Security Assessment & Formal Verification Report

March 2024

Prepared for
M^Zero

Table of contents

certora

M^0

Table of Contents

Project Summary

Project Scope

Repo Name	Repository	Commits	Compiler version	Platform
M^0 protocol	https://github.com/MZero-Labs/protocol	2e9f5cd32b292d3dea71ba1dfff0b2b68cab9467	solidity 0.8.23	EVM
TTG	https://github.com/MZero-Labs/ttg	7581622f35d64820bac667580cb9690982298eb2	solidity 0.8.23	EVM
Common	https://github.com/MZero-Labs/common	2ba852c44e02e3bee58cafea3ce7964022bd4486	solidity 0.8.23	EVM

Project Overview

This document describes the specification and verification of the **MZero protocol and Two Token Governance (TTG)** using the Certora Prover and manual code review findings. The work was undertaken from **18 January 2024** to **7 March 2024**.

The following contract list is included in our scope:

TTG:

```
src/abstract/interfaces/IBatchGovernor.sol
src/abstract/interfaces/IEpochBasedInflationaryVoteToken.sol
src/abstract/interfaces/IEpochBasedVoteToken.sol
src/abstract/interfaces/IERC5805.sol
src/abstract/interfaces/IERC6372.sol
src/abstract/interfaces/IGovernor.sol
src/abstract/interfaces/IThresholdGovernor.sol
```

src/abstract/BatchGovernor.sol
src/abstract/EpochBasedInflationaryVoteToken.sol
src/abstract/EpochBasedVoteToken.sol
src/abstract/ERC5805.sol
src/abstract/ThresholdGovernor.sol
src/interfaces/IDeployer.sol
src/interfaces/IDistributionVault.sol
src/interfaces/IEmergencyGovernor.sol
src/interfaces/IEmergencyGovernorDeployer.sol
src/interfaces/IPowerBootstrapToken.sol
src/interfaces/IPowerToken.sol
src/interfaces/IPowerTokenDeployer.sol
src/interfaces/IRegistrar.sol
src/interfaces/IStandardGovernor.sol
src/interfaces/IStandardGovernorDeployer.sol
src/interfaces/IZeroGovernor.sol
src/interfaces/IZeroToken.sol
src/libs/PureEpochs.sol
src/DistributionVault.sol
src/EmergencyGovernor.sol
src/EmergencyGovernorDeployer.sol
src/PowerBootstrapToken.sol
src/PowerToken.sol
src/PowerTokenDeployer.sol
src/Registrar.sol
src/StandardGovernor.sol
src/StandardGovernorDeployer.sol
src/ZeroGovernor.sol
src/ZeroToken.sol

Protocol:
src/abstract/ContinuousIndexing.sol
src/interfaces/IContinuousIndexing.sol
src/interfaces/IMinterGateway.sol
src/interfaces/IMToken.sol
src/interfaces/IRateModel.sol
src/interfaces/ITTGRegistrar.sol
src/libs/ContinuousIndexingMath.sol
src/libs/TTGRegistrarReader.sol
src/rateModels/interfaces/IEarnerRateModel.sol

```
src/rateModels/interfaces/IMinterRateModel.sol
src/rateModels/interfaces/IStableEarnerRateModel.sol
src/rateModels/MinterRateModel.sol
src/rateModels/SplitEarnerRateModel.sol
src/rateModels/StableEarnerRateModel.sol
src/MinterGateway.sol
src/MToken.sol
```

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Protocol Overview

The core M^0 protocol is a coordination layer for permissioned institutional actors to generate M. M is a fungible token that can be generated by locking Eligible Collateral in a secure off-chain facility. The protocol enforces a common set of rules and safety procedures for the management of M.

The M^0 protocol is a set of immutable smart contracts implemented for the Ethereum Virtual Machine. It receives all external inputs from a governance mechanism called the M^0 Two Token Governor, TTG.

Aside from users accessing the M token, which is permissionless, only actors permissioned through the TTG mechanism have access to the M^0 Protocol. The primary actors in the protocol are called Minters and Validators.

The M^0 protocol uses an on-chain governance mechanism called a Two Token Governor (TTG) to manage its various inputs. With TTG, holders of the voting tokens are penalized for failing to vote.

There are two utility tokens used in the M^0 TTG: POWER and ZERO. POWER is used to vote on active proposals and can be considered the primary management token of the mechanism. POWER holders will earn ZERO in exchange for their direct participation in governance.

ZERO holders at any time may Reset the POWER token supply to themselves.

The goal of the mechanism is to ensure credible neutrality of governance.

POWER holders are treated as a managerial class that is able to earn compensation through continued benevolent participation. This continued benevolence is judged by the ZERO holders who can always strip the POWER holders of their management rights, and thus their ability to earn future ownership in the protocol.

Findings Summary

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

Severity	Discovered	Acknowledged	Code Fixed
Critical	2		
High	1		
Medium	2		
Low	3		
Informational	5		
Total	**13**		

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Probability	

certora

M^0

Detailed Findings

ID	Title	Severity
C-1	Attacker can double its PowerToken balance and voting power through bootstrap()	Critical
C-2	Attacker can inflate his PowerToken balance by repeatedly claiming historic inflation with sync() function	Critical
H-1	Past Voting Power isn't read from bootstrap after sync.	High
M-1	A changed rate would not take effect until the updateIndex() function is called	Medium
M-2	One rogue validator might allow a minter to avoid paying penalties	Medium
L-1	Overflow check reverts due to overflow without emitting a revert message.	Low
L-2	Proposal fee is not returned to proposer after RESET event	Low
L-3	_lastSyncs array is redundant, resulting in waste of gas	Low
INFO-1	Lack of EIP-712 compliance: using keccak256() directly on an array or struct variable	Informational
INFO-2	Array lengths not checked	Informational
INFO-3	Certain functions should not be marked as payable	Informational
INFO-4	Unnecessarily small uint type for an incremented variable	Informational

INFO-5	Possible gas griefing when emitting voting event	Informational

Critical Severity Issues

C-1. Attacker can double its PowerToken balance and voting power through bootstrap()

Impact: Critical
Probability: High

Description:

The _bootstrap() function is supposed to be called once and be followed by _sync() function. However, in the case of a RESET event or after deployment, a malicious user can enter _bootstrap() twice before entering _sync(). It can happen when calling transfer() or delegate() to himself. This will affect accounting. Attacker will double his power token balance and vote power balance (without properly updating totalSupply). Thus it gives unfair advantage in Power Threshold proposals.

Customer's response: Ack + fixed

C-2. Attacker can inflate his PowerToken balance by repeatedly claiming historic inflation with sync() function

Impact: Critical
Probability: High

Rules violated:
synchingDoesntChangeBalance,
pastUnrealizedInflationIsImmutable,
pastBalanceIsImmutable

Description:

The issue originates in sync(address account, uint16 epoch) function where it is possible to specify **any** epoch for syncing. Later down the trace, in _getUnrealizedInflation(), it will try to compute inflation for epoch in range (_getLastSync(_lastEpoch), _lastEpoch) The problem is that _getLastSync() can be gamed. If we start with array _lastSyncs[alice] = [20, 30] then we can successfully sync(alice, 29) multiple times and we will print her inflation from epochs in range (20...29). The balance will be updated with respect to the current epoch.

The root cause of the exploit is an error in registering a new event: Instead of the epoch of the sync(address account, uint16 epoch), the _addBalance function always registers a new event with the current epoch instead, or updates the current epoch balance if exists.

Customer's response:

Was fixed in commit 6b487a018a5b4496484b76a0815ee8a9577214d2

High Severity Issues

H-1. Past Voting Power isn't read from bootstrap after sync.

Impact: High
Probability: High

Rules violated:
pastBalanceIsImmutable

Description: PowerToken._getBalance(account_, epoch_) function checks the latest saved balance with respect to specified epoch_ or consults bootstrap token (in case the balance array is empty). However, if balance becomes non-empty after specified epoch_, the function will skip consulting the bootstrap token and likely return 0. This is a serious issue in case of Emergency POWER Threshold proposals. If a user performs his first activity (transfer, delegate, or anything that will save his balance) during the epoch which Emergency proposal was proposed, then his voting power will likely be 0 (PowerToken will not read balance from BootstrapToken).

Customer's response:

Was fixed in commit 6b487a018a5b4496484b76a0815ee8a9577214d2

Medium Severity Issues

M-1. A changed rate would not take effect until the updateIndex() function is called

Impact: Medium
Probability: Medium

Description: Updating the rate does not necessarily trigger a call to the updateIndex() function, and thus the growth of the index might still be calculated using the old rate until the updateIndex() function is called.

Customer's response:

M-2. One rogue validator might allow a minter to avoid paying penalties

Impact: Medium
Probability: Low

Description: In updateCollateral the minter has to reach a quota of signatures to validate an off chain deposit. Each signature contains a timestamp and the _verifyValidatorSignatures function returns the minimum of all these timestamps. If the minter has 1 rogue signer on his side, on every update, the minter can always add a signature with timestamp==0, which will keep _minterStates[minter_].updateTimestamp at 0 for eternity. That might allow him to avoid any potential penalties the minter should have faced for infrequent updates.

Customer's response:

Low Severity Issues

L-1. Overflow check reverts due to overflow without emitting a revert message.

Impact: Low
Probability: Low

Description: The line: `revert OverflowsPrincipalOfTotalSupply();` in line 229 is only called if the condition:

`principalOfTotalEarningSupply + _getPrincipalAmountRoundedDown(totalNonEarningSupply) >= type(uint112).max` is true. However, this could only happen if the left hand side of the expression is exactly `type(uint112).max`, as if the result is greater an overflow would occur and the transaction would revert without emitting the `OverflowsPrincipalOfTotalSupply()` revert message.

Customer's response:

L-2. Proposal fee is not returned to the proposer after the RESET event.

Impact: Low
Probability: Low

Description: In case of RESET event, active or unexecuted standard proposals will not return the fee because the StandardGovernor.sol will be replaced by the new one and thus will revert on proposal executions.

Customer's response:

L–3. _lastSyncs array is redundant, resulting in waste of gas with any access or write operation.

Impact: Low
Probability: N/A

Description: The _lastSyncs storage mapping, which holds the epochs array for every account, has become redundant, as it is a perfect mirror of the starting epochs of the _balances mapping, for every account. In other words, the following always holds, for every account:
Forall uint256 index.
_lastSyncs[account][index].startingEpoch == _balances[account][index].startingEpoch

Hence, the _lastSyncs doesn't serve any special purpose that cannot be already done by the _balances epochs.

Recommendation: Remove the _lastSyncs array from the contract storage, and replace its usage everywhere in the code with the equivalent epochs of the _balances array.

Customer's response:

Informational Severity Issues

INFO-1. Lack of EIP-712 compliance: using keccak256() directly on an array or struct variable

Impact: Info
Probability: N/A

Description: Directly using the actual variable instead of encoding the array values goes against the EIP-712 specification. Note: OpenSea's Seaport's example with offerHashes and considerationHashes can be used as a reference to understand how arrays of structs should be encoded.

Customer's response:

INFO-2. Array lengths not checked

Impact: Info
Probability: N/A

Description: If the length of the arrays are not required to be of the same length, user operations may not be fully executed

Customer's response:

INFO-3. Certain functions should not be marked as payable

Impact: Info
Probability: N/A

Description: execute() functions in StandardGovernor.sol and ThresholdGovernor.sol don't need to be marked as payable. And the following fragment can be deleted in BatchedGovernor.sol

Customer's response:

INFO-4. Unnecessarily small uint type for an incremented variable

Impact: Info
Probability: Low

Description: In MinterGateaway.sol, the size of _mintNonce and _retrievalNonce is uint48. While calling the respective incrementing functions `2**48` times to cause an overflow is unrealistic at the time being, It is not completely out of the realm of possibility that something like that might be possible in the future if the blockchain will become more efficient, while setting the type to be of a slightly larger size would have no negative impact on gas.

Customer's response:

INFO-5. Possible gas griefing when emitting voting event

Impact: Medium
Probability: N/A

Description: It's possible to cast a vote with a reason string through a relayer. The string is later included in the VoteCast event. However, the string can be arbitrarily long which could make emitting such an event very gas costly for relayer.
Note: We mark this issue as **Informational** because the client doesn't support vote relaying on behalf of users at the moment of writing this report.

Customer's response:

certora

M^0

Formal Verification

Assumptions and Simplifications

General Assumptions

A. We assume that all arrays are at most max(uint128) long.
B. The maximum possible timestamp corresponds to the beginning of the year 2525.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

ZeroToken (TTG)

Assumptions

- Any loop can iterate at most 2 times.
- We assume no overflow reverts for minting events.

Properties

Rule Name	Description
VotingEpochsAreMonotonic	The starting epochs of the voting powers array elements are strictly monotonic.
DelegatesEpochsAreMonotonic	The starting epochs of the delegatees array elements are strictly monotonic.
BalancesEpochsAreMonotonic	The starting epochs of the balances array elements are strictly monotonic.
TotalSuppliesEpochsAreMonotonic	The starting epochs of total supplies array elements are strictly monotonic.
LatestEpochIsAtMostCurrentEpoch	The latest epoch for every array is at most the current epoch.
TotalSupplyIsNonZero[1]	The total supply of zero tokens is never zero, in any epoch.
pastTotalSupplyIsImmutable	The total supply in past epochs is immutable.
pastBalanceIsImmutable	The balances in past epochs are immutable, for every account.

[1] Could be violated right after the constructor if an empty accounts array is provided, or if all amounts are zero.

pastDelegateeIsImmutable	The delegatees in past epochs are immutable, for every account.
mintingSuccessIsAccountIndependent	Minting tokens for different accounts cannot interfere with each other (make the other one revert).
mintingAmountIsAccountIndependent	Different accounts balances post-mint are independent.

PowerToken (TTG)

Assumptions

- Any loop can iterate at most 2 times.
- We assume no overflow reverts for any total supply update.
- We assume that the sum of voting powers for any two users cannot exceed max(uint240).

Properties

Rule Name	Description
synchingDoesntChangeBalance	**Calling sync() cannot change the value of balanceOf()** Was violated in previous commits, see: Issue C-2
delegatingPreservesVotingPower	**Calling delegate() preserves the total supply and the sum of voting power of the old and new delegatees.**
pastUnrealizedInflationIsImmutable	**The past unrealized inflation for any account is immutable** Was violated in previous commits, see: Issue H-1, Issue C-2
pastVotesAreImmutable	**The past votes for any account are immutable.** Was violated in previous commits, see: Issue H-1
pastBalanceIsImmutable	**The past balance for any account is immutable.** Was violated in previous commits, see:

	Issue H-1, Issue C-2
AllDelegateesHaveSynched	**Any delegatee has been synched at least once.**
BalancesEpochsAreMonotonic	**The starting epochs of the balances array are strictly monotonic.**
BootstrapEpochIsInThePast	**The bootstrap epoch is always in the past.**
DelegatesEpochsAreMonotonic	**The starting epochs of the delegatees array are strictly monotonic.**
DidntParticipateBeforeBootstrap	**No participation events are registered before the bootstrap epoch.**
EpochsAreAfterBootstrap	**All starting epochs are greater or equal to the bootstrap epoch.**
EpochsAreNotInTheFuture	**All starting epochs are bounded by the current epoch.**
LastSyncsAreBalancesEpochs[2]	**The last syncs epochs are identical to the balances epochs, for any account.**
NoDelegateeIfNoSyncs	**If an account hasn't yet synced, it has no delegatees.**
NoPastDelegatesAtUnsync	**The past delegatee of an account at an epoch the account hasn't synched yet is zero.**

[2] The property was verified before the introduction of the external sync() function. In the latest commit, the property was violated, but the lastSyncs array will be removed from the contract in future commits.

TotalSuppliesEpochsAreMonotonic	The starting epochs of the total supplies array are strictly monotonic.
VotingEpochsAreMonotonic	The starting epochs of the voting powers array are strictly monotonic.
markParticipationIsUserIndependent	Marking participation for a user doesn't change the balance of voting power of another user.
markParticipationSucceesIsUserIndependent	markParticipation() success (revert conditions) is user-independent.
onlyBuyingCanChangeTheEpochSpecificAuctionAmount	The only action that can change the auction amount in the same epoch is buy()
pastTotalSupplyIsImmutable	The past total supply is immutable.
transferDoesntIncreaseBalanceOfSender	No user can increase his balance by calling transfer().

StandardGovernor (TTG)

Assumptions

- Any loop can iterate at most 2 times.
- The zero address is never a msg.sender or a valid signer of any message.
- We assumed a simple implementation of ERC20 tokens in all external calls to tokens transfer.
- We assumed PowerToken.markParticipation() doesn't revert unexpectedly.
- The number of proposals at any epoch < max(uint256)

Properties

Rule Name	Description
pastEpochIsNeverCurrentEpoch	Any query of the past total supply / past votes satisfies epoch < current epoch = clock()
VoteStartIsNotInTheFarFuture	The vote start of any proposal is at most the current epoch + voting delay.
ExecutedHasNonZeroVoteStart	An 'executed' proposal has a non-zero vote start.
ActiveHasNonZeroVoteStart	An 'active' proposal has a non-zero vote start.
feeIsTakenOnSubmision	Upon calling to propose(), the proposer pays the proposalFee() in cashToken(), to the contract.
onlyOneProposalStateChangesAtOnce	Any operation can instantly change the state of only one proposal at a time.
queryOfStateCannotRevert	If a proposal exists, querying its

	state never reverts.
votingCannotInstantlyChangeTheState	Casting votes for any (active) proposal, cannot instantly change its status, thus it remains ‘active’.
canAlwaysVoteInVotingEpoch	If a proposal is active and the voter has not yet voted and the support type is ‘Yes’ or ‘No’ then casting a vote cannot revert.
cannotVoteWithoutVotingPower	An actor without voting power in the previous epoch cannot change any proposal vote count.
proposalSucceedsOnlyWhenMajorityReached (1)	A proposal can only turn to the ‘succeeded’ state if its ‘Yes’ count exceeds its ‘No’ count.
proposalSucceedsOnlyWhenMajorityReached (2)	A proposal can only turn to the ‘defeated’ state if its ‘No’ count is at least its ‘Yes’ count.
executedIsTerminal	The 'executed' state is a terminal state of any proposal.
canceledIsTerminal	The 'canceled' state is a terminal state of any proposal.
defeatedIsTerminal	The 'defeated' state is a terminal state of any proposal.
expiredIsTerminal	The 'expired' state is a terminal state of any proposal.

succeededIsAlmostTerminal	The 'succeeded' state can turn only to ‘executed’, or remain ‘succeeded’'.
pendingIsPrimal	The ‘pending’ state is a primal state.
pendingCanBecomeActiveOnly	The ‘pending’ state can turn only to ‘active’, or remain ‘pending’.
castVotesIsAssociative	castVotes() is proposal-associative.
VotedIfProposalExists	If any user has voted for a proposal, then its vote start must be non-zero.
cannotVoteTwice	If a user has voted once for a proposal, no operation can grant him the option to vote again.
votingCannotFrontRunEachOther	Casting votes with castVote() by different actors cannot interfere with each other (make the other revert).
cannotMarkZeroAddressAsParticipant	The zero address cannot be marked as a participant.
cannotChangeVotesBeforeExecution	Before any proposal execution, no operation in the same block can change its votes.

MinterGateway (Protocol)

Assumptions

- Any loop can iterate at most 2 times.
- We assume no overflow reverts for any arithmetic operations
- For earning address: rawBalance < type(uint112).max
- For non-earning address: rawBalance < type(uint112).max * type(uint128).max
- The vault is non earning
- The vault is not any of the following: the zero address, the minterGateway, the MToken or the TTGRegistrar contracts
- The maximum possible timestamp corresponds to the beginning of the year 2100
- MinterRate and EarnerBaseRate < 400% (40_000 BPS APY)

Properties

Rule Name	Description
totalOwedMExceedsTotalMSupply	totalOwedM >= totalMSupply
totalOwedMCorrectness	totalOwedM = totalActiveOwedM + totalInactiveOwedM
totalMSupplyCorrectness	totalMSupply = totalNonEarningMSupply + totalEarningMSupply
mintersCannotMintMUnlessAreOvercollateralized	Minters cannot generate M unless their Eligible Collateral, tracked by Collateral Value, is sufficiently high
collateralOfMinterIsZeroIfMinterDoesNotCallUpdate	If a minter does not call update during the required interval, their eligible collateral is assumed 0
minterMustWaitMintDelayTimeToMintM	A Propose Mint call cannot succeed before MintDelay time has elapsed

minterStatesCannotBeContradicting	It is not possible that minter can be both deactivated and active
consecutiveCallOfUpdateIndexCannotIncreaseMToken	In two consecutive calls at the same timestamp of updateIndex(), the second call should not increase the MToken balance of the ttgVault or the total supply
burnReducesOnlyTheBalanceOfMsgSender	Only the balance of msg.sender can be reduced when calling burnM()
validatorCanAlwaysFreezeAnyMinter	Any validator can always freeze any minter
validatorCanAlwaysCancelAnyExistingMintId	Any validator can cancel any existing mintId
mgUpdateIndexZeroesExcessOwedM	After calling updateIndex() the excessOwedM() should be zero

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.