OpenZeppelin | security

MZero Final Audit

February 15, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-01-08 To 2024-02-09
Languages	Solidity

Total Issues	22 (0 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (0 resolved)
Medium Severity Issues	6 (0 resolved)
Low Severity Issues	7 (0 resolved)
Notes & Additional Information	8 (0 resolved)

Scope

We audited the following repositories:

- MZero-Labs/protocol repository at commit 3499f50ff3382729f3e59565b19386ba61ef8e36
- MZero-Labs/ttg at commit a8127901fa1f24a2e821cf4d9854a1aa6ac8088c
- MZero-Labs/common at commit 4a37119f2da946c6d8ad7b9a70dfdd219225115b

All the contracts and interfaces present in the `src` folder of these repositories were in the scope of the audit.

System Overview

MZero is an EVM-compatible, immutable protocol that allows permissioned actors to mint and burn *M*, a rebasing ERC-20 token.

There are three permissioned actors in this system:

- Minters who mint M-tokens against their collateral
- Validators who ensure that the collateral deposited in the *Eligible Custody Solution* (ECS) matches the minter's the on-chain collateral value
- Yield earners who earn interest on their M-token balance

A minter can mint M-tokens backed by US treasury bills if they deposit sufficient *Eligible* collateral in an off-chain ECS. The validators verify the value of the off-chain collateral and provide signatures to the minter. The minter then uses the validator signatures to update their on-chain collateral value by calling the `updateCollateral` function. The number of unique signatures required to update collateral on-chain dependends on the `threshold` parameter. Minters are required to update their collateral on-chain within the `Update Collateral Interval` of their last update and are penalized if they fail to do so. The minters are also penalized if they become undercollateralised.

Once the collateral is updated, the minter can call the `proposeMint` function with the mint amount and the mint address as the user inputs. This function first ensures that the mint amount will not make the minter undercollateralized and then generates a `mintId`. A minter can have only one `mintId` at a time. A delay, named `mintDelay`, is added between the creation of a mint proposal and the actual minting of the M-tokens. This is the time provided to the validators to act on malicious minting requests by canceling them. If the `mintDelay` has passed, the minter can call the `mintM` function to execute the mint. This function can only be called until `mintTTL`, after which the `mintId` expires. The system checks for sufficient collateral again, mints the M-tokens, and updates the index. Any protocol user can repay the debt of a `minter` and burn their M-tokens by calling the `burnM` function. To retrieve the collateral from the ECS, a minter needs to call the `proposeRetrieval` function to generate a retrieval Id. This function ensures that the collateral retrieval will not make the minter undercollateralized. Once the retrieval Id is generated, the minter can use it to withdraw their off-chain collateral from the *ECS*.

If governance whitelists a minter, a call to the `activateMinter` function can be made to activate the minter in the protocol. Similarly, if governance removes a minter from the whitelist, it is reflected in the protocol by calling the `deactivateMinter` function. Any account with an M-token balance, that is whitelisted by the governance, can opt-in or opt-out of earning interest on their balance. The earner's rate is calculated by an earner rate model contract. The minter rate is calculated by a minter rate model contract. These rate model contracts are upgradeable by the governance. All important parameters of the protocol such as the list of approved minters, collateral-to-mint ratio, mint delay, the interest of the yield earners, etc. are set by governance and read using the `TTGRegistrarReader` library which calls the `Registrar` contract.

MZero's novel two-token governance (TTG) is oblivious to the existence of the protocol. The protocol gets information from the TTG using a pull mechanism. TTG, as its name indicates, has two governance tokens: Zero token and Power token. The Power token holders are the managerial class and deal with the day to day activities of the system. The Zero token holders have more of an oversight role. TTG operates on the concept of epochs. Each epoch has two sub-epochs: a voting epoch of 15 days and a transfer epoch of 15 days. There are three types of proposals in the system: Standard proposals, Power threshold proposals, and Zero threshold proposals.

Only Power token holders can vote on a standard proposal. Voting on a standard proposal happens in a voting epoch and the proposals are executed in the next transfer epoch. A proposal created during a voting or a transfer epoch gets voted on in the next voting epoch. Creating a proposal requires depositing proposal fees. If the proposal succeeds and is executed then the proposal fee is refunded to the proposer otherwise it is forfeited. A standard proposal can be passed by a simple majority of Power token holders. There is no quorum requirement. The `StandardGovernor` deals with the standard proposals.

Power threshold proposals can only be voted on by the Power token holders. When proposed this type of proposal becomes eligible to be voted on instantly and its voting period lasts till the end of the next epoch. For a Power threshold proposal to pass, a *threshold* amount of the total supply of Power token holders need to vote *yes* on it. The `EmergencyGovernor` deals with the Power threshold proposals. Zero threshold proposals are similar to Power threshold proposals as they can be voted on instantly, their voting period lasts till the end of the next epoch, and the threshold amount of Zero token holders need to vote *yes* on it. The `ZeroGovernor` deals with the Zero threshold proposals.

The Power token is a rebasing token. Every voting epoch, its supply increases by 10%. To be eligible for the rebase, one has to vote on all the standard proposals of that voting epoch. The share of the token holders who did not vote on all proposals is auctioned in the transfer epoch using a Dutch auction. The Power token holders also receive a pro-rata share of a fixed inflation of Zero tokens. TTG only has *yes* and *no* votes, there is no option to *abstain*. Once a token holder votes on a proposal they cannot change their vote. The proposals of the three governors of TTG only allow certain types of calldata and can only call themselves to execute the calldata upon the passing of a proposal. The `Registrar` stores all data related to the protocol like the *minter list*, *earner list*, *earner rate*, *mint ratio*, etc., and can only be called by the `StandardGovernor` or the `EmergencyGovernor`. The proceeds of the protocol and the governance are sent to the `DistributionVault` from where Zero token holders can pro-rata claim a share of the proceeds.

Security Model and Trust Assumptions

During the course of this audit, it was assumed that the validators and the ECS operators were honest entities and would always work in the interests of the protocol. It was also assumed that the protocol parameters set by the TTG will always be in the best interest of the protocol and its users. However, the off-chain components of the system pose certain game-theoretical attack risks to the protocol. For instance, a malicious minter can get approved by the TTG, store $1M worth of collateral in the ECS, ask a validator to verify and sign their collateral, and call the `updateCollateral` function to reflect the same on-chain.

The malicious minter can then create a retrieval ID for $600k of collateral, while convincing the validator that they do not wish to withdraw the $600k worth of collateral from the ECS. This is a valid action according to the whitepaper. The validator then checks the available collateral and signs the retrieval ID along with the $1M collateral value. However, the malicious minter goes against their word and informs the ECS operator that they want to withdraw $600k by providing the retrieval ID. The operator calls the `pendingCollateralRetrievalOf` function of the `MinterGateaway` contract to confirm the same.

Since the minter has not called `updateCollateral()` yet, the retrieval ID is not deleted and the minter is able to withdraw the $600k from the ECS. The minter then calls `updateCollateral()` which deletes the retrieval ID and makes his `totalPendingRetrievals` zero. The minter then mints $900K worth of M-tokens and sells it on the market. Consequently, the minter is able to make a profit of $500k since they deposited $1M and are able to get a value of $1.5M out of the protocol ($900k through M-tokens and $600k through withdrawal of collateral).

Privileged roles

In essence, almost everything that happens on the protocol is controlled by the TTG. TTG approves the list of minters and validators who can interact with the `MinterGateaway` contract.

There are two permissioned roles in the `MinterGateaway` contract:

- `minters`: Minters are responsible for the circulation of M-tokens. Only active minters can update their collateral and propose retrieval. Active minters that are not frozen can propose mints and mint M-tokens.
- `validators`: Approved validators can cancel mint requests and freeze minters. In addition, there are off-chain ECS operators that have the privilege of physically storing and removing users' collateral from the ECS.

The `mint` and `burn` functions in the `MToken` contract can only be called by the `MinterGateaway` contract.

Within the TTG system, each governor has a set of powers.

The `ZeroGovernor` can:

- Deploy the `PowerToken`, `StandardGovernor`, and `EmergencyGovernor` contracts when resetting the Power token to Power token holders or Zero token holders
- Set the Power threshold proposal ratio
- Set the Zero proposal threshold ratio
- Set the cash token of the system and proposal fee of standard proposals

The `EmergencyGovernor` contract can:

- Set the Standard proposal fee
- Add and delete account from lists, and modify key-value pairs of the `Registrar` contract

The `StandardGovernor` can:

- Set its proposal fee in the `StandardGovernor` contract
- Call the `markNextVotingEpochAsActive`, and `markParticipation` functions of `PowerToken` contract
- Mint Zero tokens
- Add and delete account from lists, and modify key-value pairs of the `Registrar` contract

Protocol Invariants

The main invariant of the protocol is that `totalOwedM >= totalSupply`. Due to the exponential nature of `totalActiveOwedM` and `totalEarningSupply` (constituents of `totalOwedM` and `totalSupply` respectively) , there are two scenarios where this invariant can break:

- Earner rate > minter rate
- `totalEarningSupply` > `totalActiveOwedM`

Taking the above into consideration, consider adding a protocol invariant check to protect the protocol against overprinting of M if the current or future earner rate model produces errant values or if the protocol is not interacted with for a long time.

Possible Protocol Invariant Check

The `currentIndex()` function needs to be overridden in the `MToken` contract and the following invariant needs to be checked at the end of the function before the execution ends: `totalOwedM >= totalNonEarningSupply + principalOfTotalEarningSupply * current calculated index`. This is because `currentIndex()` is the only function that can cause an increase in `totalSupply` without causing an increase in `totalOwedM` since it is used in the calculation of `totalEarningSupply()`. Other functions, such as mint and burn increase or decrease `totalOwedM` and `totalSupply` equally. Since `currentIndex()` is a `view` function and multiple functions like `balanceOf()` and `totalEarningSupply()` depend on it, this function must not revert if the invariant is found to be broken. Instead, it should scale down the index returned by the function. There are two ways to achieve this:

1. The `currentIndex()` function returns the `_latestIndex`.
    - This is not in favor of the earners as the cashflow generated since the `_latestUpdateTimestamp` by the owed M will just be minted to `DistributionVault` when `updateIndex()` of `MinterGateway` is called.
2. The `currentIndex()` returns the maximum possible index that will not cause the overprinting of M. This can be achieved by subtracting the `totalNonEarningSupply`

from `totalOwedM`, multiplying it by `EXP_SCALED_ONE`, and then dividing the entire thing by `principalOfTotalEarningSupply`.

- The advantage of this approach is that it is fairer to the earners. They will not lose their earner interest and the invariant will also remain protected.
- This method will allow earners to earn interest even when `totalActiveOwedM_ < totalEarningSupply_`.

High Severity

H-01 Signature Replay Attack Possible in `MinterGateway`

The `_verifyValidatorSignatures` function of the `MinterGateway` contract is called within the `updateCollateral` function to verify if the validator signatures provided by the `minter` for their collateral are valid. However, if a signature has its `timestamp` as zero, this function replaces it with `block.timestamp`.

If the validator threshold is one, the `_minterStates[minter_].updateTimestamp` is stored as `block.timestamp`. If the validator threshold is greater than one, this zero timestamp signature can be used along with other signatures and the lowest non-zero timestamp will be stored as `_minterStates[minter_].updateTimestamp`. If all the signatures have a zero timestamp then `_minterStates[minter_].updateTimestamp` will be stored as `block.timestamp`. This quirk in `_verifyValidatorSignatures` allows a minter to essentially reuse zero-timestamp signatures.

Consider ignoring any signature which has a timestamp less than `_minterStates[minter_].updateTimestamp` in the `_verifyValidatorSignatures` function.

Medium Severity

M-01 Claiming of Distribution From `DistributionVault` Can Be Gamed

For reward distribution, the latest balance of a user is stored for each epoch and rewards are distributed according to the epoch balance. Each user can claim the reward for the last epoch as soon as the next epoch begins, pro-rata based on their Zero token balance in the last epoch.

This can lead to the following scenario:

- A user can borrow or buy a large amount of Zero tokens one block before the new epoch begins.
- This new balance will be stored as their balance for that epoch.
- In the next block (after the epoch has begun), they can claim a disproportionate amount of rewards of the last epoch and repay their debt or sell back the Zero tokens.
- This attack can be carried out in two blocks and since blocks on Ethereum are ~12 seconds, the malicious user would only need to borrow or buy tokens for less than a minute.

Possible Mitigation

Use `shares` for reward distribution instead of balances. The claimable amount can be calculated as the `shares` of a user divided by the `total shares` of that epoch. `Shares` of a user can be calculated by multiplying the user's balance at the epoch beginning by the duration of the epoch. Then, whenever tokens are transferred, the delta multiplied by the time left in the epoch is added or subtracted from the user's `shares` depending on whether they sent or received the tokens. `Total shares` of an epoch will be equal to the total supply at the epoch beginning multiplied by epoch duration. `Total shares` will increase only when new tokens are minted.

- Share-based accounting can also be used for voting to prevent governance attack vectors like borrowing or buying a large amount of tokens before the start of an epoch.
- The maximum balance would need to be capped at `uint232` otherwise multiplication with 15 days will overflow the `uint256` container.

Consider using the described mitigation strategy to prevent the gaming of reward distribution.

M-02 M-Token and POWER Token Will Have Integration Issues on DEXs

Anyone can steal the rebases of M-token or POWER token if they are part of a pool in a Uniswap v2 fork by calling the skim() function. Similarly, a lot of DeFi protocols are not equipped to handle rebasing tokens and this can lead to a loss for the protocol, or the M-token or Power token holder.

Consider having a wrapped version of these tokens ready at the time of launch to mitigate any issues faced when integrating rebasing tokens with the wider DeFi ecosystem.

M-03 `DistributionVault` Cannot Handle Rebasing Tokens

M-tokens are sent to the `DistributionVault` by the `MinterGateway` and the `PowerToken` contract. If the `DistributionVault` becomes an active earner, its M-token balance will start increasing. The reward for an epoch is calculated by subtracting the last recorded balance of the vault from the current balance of the vault. Since the current balance of the vault will increase due to rebasing without any new M-tokens being transferred to it, the accounting of this method is incorrect.

Furthermore, if a user does not claim their M-token rewards of a particular epoch, the rebasing that their portion of the reward will receive would be entitled to everyone. This will essentially eat into their rewards.

Listed below are the possible mitigations, in order of preference:

- Wrap the M-tokens and then send them to the `DistributionVault` contract. An example of this approach is the `stETH` token which is integrated in DeFi protocols by wrapping it as `wstETH`.
- Add logic to the `DistributionVault` to handle rebasing tokens. In the case of M-tokens, this can be done by storing the `principal` of M-token by calling the `currentIndex()` on the `MToken` contract and dividing the `present amounts` by it. Epoch distribution amount should only be updated if `current balance` divided by `currentIndex()` is greater than the last stored `principal`.
- Ensure that the `DistributionVault` can never become an earner. However, this is not in the best interest of the users.

Consider implementing one of the possible mitigations to ensure correct reward accounting in the `DistributionVault`.

M-04 Rewards of an Epoch Are Claimable by Zero holders of a Future Epoch

Zero token holders claim their rewards, which are calculated per epoch, from the `DistributionVault`. The Zero token balance of a user in a past epoch is used to calculate their pro-rata share of the rewards of that epoch. Rewards in the form of tokens are sent to the `DistributionVault` when:

- `totalOwedM_` becomes higher than `totalMSupply_`

- When a standard proposal fails or expires
- When Power tokens are bought in an auction

Sending tokens to the `DistributionVault` does not guarantee that Zero token holders of that epoch will be able to claim their share of rewards. The `distribute` function needs to be called by someone to make unaccounted rewards claimable for that epoch.

Any tokens sent to the `DistributionVault` after the last call to `distribute` in an epoch are eligible to be accounted for in the rewards of future epochs. Since the distribution of Zero holders can change every epoch, the unaccounted rewards of an epoch can be claimed by the Zero holders of future epochs.

Consider calling the `distribute` function after making transfers to the vault to ensure that the rewards of the current epoch are claimable by Zero holders of the current epoch and not the Zero holders of future epochs.

M-05 Denial of Service Due to Transaction Running Out of Gas

Multiple functions in TTG are susceptible to running out of gas during execution if a user has not interacted with the protocol in a long time. As the protocol has been built to function for decades, this could lead to DoS for multiple users when they are trying to transfer, delegate, vote, check balance, claim rewards, etc.

- The `PowerToken` contract needs an externally-callable `sync` function which allows users to sync `x` epochs since the last synced epoch. Consider adding this function to enable users to sync and potentially unbrick their bricked accounts.

- The `_getValueAt` and `_getDelegateeAt` internal functions in the `EpochBasedVoteToken` contract iterate over an unbounded loop. A linear search is performed from the last recorded epoch in an `AccountSnap` or `AmountSnap` array. These functions are used to fetch data at:

- the current epoch
- the epoch before voting (within the last three epochs)
- an arbitrary epoch

Consider making a binary search version of the two functions for the third case.

- Similarly, the `_hasParticipatedAt` function uses linear search to find whether a user voted in an epoch. Consider using binary search to reduce gas costs.

- The `_getUnrealizedInflation` function is very gas-heavy and susceptible to running out of gas. Consider traversing the `_participations` array from the relevant point, counting the epochs and using the compound interest formula at the end to calculate the unrealized inflation.

- Consider duplicating the functionality of the `getClaimable` function in the `_claim` function to make it easier for users to claim rewards for more epochs without running out of gas.

M-06 Reward Tokens Can Get Stuck in the `DistributionVault`

The `getClaimable` function calculates the pro-rata share of the Zero token holders for the distribution of rewards. If Zero tokens are sent to the vault, the vault becomes eligible for a portion of rewards along with the other token holders. Since no functionality allows the `DistributionVault` to claim its share of tokens, these tokens will be locked in forever. In addition, every epoch, a portion of any token (e.g., M-token or wETH) that is sent to the `DistributionVault` will also get stuck since the vault would always have a claim to the rewards. This will result in the loss of funds.

Consider subtracting an epoch's Zero token balance of the `DistributionVault` from that epoch's `totalSupply` when calculating the claimable amount of a user.

Low Severity

L-01 Lack of Validation

Throughout the codebase, there are instances of missing checks:

- The `transferFrom` function in the `ERC20Extended` library should check if `amount_ > spenderAllowance_` and revert with a meaningful error message instead of reverting at the arithmetic underflow.

- The `_verifyValidatorSignatures` function in the `MinterGateway` contract should check if the `threshold_` is greater than 0.

- The `proposeMint` function in `MinterGateway` contract does not validate that the `destination_` address is not `address(0)`.
- The `castVotes`, `castVotesBySig`, and `castVotesBySig` functions in the `BatchGovernor` contract allow the user to cast votes on multiple proposals in single function call. These functions accept `proposalIds_` and `support_` arrays as user inputs but fail to check if the length of these arrays is the same.
- According to the MZero Protocol Whitepaper, consider adding a check to the constructor of the Zero token which ascertains that the initial supply minted is equal to 1 billion.

For input sanitization and reducing the attack surface of the codebase, consider implementing these checks.

L-02 Minter Disapproved by TTG Can Continue to Interact With the Protocol

The `MinterGateway` contract verifies whether a `minter_` is approved by governance within the `isMinterApproved` function which reads the value from the `TTGRegistrarReader`. This `isMinterApproved` function is called from the `activateMinter` and `deactivateMinter` functions that are used to activate or deactivate minters respectively. These functions set the `isActive` flag for the `minter_` in the `_minterStates` mapping as true if the minter is active and false if inactive. This `_minterStates` mapping is then used in the rest of the contract to verify if the minter is active.

There can be a scenario where the governance decides to disapprove a minter. Until the `deactivateMinter` function is called in the `MinterGateway` contract, the disapproved minter can continue to interact with the protocol, as long as their `isActive` flag in the `_minterStates` mapping is true.

Consider calling the `isMinterApproved` function within the `_revertIfInactiveMinter` function to ensure that the protocol is always aware of a disapproved minter.

L-03 Incorrect Comments

Consider fixing the following comments that were found to be incorrect or misleading:

- At line 103 of the `IBatchGovernor` interface, the docstring above the `castVotesBySig` function states that a signer can cast votes via an arbitrary signature. However, this function allows anyone to cast a vote on behalf of the signer if they have a valid signature.

- At line 118 of the `StandardGovernor` contract, the comment wrongly asserts that standard proposals can be executed in epoch N+2, where N is the epoch in which the vote took place.

L-04 `nextDeploy()` Returns Incorrect Contract Address

The `nextDeploy()` function in `EmergencyGovernorDeployer.sol`, `PowerTokenDeployer`, and `StandardGovernorDeployer` wrongly computes the next deployment address. It incorrectly increments the nonce when calculating the deployment address.

Consider using the nonce as it is to compute the deployment address.

L-05 EIP-1271 Signature Replay Attack Possible

Most smart contract wallets' `isValidSignature` function is just a wrapper over `ECDSA.recover`. If the output of the `ECDSA.recover` matches the owner, then the magic value `0x1626ba7e` is returned. If a user is the owner of multiple smart contract accounts and wants to interact with `claimBySig`:

- They can sign a message entitling the `destination` address to the rewards of their EOA address.
- There is nothing stopping the `destination` address from replaying that signature substituting `account_` with a smart contract where the user is the owner as `account_` is not used in calculating the digest.
- Likewise, if the EOA signs a message for one of their smart contract accounts, it can be replayed for their EOA and other smart contract accounts.

Consider making the protocol secure against this type of attack by:

- Using `account_` to compute the digest in `claimBySig`
- Using `voter_` to compute the digest in `castVoteBySig`
- Using `voter_` to compute the digest in `castVotesBySig`
- Using `account_` to compute the digest in `delegateBySig`

- Using `validator[index]` to calculate the digest in `_verifyValidatorSignatures`

L-06 Zero Amount of Power Tokens Can Be Minted

If the `buy` function is called with `minAmount_` and `maxAmount_` set to zero, it leads to the minting of zero `Power` tokens.

Consider adding a check to the `buy` function to validate that `minAmount_` is greater than zero.

L-07 Zero Amount of M-tokens Can Be Minted or Burned

The `burnM` function in the `MinterGateway` contract is used for the burning of M-tokens. It can burn zero amount of M-tokens if:

- The `maxPrincipalAmount_` or `maxAmount_` is zero
- The `minter_` does not owe any M-tokens
- The `minter_` does not exist

Similarly, the `mintM` function does not check that the `amount_` value is zero and mints zero amount of M-tokens.

Consider adding checks to the `mintM` and `burnM` functions to prevent minting or burning of zero amount of M-tokens.

Notes & Additional Information

N-01 Todo Comments in the Code

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

Multiplies instances of TODO/Fixme comments were found in the codebase:

- The `TODO` comment at line 303 of `BatchGovernor.sol`
- The `TODO` comment at line 12 of `ThresholdGovernor.sol`

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

N-02 Typographical Errors

To improve the readability of the codebase, consider fixing the following typographical errors:

- At line 26 of the `IERC5805` interface, "*who's*" should be "*whose*" .
- At line 46 of the `ERC5805` contract, "*verifier*" should be "*verified*" .
- At line 40 of the `IBatchGovernor` interface, "*no this*" should be "*not this*" .
- At line 12 of the `EpochBasedInflationaryVoteToken` contract, "Specifically,a nd only" should be "Specifically, and only".

N-03 Lack of Security Contact

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

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

N-04 TTG Is Not Fully Compatible With Community Tools

The omission of `string calldata reason` as an input parameter from the `castVoteWithReason` function changes the function signature, making it incompatible with Tally and possibly other tools.

Consider adding and using the missing parameter in the `castVoteWithReason` function.

N-05 Naming Suggestions

To favor explicitness and readability, consider renaming the `_getTotalSupply` function in the `EpochBasedVoteToken` contract to `_getTotalSupplyAtEpoch`.

N-06 Unused State Variables

In the `BatchGovernor` contract, the `quorumRatio` variable in the `Proposal` struct is unused.

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

N-07 Unused Import

The `SignatureChecker` import in `ERC3009.sol` is unused. Consider removing this and any other instances of unused imports to improve the overall clarity and readability of the codebase.

N-08 Unused Function With `internal` Visibility

In `EpochBasedVoteToken.sol`, the `subUnchecked` function is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused function.

Conclusion

The MZero team has devised a coordination protocol to allow privileged actors to mint and burn overcollateralized quasi-stablecoins (M-tokens) backed by US treasury bills. M-token holders are paid interest using a rebasing mechanism. To govern the protocol, the team has also built a novel two-token governance mechanism where Power token holders manage the daily activities of the protocol while Zero token holders oversee the functioning of the governance.

One high-severity and several medium-severity issues were found. Other than this, several lower-severity issues were reported that mainly identified improvement opportunities in the overall quality of the codebase. Throughout the audit, the MZero team was very responsive and provided us with extensive information about the project.