Morph L2

Security Assessment

July 18, 2024

Prepared for:
Ender
Morph

Prepared by: **Anish Naik, Damilola Edwards, and Alexander Remie**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 18, 2024	Pre-project kickoff call
May 6, 2024	Status update meeting #1
May 13, 2024	Status update meeting #2
May 17, 2024	Status update meeting #3
June 7, 2024	Status update meeting #4
June 18, 2024	Status update meeting #5
June 24, 2024	Status update meeting #6
July 1, 2024	Delivery of report draft and report readout meeting
July 18, 2024	Delivery of comprehensive report with fix review addendum

Executive Summary

Engagement Overview

Morph engaged Trail of Bits to review the security of its Ethereum Layer 2 (L2) scaling solution.

A team of three consultants conducted the review from April 29 to June 28, 2024, for a total of 15 engineer-weeks of effort. Our testing efforts focused on assessing the data validation, access controls, and testing within the system to ensure that there are no opportunities to steal funds from the system or trigger a violation of a consensus-level or execution-level invariant. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

During the audit, we identified two core patterns of vulnerabilities. First, the smart contracts on the L1 and L2 side were insufficiently tested. For example, TOB-MORPH-1, which allows a delegator to steal funds from the system, should have been caught by unit testing. This issue would have been detected with stronger post-condition checks that assert upon the state changes that occur during an undelegation. Similarly, TOB-MORPH-7, which prevents challenges from being initiated if the system is paused, would have been caught if the validation of the contract's state was sufficiently checked after the system was paused and subsequently unpaused. Identification of system-level invariants and additional end-to-end testing for more complex flows (e.g., undelegations across epochs or pausing and unpausing the system) would have also helped in catching these kinds of bugs. The offchain components were generally better tested. However, it would be beneficial to expand the testing of the circuit capacity checker (CCC) outside of using a mock object. TOB-MORPH-11 showcases that the various errors that may be encountered during the application of the CCC are not sufficiently evaluated through unit and integration tests.

Second, we identified a number of instances of insufficient data validation in the smart contracts. Although a variety of these issues were generally benign (TOB-MORPH-2, TOB-MORPH-3, and TOB-MORPH-4), they highlight potential edge cases that may become problematic as the codebase changes over time. Adding stateless and isolated unit tests and developing a specification that describes each function's expected arguments and behaviors would have helped in catching these data validation issues. The offchain components had strong data validation across each critical workflow, preventing any footguns that may have occurred if validators were not in sync or were provided with inconsistent data.

More generally, the offchain components followed best practices with regards to data validation, authentication, error handling, and testing. We identified no issues in relation to violations of consensus-level invariants or execution-level invariants that would severely harm the security posture of the larger system.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Morph take the following steps prior to the next phase of development:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Improve testing for the L1 and L2 smart contracts.** The presence of issues that would lead to a theft of funds and prevent challenges from being initiated showcase shortcomings in the unit and integration testing suite of the smart contracts. Additional testing and stronger post-condition checks will significantly improve the maturity of the smart contract suite.

- **Improve testing of the CCC.** Currently, only a mock version of the CCC is used to test the execution engine. However, this prevents testing critical code paths that may occur if the CCC returns an error. It would be beneficial to invest into integration testing with a full implementation of the CCC to ensure that all errors are correctly handled and tracing works as expected.

- **Document invariants across critical code paths.** A variety of critical flows occur end-to-end within the system (e.g., block proposals, block validation, block commitments, or packing/committing a batch). These flows work across the entire stack of the system and thus have a large number of invariants that must be upheld to ensure that the system works as expected. Documenting and subsequently testing these invariants through integration and test net testing would aid in validating system behavior.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	2
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	10
Denial of Service	1
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Morph L2 codebase. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can funds be stolen from the system?
- Are rewards of stakers and delegators properly tracked across epochs?
- Is the batch challenge process susceptible to denial-of-service attacks?
- Is there a way to force the sequencer to replay or drop an L1 transaction?
- Can the rollup node be tricked into deriving deposits that were not actually made?
- Does the execution engine violate any EVM-level invariant and sufficiently handle all errors that may be returned by the EVM or the CCC?
- Can any code paths lead to the violation of a consensus-level invariant?
- Is gas currently accounted for on the L2 side?
- Do the various end-to-end flows (e.g., block proposals, voting, or block validation) have sufficient data validation and authentication to ensure consistent block production?
- Are the sequencer set and consensus parameters correctly updated across blocks?
- Is batch management across the system stack performed correctly?
- Does the transaction submitter correctly identify which sequencer is responsible for submitting batches during a given epoch?
- Does the block derivation flow correctly validate the committed batches using the blob data posted on L1?

Project Targets

The engagement involved a review and testing of the targets listed below.

morph

Repository	https://github.com/morph-l2/morph
Version	`1f623e0eaece566ef92a22615ab8ddff9bcee206`
Type	Solidity and Golang
Platform	Linux, macOS, Windows, Morph L2, EVM

go-ethereum

Repository	https://github.com/morph-l2/go-ethereum
Version	`03fd4c3e771de55015d913680e1cc0209cc92b10`
Type	Golang
Platform	Linux, macOS, Windows

tendermint

Repository	https://github.com/morph-l2/tendermint
Version	`8ff3c98baa2f1608398d9c9532722eb49bbf76b0`
Type	Golang
Platform	Linux, macOS, Windows

tx-submitter

Repository	https://github.com/morph-l2/morph/tx-submitter
Version	`34a892188644b8916907db154d9f16dc9d3912b0`
Type	Golang
Platform	Linux, macOS, Windows

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Staking system.** The staking system is responsible for handling deposits and withdrawals from stakers, delegations to stakers from delegators, and the distribution of sequencer rewards.
    - We reviewed whether it was possible to steal deposited funds on the L1 or L2 side. This led to the discovery of TOB-MORPH-1, which highlights that delegators can steal funds from the L2 staking contract.
    - We reviewed the maintenance of the active sequencer set during deposits, withdrawals, delegations, and undelegations for general correctness. This investigation did not lead to any findings.
    - We reviewed the reward distribution arithmetic and token minting logic to assess whether rewards and minting are correctly recorded and tracked per epoch. This led to the discovery of TOB-MORPH-4 and TOB-MORPH-10, which highlight that reward claiming may lead to a loss of funds or unexpected reverts.
- **Rollup.** The rollup contract is responsible for storing incoming rollup batches, challenging an invalid batch, and finalizing batches. We reviewed the rollup contract to identify opportunities for access control bypasses or issues related to the posting of invalid batches, pausing the contract, finalizing batches that are being challenged, challenging batches to cause a denial of service, and the management of the skipped L1 transaction bitmap. This led to the discovery of TOB-MORPH-7, which highlights that pausing the contract would prevent future challenges from being initiated.
- **Cross-domain messengers, message queue, and withdrawal tree.** The cross-domain messengers are responsible for relaying messages between the L1 and L2 side of the system. The message queue holds incoming L1 deposit transactions. Outgoing L2 transactions get stored in the withdrawal tree to be later finalized on the L1 side.
    - We reviewed whether it was possible to steal funds, replay messages that have already been executed, spoof messages, drop messages that were not skipped, or bypass any expected step of the deposit/withdrawal flow. This investigation did not lead to any findings.

    - We reviewed the withdrawal tree implementation for general correctness and to ensure that withdrawals cannot be spoofed or finalized before the challenge period. This investigation did not lead to any findings.

- **Governance.** The L2 governance contract allows for the creation of proposals and execution of those proposals if sufficient votes have been collected. We reviewed the contract for general correctness and to determine whether proposals are executed only if there are a sufficient number of votes. This led to the discovery of TOB-MORPH-6, which highlights that votes by non-sequencers may be counted as valid votes when they should not.

- **Tendermint.** The `tendermint` fork is responsible for proposing blocks, validating blocks, voting on blocks, and the extension of the canonical chain.

    - We reviewed the changes made to vanilla `tendermint` to assess whether any new code paths would unexpectedly cause an error or panic and halt the processing of blocks. This investigation did not lead to any findings.

    - We reviewed the block proposal lifecycle to assess whether the batch cache is correctly updated and whether communication with the L2 node is performed correctly. This investigation did not lead to any findings.

    - We reviewed the voting logic to determine whether votes on blocks with a non-empty batch hash must have a BLS signature as part of the vote. This investigation did not lead to any findings.

    - We reviewed changes made to the block replay capability and the write-ahead logger for general correctness. This investigation did not lead to any findings.

    - We reviewed the block commitment process for general correctness. This investigation did not lead to any findings.

    - We reviewed updates to the active sequencer set and consensus parameters to assess whether the data is well-formed and the updates are performed correctly. This investigation did not lead to any findings.

- **`go-ethereum`.** The `go-ethereum` fork is responsible for executing L1 and L2 transactions and the creation/validation of L2 blocks.

    - We reviewed the block creation process to determine if L1 transactions are well-ordered and if errors from the CCC and the EVM are sufficiently handled. This investigation led to the discovery of TOB-MORPH-11, which highlights that L2 transactions that fail may be added to the skipped transaction list, which may halt block production.

- We reviewed the block validation process to assess whether only valid L2 blocks are accepted. This investigation did not lead to any findings.
- We reviewed changes made to the `go-ethereum` block validation logic, blockchain data structure, and consensus engine logic to identify any changes that could lead to undefined behavior or violate any critical execution engine-level invariant. This investigation did not lead to any findings.

- **L2 node.** The L2 node is responsible for handling incoming L1 transactions, maintaining the current rollup batch, and communicating with `tendermint` and `go-ethereum` to maintain the blockchain.
    - We reviewed the L1 syncer logic to determine if only valid L1 transactions are added to the database and if any necessary variables are updated in preparation to accept the next sequence of L1 transactions. This investigation did not lead to any findings.
    - We reviewed the executor logic to assess whether communications with the `go-ethereum` execution engine are sufficiently authenticated. This investigation did not lead to any findings.
    - We reviewed the executor's batch cache management to assess whether batch points are correctly tracked, the necessary variables are updated, there are no edge cases that could lead to panics or recursive loops, and block packing to the current batch cache is performed correctly. These investigations did not lead to any findings.
    - We reviewed the block derivation logic to test whether batches committed on L1 can be validated correctly using the posted blob data. This investigation did not lead to any findings.
    - We reviewed the block proposal, validation, and commitment logic for general correctness and to assess whether there are edge cases that would halt block production. This investigation did not lead to any findings.
    - We reviewed the transaction submitter logic to assess whether the sequencer responsible for posting batches is chosen correctly and whether the transactions submitted to the L1 chain are well-formed. This investigation did not lead to any findings.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **RLP encoding of L1 messages.** L1 messages submitted to the `L1MessageQueueWithGasPriceOracle` contract are RLP-encoded to calculate the transaction hash. Given that the hash is primarily used for storing messages and does not affect critical system behavior, we deprioritized this logic.
- **Batch commitments.** Batch commitments use low-level assembly to migrate calldata into EVM memory. This logic is of high complexity and would take a large amount of time to validate manually. We deprioritized low-level memory management performed in this process.
- **Circuit capacity checker (CCC).** The underlying implementation of the CCC is considered out of scope. Thus, we assumed that the interactions with the API consume and return the necessary data structures.
- **BLS signatures.** We deprioritized the usage and verification of BLS signatures in the `tendermint` implementation.
- **`go generate` files.** A number of files in `go-ethereum` use `go generate` to provide custom serialization of certain data structures. We deprioritized review of these files.
- **Protobuf files**. We deprioritized the protocol buffer messages and services used in `tendermint`.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We did not find any issues related to the arithmetic used in the system. The accounting performed within the L1 and L2 code is not overly complex or difficult to evaluate.	**Strong**
Auditing	All critical state-changing functions emit events in the smart contracts on the L1 and L2 side. The offchain components also have sufficient logging of events and metrics.	**Strong**
Authentication / Access Controls	The authentication and access controls in the system are generally appropriate. Outside of a minor issue highlighted in TOB-MORPH-8, we did not find any vectors that could lead to an access control bypass or authorization issue.	**Satisfactory**
Complexity Management	The smart contracts are not overly complex, and their functionality is generally fairly simple and easy to understand. Functions within the offchain components are well commented to describe expected behavior.	**Satisfactory**
Configuration	The configuration of the rollup node and execution engine seems appropriate. We did not identify any configuration issues.	**Strong**
Cryptography and Key Management	The system heavily relies on the use of BLS signatures to sign and verify rollup batches. However, due to time limitations, we did not review the implementation of the	**Further Investigation Required**

	BLS signature scheme. Additionally, the BLS signatures are currently not verified on the L1 side due to limitations in the EVM precompile set.	
Data Handling	Data validation throughout the offchain components is appropriate in most cases. However, the smart contracts on the L1 and L2 side had a number of data validation concerns. For example, we identified a case where the wrong input parameter is passed to a function, allowing a theft of funds on the L2 side (TOB-MORPH-1). Additionally, we identified more benign off-by-one errors that could lead to unexpected reverts and behavior on the smart contract side (e.g. TOB-MORPH-4, TOB-MORPH-10).	**Weak**
Decentralization	The Morph network will be run by an allowlisted, decentralized network of sequencers, which mitigates risks related to censorship resistance. Additionally, the smart contract side has only a few privileged actors that have well-defined roles.	**Satisfactory**
Documentation	The code is generally well commented. The supplied documentation regarding the staking system and descriptions of the offchain components were generally sufficient. However, given the complexity of the system and large number of components, it would be beneficial to develop a specification that highlights critical workflows and describes what changes were made to vanilla `go-ethereum` or `tendermint` to achieve the expected system behavior.	**Moderate**
Low-Level Manipulation	The smart contracts on the L2 side lean heavily on the use of assembly to validate incoming rollup batches. We did not review this logic during the audit.	**Further Investigation Required**
Memory Safety and Error Handling	We did not identify any memory safety concerns. Error handling is generally appropriate throughout the codebase. Outside of the panics that are explicitly used to indicate an unexpected system state, we did not identify any vectors or code paths that could lead to a system panic.	**Satisfactory**

Testing and Verification	The testing of the offchain components is generally sufficient. However, as indicated by TOB-MORPH-11, using a mock CCC has limitations in testing all of the possible error cases that could be returned by the CCC. The smart contract side lacks sufficient unit testing and post-condition assertions, as highlighted by TOB-MORPH-1 and TOB-MORPH-7. Additional deep-rooted issues can be identified by documenting the system's critical invariants and testing them through a fuzz testing methodology.	**Weak**
Transaction Ordering	We did not find any issues related to front-running that could lead to a theft of funds or extract value from the system.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Delegators can steal funds	Data Validation	High
2	Removal of stakers may not update the sequencer set	Data Validation	Informational
3	Minting may occur in the current epoch	Data Validation	Informational
4	Reward claims may unexpectedly revert	Data Validation	Low
5	Insufficient pruning of stale data	Data Validation	Informational
6	Proposals may be executed due to votes cast by non-sequencers	Data Validation	Medium
7	Pausing the Rollup contract may prevent future challenges from being initiated	Denial of Service	High
8	Contract owner can arbitrarily delete an unfinalized batch	Data Validation	Informational
9	Lack of access controls on withdraw function	Data Validation	Informational
10	Sequencers may lose part of their commission	Data Validation	Low
11	Malicious L2 transactions can halt block production	Data Validation	High
12	Double-signing does not lead to the update of the sequencer set	Undefined Behavior	Informational

Detailed Findings

1. Delegators can steal funds	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-1
Target: `morph/contracts/contracts/L2/staking/L2Staking.sol`	

Description

Delegators can steal funds by claiming more rewards than they have accrued.

During the undelegation process, the `L2Staking` contract calls the `Distribute` contract to notify it that a delegator is undelegating from a staker (delegatee) (figure 1.1). The `Distribute` contract will then update some state variables, such as the total delegated amount for the staker, for the epoch when the undelegation completes (figure 1.2).

```
    function undelegateStake(address delegatee) external nonReentrant {
        // must claim before you can delegate stake again
        require(!_unclaimed(_msgSender(), delegatee), "undelegation unclaimed");
        require(_isStakingTo(delegatee), "staking amount is zero");

        // staker has been removed, unlock next epoch
        bool removed = stakerRankings[delegatee] == 0;

        uint256 effectiveEpoch;
        uint256 unlockEpoch;

        if (rewardStart) {
            effectiveEpoch = currentEpoch() + 1;
            unlockEpoch = removed
                ? effectiveEpoch
                : effectiveEpoch + undelegateLockEpochs;
        }

        Undelegation memory undelegation = Undelegation(
            delegatee,
            delegations[delegatee][_msgSender()],
            unlockEpoch
        );

        undelegations[_msgSender()].push(undelegation);
        delete delegations[delegatee][_msgSender()];
        stakerDelegations[delegatee] -= undelegation.amount;
```

```
417       delegators[delegatee].remove(_msgSender());
[...]
446       // notify undelegation to distribute contract
447       IDistribute(DISTRIBUTE_CONTRACT).notifyUndelegation(
448           delegatee,
449           _msgSender(),
450           effectiveEpoch,
451           undelegation.amount,
452           delegators[delegatee].length()
453       );
[...]
472    }
```

Figure 1.1: `morph/contracts/contracts/L2/staking/L2Staking.sol#L390–L472`

```
134    function notifyUndelegation(
135       address delegatee,
136       address delegator,
137       uint256 effectiveEpoch,
138       uint256 totalAmount,
139       uint256 remainsNumber
140    ) public onlyL2StakingContract {
141       // update distribution info
142       distributions[delegatee][effectiveEpoch].delegationAmount = totalAmount;
143       distributions[delegatee][effectiveEpoch].remainsNumber = remainsNumber;
[...]
167    }
```

Figure 1.2: `morph/contracts/contracts/L2/staking/Distribute.sol#L134–L167`

However, notice the deviation in what is passed to the `Distribute.notifyUndelegation` function and what the function expects. The `L2Staking.undelegateStake` function passes in the amount that was undelegated from the staker (line 451 in figure 1.1). However, the `notifyUndelegation` function expects that value to be the total amount of delegated stake (line 138 in figure 1.2). Thus, a delegator can manipulate the total delegated stake for a delegatee for a given epoch (line 142 in figure 1.2).

This is problematic when a delegator attempts to claim their rewards (figure 1.3). Since the denominator in the calculation (lines 352-355 in figure 1.3) is significantly smaller than what it should be (the value of a single undelegation versus the value of all delegated stakes), the amount that the delegator can claim is significantly larger than what they have accrued. This is a theft of funds.

```
332    function _claim(
333       address delegatee,
334       address delegator,
335       uint256 endEpochIndex
336    ) internal returns (uint256 reward) {
```

```
[...]

        for (
            uint256 i = unclaimed[delegator].unclaimedStart[delegatee];
            i <= endEpochIndex;
            i++
        ) {
            // compute reward
            reward +=
                (distributions[delegatee][i].delegatorRewardAmount *
                    distributions[delegatee][i].amounts[delegator]) /
                distributions[delegatee][i].delegationAmount;
[...]
        }

        emit RewardClaimed(delegator, delegatee, endEpochIndex, reward);
    }
```

Figure 1.3: `morph/contracts/contracts/L2/staking/Distribute.sol#L332–L398`

Exploit Scenario

Eve, a delegator, delegates 1 wei of MorphToken to a staker and then immediately undelegates it. This allows Eve to manipulate the total delegated stake for the staker `undelegateLockEpochs` epochs later (see line 405 in figure 1.1). After `undelegateLockEpochs` epochs have passed, Eve claims her rewards. Since the calculation for rewards is incorrect, Eve is able to steal funds from the system. Eve can continuously do this every few epochs to drain all user funds.

Recommendations

Short term, update the call to `Distribute.notifyUndelegation` as follows:

```
        IDistribute(DISTRIBUTE_CONTRACT).notifyUndelegation(
            delegatee,
            _msgSender(),
            effectiveEpoch,
            stakerDelegations[delegatee],
            delegators[delegatee].length()
        );
```

Figure 1.4: The call to `Distribute.notifyUndelegation` correctly passes in the total delegated stake

Long term, improve unit testing such that all state changes after a specific function call are validated. For example, unit tests for the `undelegateStake` function should ensure that the state of the `Distribute` contract is updated correctly.

2. Removal of stakers may not update the sequencer set	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-2
Target: `morph/contracts/contracts/L2/staking/L2Staking.sol`	

Description

When a combination of stakers and sequencers are removed from the L2 chain, the sequencers may not be removed from the active sequencer set.

The `L2Staking.removeStakers` function is invoked by a cross-chain message call to remove stakers on the L2 chain (figure 2.1). This cross-chain message call occurs when a staker on L1 chooses to withdraw their funds or a cabal of sequencers posted an invalid batch and are being slashed.

```
    function removeStakers(
        address[] calldata remove
    ) external onlyOtherStaking {
        bool updateSequencerSet = false;
        for (uint256 i = 0; i < remove.length; i++) {
            updateSequencerSet = rewardStart
                ? stakerRankings[remove[i]] <= latestSequencerSetSize
                : stakerRankings[remove[i]] <= sequencerSetMaxSize;

            if (stakerRankings[remove[i]] > 0) {
                // update stakerRankings
                for (
                    uint256 j = stakerRankings[remove[i]] - 1;
                    j < stakerAddresses.length - 1;
                    j++
                ) {
                    stakerAddresses[j] = stakerAddresses[j + 1];
                    stakerRankings[stakerAddresses[j]] -= 1;
                }
                stakerAddresses.pop();
                delete stakerRankings[remove[i]];

                // update candidateNumber
                if (stakerDelegations[remove[i]] > 0) {
                    candidateNumber -= 1;
                }
            }
        }
```

```
211        emit StakerRemoved(remove);
212
213        if (updateSequencerSet) {
214            _updateSequencerSet();
215        }
216    }
```

Figure 2.1: morph/contracts/contracts/L2/staking/L2Staking.sol#L183–L216

The function iterates across all stakers that need to be removed and evaluates whether the staker is a sequencer. If the staker is a sequencer, the `updateSequencerSet` value is set to `true` (lines 188-190 in figure 1.1). If `updateSequencerSet` is `true`, the `_updateSequencerSet` function is called to update the sequencer set (lines 213-215).

However, notice that the value of `updateSequencerSet` may continuously fluctuate across iterations of the loop. It may be set to `true` if the first staker in the array is a sequencer and then set to `false` if the second staker in the array is not a sequencer. Thus, when the array has a mixture of stakers and sequencers, the sequencers will not be removed from the sequencer set.

Note that this bug is currently not triggerable because:

1. **A staker is withdrawing on L1**: The length of the removal array is one, which means that the value of `updateSequencerSet` cannot change across iterations.

2. **A group of sequencers is being slashed**: The entire array of elements consists of sequencers, which means that `updateSequencerSet` will always be `true`.

Exploit Scenario

The `removeStakers` function is called with an array of stakers. The first staker is a sequencer, and the second staker is not a sequencer. After iterating through the array, the value of `updateSequencerSet` is `false`. Thus, the sequencer is not removed from the active sequencer set.

Recommendations

Short term, update the conditional logic for calculating `updateSequencerSet` such that as long as there is one sequencer in the array, `updateSequencerSet` will be `true`.

Long term, improve unit testing by adding a test that specifically validates this edge case.

3. Minting may occur in the current epoch	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-3
Target: `morph/contracts/contracts/system/MorphToken.sol`	

Description

The evaluation of the current epoch is performed incorrectly during MorphToken minting, which may allow for minting during the current epoch.

MorphToken is minted each epoch to support rewards for delegators and sequencers. Minting should not occur for a given epoch until the *next* epoch starts. The minting process is triggered by the `Record` contract, which calls the `MorphToken.mintInflations` function (figure 3.1).

```
130    function mintInflations(uint256 upToDayIndex) external onlyRecordContract {
131        uint256 currentDayIndex = (block.timestamp -
132            IL2Staking(L2_STAKING_CONTRACT).rewardStartTime()) /
133            DAY_SECONDS +
134            1;
135        require(
136            currentDayIndex > upToDayIndex,
137            "the specified time has not yet been reached"
138        );
[...]
155    }
```

Figure 3.1: `morph/contracts/contracts/system/MorphToken.sol#L130–L155`

The function performs some data validation to ensure that minting does not occur for the current epoch. However, note that the calculation for the current epoch is incorrect due to the erroneous addition of one epoch to the expected/correct value (lines 131-134 in figure 3.1). Thus, minting may occur for the current epoch.

It is important to note that this bug is currently not exploitable. This is because the `Record` contract has the necessary data validation to prevent minting for the current reward period. This data validation is performed before the call to `mintInflations`. Thus, the value of `upToDayIndex` (line 130) will not be the value of the current epoch. Additionally, no other functions can trigger minting.

Recommendations

Short term, set the value of `currentDayIndex` to the return value of the `L2Staking.currentEpoch` function.

Long term, ensure that there is only one source to calculate the current epoch (e.g. `L2Staking.currentEpoch`). Additionally, document the relationships between the various storage variables that track epochs (e.g., `Record.nextRollupEpochIndex`, `MorphToken.inflationMintedDays`, or `Distribute.mintedEpochCount`). Use fuzz testing to validate these relationships.

4. Reward claims may unexpectedly revert	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-4
Target: `morph/contracts/contracts/L2/staking/Distribute.sol`	

Description

Delegators who attempt to claim their rewards across all of the sequencers to which they have delegated may have their transactions revert.

A delegator can call the `Distribute.claimAll` function (indirectly, through the `L2Staking.claimReward` function) to claim their rewards across all of the sequencers to which they have delegated (figure 4.1).

```
226    function claimAll(
227        address delegator,
228        uint256 targetEpochIndex
229    ) external onlyL2StakingContract {
230        require(mintedEpochCount != 0, "not mint yet");
231        uint256 endEpochIndex = targetEpochIndex;
232        if (targetEpochIndex == 0 || targetEpochIndex > mintedEpochCount) {
233            endEpochIndex = mintedEpochCount - 1;
234        }
235        uint256 reward;
236        for (uint256 i = 0; i < unclaimed[delegator].delegatees.length(); i++) {
237            address delegatee = unclaimed[delegator].delegatees.at(i);
238            if (
239                unclaimed[delegator].delegatees.contains(delegatee) &&
240                unclaimed[delegator].unclaimedStart[delegatee] <= endEpochIndex
241            ) {
242                reward += _claim(delegatee, delegator, targetEpochIndex);
243            }
244        }
245        if (reward > 0) {
246            _transfer(delegator, reward);
247        }
248    }
```

Figure 4.1: `morph/contracts/contracts/L2/staking/Distribute.sol#L226–L248`

The function allows delegators to specify a “target epoch” (line 228 in figure 4.1), which is the last epoch for which the delegator wants to claim rewards. If the target epoch provided is zero, or greater than the last epoch during which MorphToken minting occurred, the function will provide rewards until the last epoch when minting occurred (`endEpochIndex` on line 233).

However, notice that the call to the `_claim` function (line 242) does not use the value of `endEpochIndex` and instead passes in the target epoch value. Thus, if a delegator attempts to claim rewards with a target epoch of zero, or a value greater than the epoch when minting last occurred in, the transaction will revert.

Exploit Scenario

Alice, a delegator, attempts to claim all of her rewards from multiple sequencers to which she has delegated. Alice passes in zero for the target epoch, which means she wants all of her rewards up until the last possible epoch. Her transaction unexpectedly reverts.

Recommendations

Short term, update the call to the `_claim` function as follows:

```
242                 reward += _claim(delegatee, delegator, endEpochIndex);
```

Figure 4.2: The call to `_claim` correctly passes in the epoch up till which rewards should be claimed

Long term, improve unit testing to identify additional edge cases like this one.

5. Insufficient pruning of stale data

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-5
Target: `morph/contracts/contracts/L2/staking/Distribute.sol`, `morph/contracts/contracts/L1/staking/L1Staking.sol`	

Description

We identified two instances of stale data that are not pruned.

The first instance is during the undelegations of delegators. Once a delegator has undelegated from a staker and has claimed all of their rewards, any relevant data pertaining to that delegator should be pruned. The pruning of delegator-related data is done in the `_claim` function (figure 5.1).

```
function _claim(
    address delegatee,
    address delegator,
    uint256 endEpochIndex
) internal returns (uint256 reward) {
[...]
    for (
        uint256 i = unclaimed[delegator].unclaimedStart[delegatee];
        i <= endEpochIndex;
        i++
    ) {
[...]
        // if distribution is empty, update next distribution info
        if (
            !distributions[delegatee][i + 1].delegators.contains(delegator)
        ) {
            distributions[delegatee][i + 1].delegators.add(delegator);
            distributions[delegatee][i + 1].amounts[
                delegator
            ] = distributions[delegatee][i].amounts[delegator];
[...]
        }
[...]
        // if undelegated, remove delegator unclaimed info after claimed all
        if (
            unclaimed[delegator].undelegated[delegatee] &&
            unclaimed[delegator].unclaimedEnd[delegatee] == i
        ) {
            unclaimed[delegator].delegatees.remove(delegatee);
            delete unclaimed[delegator].undelegated[delegatee];
```

```
390                 delete unclaimed[delegator].unclaimedStart[delegatee];
391                 delete unclaimed[delegator].unclaimedEnd[delegatee];
392                 break;
393             }
394             unclaimed[delegator].unclaimedStart[delegatee]++;
395         }
396
397         emit RewardClaimed(delegator, delegatee, endEpochIndex, reward);
398     }
```

Figure 5.1: `morph/contracts/contracts/L2/staking/Distribute.sol#L332–L398`

As shown on lines 384–393 in figure 5.1, if the epoch being evaluated is the last epoch that the delegator has delegated to a given staker, all relevant data about that delegator is pruned. However, the caveat is that lines 358-364 will actually migrate information about the delegator in the `distributions` data structure from the current epoch (which is the last epoch for the delegator) to the next epoch. Note that although this data persists for the next epoch, the delegator cannot claim any rewards for that epoch.

The second instance concerns the `stakers` data structure in the `L1StakingContract`. As shown in figure 5.2, the `register` function will update the `stakers` mapping to highlight that a new staker has registered with the system (line 168). However, neither the `withdraw` nor the `slash` function removes information about the staker from the `stakers` mapping.

```
159     function register(
160         bytes32 tmKey,
161         bytes memory blsKey
162     ) external payable inWhitelist(_msgSender()) {
163         require(stakers[_msgSender()].addr == address(0), "already registered");
164         require(tmKey != 0 && !tmKeys[tmKey], "invalid tendermint pubkey");
165         require(blsKey.length == 256 && !blsKeys[blsKey], "invalid bls pubkey");
166         require(msg.value == stakingValue, "invalid staking value");
167
168         stakers[_msgSender()] = Types.StakerInfo(_msgSender(), tmKey, blsKey);
169         stakerSet.add(_msgSender());
170         blsKeys[blsKey] = true;
171         tmKeys[tmKey] = true;
172         emit Registered(_msgSender(), tmKey, blsKey);
173
174         // send message to add staker on l2
175         _addStaker(stakers[_msgSender()]);
176     }
177
178     /// @notice withdraw staking
179     function withdraw() external {
180         require(stakerSet.contains(_msgSender()), "only staker");
181         require(withdrawals[_msgSender()] == 0, "withdrawing");
182
183         withdrawals[_msgSender()] = block.number + withdrawalLockBlocks;
184         stakerSet.remove(_msgSender());
```

```
        emit Withdrawn(_msgSender(), withdrawals[_msgSender()]);

        // send message to remove staker on l2
        address[] memory remove = new address[](1);
        remove[0] = _msgSender();
        delete whitelist[_msgSender()];
        removedList[_msgSender()] = true;
        emit StakersRemoved(remove);

        _removeStakers(remove);
    }

    /// @notice challenger win, slash sequencers
    function slash(
        address[] memory sequencers
    ) external onlyRollupContract nonReentrant returns (uint256) {
        uint256 valueSum;
        for (uint256 i = 0; i < sequencers.length; i++) {
            if (withdrawals[sequencers[i]] > 0) {
                delete withdrawals[sequencers[i]];
                valueSum += stakingValue;
            } else {
                if (stakerSet.contains(sequencers[i])) {
                    valueSum += stakingValue;
                }
                stakerSet.remove(sequencers[i]);
                // remove from whitelist
                delete whitelist[sequencers[i]];
                removedList[sequencers[i]] = true;
            }
        }

        uint256 reward = (valueSum * rewardPercentage) / 100;
        slashRemaining += valueSum - reward;
        _transfer(rollupContract, reward);

        emit Slashed(sequencers);
        emit StakersRemoved(sequencers);

        // send message to remove stakers on l2
        _removeStakers(sequencers);

        return reward;
    }
```

Figure 5.2: `morph/contracts/contracts/L1/staking/L1Staking.sol#L159–L228`

Note that in both situations, any off-chain monitoring solutions or applications that integrate with Morph may be provided incorrect information given the existence of stale data.

Exploit Scenario

Alice, a developer of an application that integrates with Morph, queries the `stakers` mapping to identify whether Bob is still staked on Morph. Unbeknownst to Alice, Bob has withdrawn his stake. However, due to the lack of pruning, Alice is provided the incorrect information.

Recommendations

Short term:

- For undelegations, remove all information about the delegator in the `distributions` mapping for the next epoch during pruning.
- For the `stakers` mapping, remove the staker from the `stakers` mapping upon withdrawals and slashing.

Long term, improve unit testing such that all state changes after a specific function call are validated.

6. Proposals may be executed due to votes cast by non-sequencers	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-6
Target: `morph/contracts/contracts/L2/staking/Gov.sol`	

Description

Proposals may be executed due to votes that were cast by sequencers that are no longer in the active sequencer set.

The `Gov.vote` function can be called by any sequencer to submit a vote for the current proposal. If the number of votes for the proposal is greater than ⅔ the size of the active sequencer set, the proposal is executed (lines 183-185 in figure 6.1).

```
162    function vote(
163        uint256 proposalID
164    ) external onlySequencer proposalCheck(proposalID) {
165        require(
166            !votes[proposalID].contains(_msgSender()),
167            "sequencer already vote for this proposal"
168        );
169
170        // update votes
171        votes[proposalID].add(_msgSender());
172
173        // checking invalidate votes
174        address[] memory latestSequencerSet = ISequencer(SEQUENCER_CONTRACT)
175            .getSequencerSet2();
176        for (uint i = 0; i < latestSequencerSet.length; i++) {
177            if (!votes[proposalID].contains(latestSequencerSet[i])) {
178                votes[proposalID].remove(latestSequencerSet[i]);
179            }
180        }
181
182        // check votes
183        if (votes[proposalID].length() > (latestSequencerSet.length * 2) / 3) {
184            _executeProposal(proposalID);
185        }
186    }
```

Figure 6.1: `morph/contracts/contracts/L2/staking/Gov.sol#L162–L186`

However, there is no guarantee that at any given time that all the votes cast for a proposal were provided by *active* sequencers. Since the active sequencer set can change per block, a vote cast by a sequencer in block X may no longer be valid by block Y. Thus, proposals may be executed incorrectly since some votes for the proposal may be cast by sequencers that are no longer in the active sequencer set.

This would allow for critical consensus layer parameters to be changed when they should not have been.

Exploit Scenario

Alice, a sequencer, votes for a proposal in block X. By block Y, Alice is no longer in the active sequencer set. In block Z, Bob, another sequencer, submits a vote (note that `X < Y < Z`). Bob's vote causes the proposal to be executed. However, the proposal should not have been executed since Alice's vote for the proposal is no longer valid.

Recommendations

Short term, call `_checkProposal` before the call to `_executeProposal`. The `_checkProposal` function ensures that only votes cast by active sequencers are counted as valid. Additionally, remove lines 176-180 (see figure 6.1) in the `vote` function. This logic incorrectly attempts to remove invalid votes.

Long term, document all the system invariants. For example, an invariant of the `Gov` contract is that a proposal should be executed if more than ⅔ of the *active* sequencer set has voted. These invariants can be tested via additional unit and fuzz testing.

7. Pausing the Rollup contract may prevent future challenges from being initiated

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-MORPH-7
Target: `morph/contracts/contracts/L1/rollup/Rollup.sol`	

Description

Due to the incomplete removal of the current challenge information when the rollup contract is paused, new challenges cannot be initiated after the system is unpaused.

When the `Rollup` contract is paused while there is an active challenge, the challenge deposit is refunded and the information about the challenge is deleted (lines 607-612 in figure 7.1).

```
603    function setPause(bool _status) external onlyOwner {
604        if (_status) {
605            _pause();
606            // if challenge exist and not finished yet, return challenge deposit to challenger
607            if (inChallenge && !challenges[batchChallenged].finished) {
608                batchChallengeReward[
609                    challenges[batchChallenged].challenger
610                ] += challenges[batchChallenged].challengeDeposit;
611            }
612            delete challenges[batchChallenged];
613        } else {
614            _unpause();
615        }
616    }
```

Figure 7.1: `morph/contracts/contracts/L1/rollup/Rollup.sol#L603–L616`

However, the `isChallenge` flag is not reset to `false`. This prevents the creation of new challenges (line 485 in figure 7.2), which effectively blocks the creation of any future challenge.

```
482    function challengeState(
483        uint64 batchIndex
484    ) external payable onlyChallenger nonReqRevert whenNotPaused {
485        require(!inChallenge, "already in challenge");
```

Figure 7.2: `morph/contracts/contracts/L1/rollup/Rollup.sol#L482–L485`

Currently, the `inChallenge` storage variable of the contract can be reset only when a challenge is resolved in the `proveState` function, but in this case, the check to verify the existence of the challenge (line 630 in figure 7.3) would revert.

```
624    function proveState(
625       uint64 _batchIndex,
626       bytes calldata _aggrProof,
627       bytes calldata _kzgDataProof
628    ) external nonReqRevert whenNotPaused {
629       // Ensure challenge exists and is not finished
630       require(batchInChallenge(_batchIndex), "batch in challenge");
631
632       // Mark challenge as finished
633       challenges[_batchIndex].finished = true;
634       inChallenge = false;
```

Figure 7.3: `morph/contracts/contracts/L1/rollup/Rollup.sol#L624–L634`

Exploit Scenario

Alice initiates a challenge on a batch in the `Rollup` contract. The contract is then paused and the challenge information is cleared. After the contract is unpaused, Alice cannot win the previous challenge or restart the challenge. This prevents any future challenges from occurring. This incentivizes malicious sequencers to start posting invalid batches in order to steal funds from the system.

Recommendations

Short term, reset the `inChallenge` state variable to `false` when clearing challenge information in `Rollup.setPause`.

Long term, improve unit testing around the pausing capability. Ensure that the system works as expected after the system is unpaused.

8. Contract owner can arbitrarily delete an unfinalized batch

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-8
Target: `morph/contracts/contracts/L1/rollup/Rollup.sol`	

Description

Unfinalized transaction batches can be reverted or deleted arbitrarily by the `Rollup` contract owner.

The `revertBatch` function allows the owner to revert a sequence of batches as long as these batches are not finalized.

```
function revertBatch(
    bytes calldata _batchHeader,
    uint256 _count
) external onlyOwner {
    require(_count > 0, "count must be nonzero");

    (uint256 memPtr, bytes32 _batchHash) = _loadBatchHeader(_batchHeader);
    // check batch hash
    uint256 _batchIndex = BatchHeaderCodecV0.getBatchIndex(memPtr);

    require(
        batchBaseStore[_batchIndex].batchHash == _batchHash,
        "incorrect batch hash"
    );
    // make sure no gap is left when reverting from the ending to the beginning.
    require(
        batchBaseStore[_batchIndex + _count].batchHash == bytes32(0),
        "reverting must start from the ending"
    );
    // check finalization
    require(
        _batchIndex > lastFinalizedBatchIndex,
        "can only revert unFinalized batch"
    );

    lastCommittedBatchIndex = _batchIndex - 1;
    while (_count > 0) {
        batchBaseStore[_batchIndex].batchHash = bytes32(0);
```

Figure 8.1: `morph/contracts/contracts/L1/rollup/Rollup.sol#L430–L457`

However, this raises some potential transaction censorship concerns, as valid unfinalized batches can be arbitrarily reverted by the contract owner even if no challenge has been initiated on them.

Exploit Scenario

Alice, the owner of the `Rollup` contract, reverts a series of batches by mistake. Note that these batches were valid and had no challenges. This prevents the finalization of the L2 chain, which prevents L1 withdrawals from completing.

Recommendations

Short term, improve user- and developer-facing documentation to highlight this capability of the owner of the `Rollup` contract. Additionally, consider constraining the owner to delete a batch only if the batch was successfully challenged. This can be done by checking that `revertReqIndex` is greater than zero.

Long term, improve user- and developer-facing documentation to highlight all capabilities of all privileged users. This documentation should be maintained as the codebase continues to evolve.

9. Lack of access controls on withdraw function

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-9
Target: `morph/contracts/contracts/L2/system/L2TxFeeVault.sol`	

Description

The L2 fee vault's withdraw function is callable by anyone (figure 9.1). This allows an attacker to create small withdrawals from L2 to L1 and cause the relayer to expend unnecessary L1 gas. Note that since L2 transactions are batched, the additional L1 gas cost is minimal.

```
function withdraw(uint256 _value) public {
    require(
        _value >= minWithdrawAmount,
        "FeeVault: withdrawal amount must be greater than minimum withdrawal amount"
    );

    uint256 _balance = address(this).balance;
    require(
        _value <= _balance,
        "FeeVault: insufficient balance to withdraw"
    );

    unchecked {
        totalProcessed += _value;
    }

    emit Withdrawal(_value, recipient, msg.sender);

    // no fee provided
    IL2CrossDomainMessenger(messenger).sendMessage{value: _value}(
        recipient,
        _value,
        bytes(""), // no message (simple eth transfer)
        0 // _gasLimit can be zero for fee vault.
    );
}
```

Figure 9.1: `morph/contracts/contracts/L2/system/L2TxFeeVault.sol#L116–L141`

Recommendations

Short term, add the `onlyOwner` modifier to the `withdraw` function.

10. Sequencers may lose part of their commission	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MORPH-10
Target: `morph/contracts/contracts/L2/staking/Distribute.sol`	

Description

Sequencers who attempt to claim their commission may unexpectedly lose part of it.

The `Distribute.claimCommission` function can be called by a sequencer who wishes to claim some of their commission reward (figure 10.1). The sequencer can provide a “target epoch” that specifies the epoch up until which the sequencer wants to claim commission. The sequencer can also choose to claim all their commission by specifying a target epoch that is zero or greater than the last epoch when MorphToken minting occurred (lines 159-261 in figure 10.1).

```
253    function claimCommission(
254        address delegatee,
255        uint256 targetEpochIndex
256    ) external onlyL2StakingContract {
257        require(mintedEpochCount != 0, "not mint yet");
258        uint256 end = targetEpochIndex;
259        if (targetEpochIndex == 0 || targetEpochIndex > mintedEpochCount) {
260            end = mintedEpochCount - 1;
261        }
262        require(
263            unclaimedCommission[delegatee] <= end,
264            "all commission claimed"
265        );
266        uint256 commission;
267        for (uint256 i = unclaimedCommission[delegatee]; i <= end; i++) {
268            commission += distributions[delegatee][i].commissionAmount;
269        }
270        if (commission > 0) {
271            _transfer(delegatee, commission);
272        }
273        unclaimedCommission[delegatee] = end + 1;
274
275        emit CommissionClaimed(delegatee, end, commission);
276    }
```

Figure 10.1: *`morph/contracts/contracts/L2/staking/Distribute.sol#L253-L276`*

However, note that the conditional on line 259 incorrectly evaluates the last epoch that minting occurred in, which is the `mintedEpoch - 1` epoch. Thus, a sequencer can pass in a value where `targetEpochIndex` equals `mintedEpochIndex`, which updates the `unclaimedCommission` mapping to disallow the sequencer to receive commission for the `mintedEpochIndex` epoch (line 273).

Note that this issue is also present in the following two functions:

- `Distribute.claim`
- `Distribute.claimAll`

Exploit Scenario

Alice, a sequencer, wishes to claim her entire commission. Thus, Alice passes in a value for `targetEpochIndex` that is currently equal to the `mintedEpochIndex`. As a result, Alice can no longer claim any commission for the `mintedEpochIndex` epoch. After a few more epochs, Alice attempts to claim her commission again. However, she loses her commission for one epoch.

Recommendations

Short term, update line 259 as follows:

```
259        if (targetEpochIndex == 0 || targetEpochIndex >= mintedEpochCount) {
```

Figure 10.2: The conditional now correctly evaluates the last epoch when minting occurred

Long term, improve unit testing to identify additional edge cases like this one.

11. Malicious L2 transactions can halt block production

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MORPH-11
Target: `go-ethereum/miner/worker.go`	

Description

An attacker can craft a malicious L2 transaction that causes the resultant L2 block to be considered invalid by validators.

When a block is first being proposed, a sequencer node will create an L2 block that includes both L1 and L2 transactions. After the application of each transaction on the EVM, the node logic will check to see if and why a transaction has failed. L1 transactions that fail due to a block gas limit error or a CCC-related error get added to the skipped transaction list. Note that L2 transactions that fail should not get added to the skipped transaction list (figure 11.1).

```
case errors.Is(err, circuitcapacitychecker.ErrBlockRowConsumptionOverflow):
    if env.tcount >= 1 {
[...]
    } else {
        // 2. Circuit capacity limit reached in a block, and it's the first tx: skip the tx
        log.Trace("Circuit capacity limit reached for a single tx", "tx", tx.Hash().String())

        if tx.IsL1MessageTx() {
[...]
        } else {
            // Skip L2 transaction and all other transactions from the same sender account
            log.Info("Skipping L2 message", "tx", tx.Hash().String(), "block", env.header.Number, "reason", "first tx row consumption overflow")
            txs.Pop()
            w.eth.TxPool().RemoveTx(tx.Hash(), true)
            l2TxRowConsumptionOverflowCounter.Inc(1)
        }

        // Reset ccc so that we can process other transactions for this block
        w.circuitCapacityChecker.Reset()
        log.Trace("Worker reset ccc", "id", w.circuitCapacityChecker.ID)
        circuitCapacityReached = false
```

```
1186
1187                var storeTraces *types.BlockTrace
1188                if w.config.StoreSkippedTxTraces {
1189                        storeTraces = traces
1190                }
1191                skippedTxs = append(skippedTxs, &types.SkippedTransaction{
1192                        Tx:     *tx,
1193                        Reason: "row consumption overflow",
1194                        Trace:  storeTraces,
1195                })
1196        }
```

Figure 11.1: `go-ethereum/miner/worker.go#L1138–L1196`

However, as shown in figure 11.1 (lines 1191-1195), an L2 transaction that is the *first* transaction in the block and causes a CCC block row consumption overflow error gets added to the skipped transaction list. Since the L2 transaction is the first transaction in the block, there are no L1 transactions in the block. Thus, the length of the skipped transaction list should be zero but is instead one.

As a result, when a validator attempts to validate the contents of the proposed L2 block, it will throw an error when it identifies that the length of the skipped transaction list is different than expected (figure 11.2). Note that an attacker can purposefully create L2 transactions that cause a CCC row block consumption overflow error by invoking a smart contract that excessively calls EVM precompiles, which would prevent the call from fitting into the zk-EVM circuit.

```
func (e *Executor) validateL1Messages(block *catalyst.ExecutableL2Data,
collectedL1TxHashes []common.Hash) error {
[...]
        // constraints 7
        if len(skipped) != len(block.SkippedTxs) {
                e.logger.Error("found wrong number of skipped txs", "expected
skippedTx num", len(skipped), "actual", len(block.SkippedTxs))
                return types.ErrInvalidSkippedL1Message
        }
[...]
        return nil
}
```

Figure 11.2: `morph/core/l1_message.go#L27–L138`

Exploit Scenario

Eve, a malicious L2 user, deploys a smart contract that excessively uses the `ecrecover` precompile. Invoking this smart contract through a transaction prevents the transaction from fitting into the zk-EVM circuit. Eve continuously calls this malicious smart contract, and her L2 transaction ends up being the first transaction in the L2 block (there are no pending L1 messages that need to be processed by the node). Her transaction causes a CCC block row consumption overflow error and gets added to the skipped transaction list. This results in a malformed block. Validators prevote nil for the proposed block since the block does not pass the block validation process. This halts the progression of the blockchain.

Recommendations

Short term, only add the transaction to the skipped transaction list if the failing transaction is an incoming L1 transaction.

Long term, improve testing of the CCC by testing code paths that occur if the CCC returns an error.

12. Double-signing does not lead to the update of the sequencer set	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MORPH-12
Target: `tendermint`	

Description

A validator who signs conflicting messages for a given round, height, and step is not ultimately removed from the sequencer set.

Currently, validators can get slashed within the system only if one or more validators post an invalid rollup batch. Slashing results in the sequencer set getting updated during the next L2 block.

However, a validator can also act maliciously at the consensus level through double-signing messages. Although evidence of malicious votes is published, the system currently does not use this evidence to remove malicious validators from the sequencer set.

Exploit Scenario

Alice, a malicious validator, double-signs two separate messages for a given block proposal. This action results in the publishing of evidence that a malicious validator is within the network. However, the underlying application, the L2 node, does not use this evidence to punish the validator and remove them from the sequencer set.

Recommendations

Long term, identify a mechanism to post evidence on the L2 chain. This evidence can then be used to identify and remove malicious validators. Additionally, document all limitations of the current system design.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Fix the following typos**. The inline comment in figure C.1 should say **"if delegator info does not exist in the next epoch, distribution must be updated"**. In figure C.2, the inline comment should say **"must claim any undelegation first"**. In figure C.3, the inline comment should start with **"`newBlockTimeout`"**.

```
// if delegator info exist in next epoch, distribution must be updated
```

Figure C.1: `morph/contracts/contracts/L2/staking/Distribute.sol#L366`

```
// must claim before you can delegate stake again
```

Figure C.2: `morph/contracts/contracts/L2/staking/L2Staking.sol#L391`

```
// newpayloadTimeout is the maximum timeout allowance for creating block.
// The default value is 3 seconds but node operator can set it to arbitrary
// large value. A large timeout allowance may cause Geth to fail creating
// a non-empty block within the specified time and eventually miss the chance to be a proposer
// in case there are some computation expensive transactions in txpool.
newBlockTimeout time.Duration
```

Figure C.3: `go-ethereum/miner/worker.go#L232–L237`

- **Add zero address validation checks**. Certain functions fail to validate incoming address arguments, so callers of these functions could mistakenly transfer funds or set important state variables to a zero address. The following areas lack the appropriate zero address checks:
  - `morph/contracts/contracts/L1/staking/L1Staking.sol#L289–L297`
  - `morph/contracts/contracts/L1/staking/L1Staking.sol#L332–L337`
  - `morph/contracts/contracts/L1/rollup/Rollup.sol#L743–L748`
  - `morph/contracts/contracts/L1/rollup/Rollup.sol#L932–L937`

- **Improve inline code comments.** Clearly document what each state variable represents and its purpose.

```
/// @param batchHash
/// @param batchVersion
/// @param originTimestamp
```

```
    /// @param finalizeTimestamp
    /// @param blockNumber
    struct BatchBase {
        bytes32 batchHash;
        uint256 batchVersion;
        uint256 originTimestamp;
        uint256 finalizeTimestamp;
        uint256 blockNumber;
    }

    /// @param blobVersionedHash
    /// @param l1DataHash
    /// @param prevStateRoot
    /// @param postStateRoot
    /// @param withdrawalRoot
    /// @param l1MessagePopped
    /// @param totalL1MessagePopped
    /// @param skippedL1MessageBitmap
    struct BatchData {
        bytes32 blobVersionedHash;
        bytes32 l1DataHash;
        bytes32 prevStateRoot;
        bytes32 postStateRoot;
        bytes32 withdrawalRoot;
        uint256 l1MessagePopped;
        uint256 totalL1MessagePopped;
        bytes skippedL1MessageBitmap;
    }

    /// @param blsMsgHash
    /// @param sequencerSetVerifyHash
    /// @param sequencers
    struct BatchSignature {
        bytes32 blsMsgHash;
        bytes32 sequencerSetVerifyHash;
        bytes signedSequencers;
    }
```

Figure C.4: `morph/contracts/contracts/L1/rollup/IRollup.sol#L35–L74`

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From July 1 to July 5, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Morph team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 12 issues described in this report, Morph has resolved 10 issues and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Delegators can steal funds	**Resolved**
2	Removal of stakers may not update the sequencer set	**Resolved**
3	Minting may occur in the current epoch	**Resolved**
4	Reward claims may unexpectedly revert	**Resolved**
5	Insufficient pruning of stale data	**Resolved**
6	Proposals may be executed due to votes cast by non-sequencers	**Resolved**
7	Pausing the Rollup contract may prevent future challenges from being initiated	**Resolved**
8	Contract owner can arbitrarily delete an unfinalized batch	**Unresolved**
9	Lack of access controls on withdraw function	**Resolved**

10	Sequencers may lose part of their commission	Resolved
11	Malicious L2 transactions can halt block production	Resolved
12	Double-signing does not lead to the update of the sequencer set	Unresolved

Detailed Fix Review Results

TOB-MORPH-1: Delegators can steal funds
Resolved in PR 282. The Morph team has updated the undelegate stake function to prevent a malicious delegator from manipulating sensitive state variables to steal funds from the system.

TOB-MORPH-2: Removal of stakers may not update the sequencer set
Resolved in PR 275. The Morph team has updated the sequencer removal logic to guarantee that all sequencers that should be removed from the sequencer set are removed.

TOB-MORPH-3: Minting may occur in the current epoch
Resolved in PR 277. The Morph team has updated the minting logic to prevent minting in the current epoch.

TOB-MORPH-4: Reward claims may unexpectedly revert
Resolved in PR 281. The Morph team has updated the reward claiming logic to prevent unexpected reverts.

TOB-MORPH-5: Insufficient pruning of stale data
Resolved in PR 308. The Morph team has updated the L1 staking contract to prune the `stakers` data structure whenever a staker is slashed or when the staker completes the withdrawal process. Additionally, the team has updated the reward claiming logic to not migrate stale information about a delegator if they have undelegated from a given staker/delegatee.

TOB-MORPH-6: Proposals may be executed due to votes cast by non-sequencers
Resolved in PR 304. The Morph team has updated the governance contract to only count votes from the current sequencer set before the execution of a proposal.

TOB-MORPH-7: Pausing the Rollup contract may prevent future challenges from being initiated
Resolved in PR 300. The Morph team has updated the pausing logic to ensure that challenges can still be initiated after the system is unpaused.

TOB-MORPH-8: Contract owner can arbitrarily delete an unfinalized batch
Unresolved. The Morph team notes that the owner's ability to delete an unfinalized batch is the expected behavior of the system. Additionally, while reviewing the issue, the Morph team independently identified a bug related to challenge resolution in the batch reverting process, which was fixed in PR 301.

TOB-MORPH-9: Lack of access controls on withdraw function
Resolved in PR 303. The Morph team has updated the withdraw function to allow only the owner to call it.

TOB-MORPH-10: Sequencers may lose part of their commission
Resolved in PR 277. The Morph team has updated the claim, claim all, and claim commission functions to correctly evaluate the last epoch up to which rewards can be claimed.

TOB-MORPH-11: Malicious L2 transactions can halt block production
Resolved in PR 99. The Morph team has updated the block production logic to prevent L2 transactions from being added to the skipped L1 transaction list.

TOB-MORPH-12: Double-signing does not lead to the update of the sequencer set
Unresolved. The Morph team has acknowledged the risk related to validators double-signing messages and will remediate this issue before the launch of a completely decentralized sequencer network (currently, the sequencers must be allowlisted).

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.