMessenger` contract which will then call the `depositTransaction` function of the `OptimismPortal`. When doing so, the `from` parameter emitted in the event is the aliased address of the `L1CrossDomainMessenger` while the `to` is the `L2CrossDomainMessenger`.
- One can skip this entire flow and directly call the `OptimismPortal` passing the correct parameters to execute the same exact operation. However, this can be done by directly triggering the `depositTransaction` from an externally owned account or through a user-controlled intermediary contract. If the case is the latter, the `from` parameter this time would be the aliased address of the user-controlled contract.

Now suppose that we are in this latest scenario and step `5` fails to execute. At the protocol level, the snapshot taken in step `3` is restored. In this hypothetical scenario, the `from` would have some WETH minted, but those would not have been transferred to `L2CrossDomainMessenger`. Now, thanks to this pattern, the user can trigger a transaction once again through its controlled contract using this time `ethValue == msg.value == 0` and `ethTxValue` the same value as before. This time, WETH will not be minted, but the same amount as before would be transferred from where they are stuck (the aliased address of the user-controlled address, which has no private key to unstuck the funds) to the original recipient `L2CrossDomainMessenger`, effectively un-stucking the funds.

All of this seems to solve an important issue. However, it also introduces an edge case: if the `depositTransaction` is executed from the `L1CrossDomainMessenger` and the `relayMessage` execution fails, the funds would then get stuck at the aliased address of `L1CrossDomainMessener` since there is no way for it to call the `depositTransaction` again with different values for `ethValue` and `ethTxValue`. However, such a scenario is extremely unlikely and can happen only in a few edge-case situations. The `relayMessage` function can revert if:

1) There is not enough gas to finish execution correctly at any step (even before the external call or before the minimum gas check). 2) Version of the message used is >= 2. 3) When `msg.value != _mntValue`. 4) If gasleft() - RELAY_RESERVED_GAS underflows.

Even if none of the above situations should occur, the possibility of introducing bugs with future developments might break this assumption. The aim of this write-up is just to showcase the impact of the correctness assumption of node and client being broken. As such, we strongly recommend thoroughly testing the cross-chain features end-to-end. On the other

hand, if any of the above should happen, funds would get stuck and the only way to recover funds is to either upgrade the contracts or fix the issue at the protocol level. Two possible solutions might be:

- Introduce a special restricted-access function in the `L1CrossDomainMessenger` contract to call the `depositTransaction` with custom parameters. This way, a call can be replicated with `msg.value == 0` and `ethTxValue != 0` and unstuck funds.
- Introduce a mechanism at the protocol level that calls `depositTransaction` with `msg.sender == L1CrossDomainMessenger & tx.origin != L1CrossDomainMessenger` with custom parameters as above. This would achieve the same result.

If no remediation is ultimately applied, consider documenting such a scenario, describing the potential risks involved.