PUBLIC

Code Assessment

of the Protocol and Governance Smart Contracts

April 24, 2024

Produced for

M^0

by

CHAINSECURITY

Contents

1 Executive Summary

Dear M^ZERO team,

Thank you for trusting us to help M^ZERO Labs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Protocol and Governance according to Scope to support you in forming an opinion on their security risks.

M^ZERO Labs implements a stablecoin (`MToken`) backed by real-world assets, like T-bills, along with a Two-Tokens Governance system (TTG).

The most critical subjects covered in our audit are asset solvency, functional correctness, and precision of arithmetic operations. Security regarding asset solvency is high. Security regarding functional correctness is satisfactory, however users should be aware that the rare event of redeployment of `PowerToken` might cancel their token transfers or inflations in the last two epochs before the redeployment event, see Side-effects of Resets. Precision of arithmetic operations is improvable due to the rounding errors in the `PowerToken` that accumulate over time, see Effects of Roundings in PowerToken.

The general subjects covered are code complexity, use of uncommon language features, and gas efficiency. The code-base extensively employs assembly code to manually compute storage slots for array entries. While no specific issues have been detected with this usage, it is worth noting that this approach bypasses the safety features implemented by Solidity. The code-base can be more efficient in terms of gas, see Gas Optimizations.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	6
• Code Corrected	3
• Code Partially Corrected	1
• Risk Accepted	1
• Acknowledged	1
Low-Severity Findings	22
• Code Corrected	14
• Specification Changed	1
• Code Partially Corrected	1
• Acknowledged	6

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Protocol and Governance `TTG`, `protocol`, and `common` repositories based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

TTG

V	Date	Commit Hash	Note
1	08 Jan 2024	a8127901fa1f24a2e821cf4d9854a1aa6ac8088c	Initial version
2	01 Mar 2024	9ef2198cc744f392bf87c15e134f27ec15aa5aaa	Version with fixes
3	15 Apr 2024	a4db6f1828f844b1bfb37348d37b2f8de9f70e37	Version with fixes
4	23 Apr 2024	47e3a9af8f5e26f1785f73e8bfa9095b27541627	Final version

protocol

V	Date	Commit Hash	Note
1	08 Jan 2024	3499f50ff3382729f3e59565b19386ba61ef8e36	Initial version
2	01 Mar 2024	2f12e10a753121d6de6f99847875d9011645770c	Version with fixes
3	23 Apr 2024	58e0ccc9211a97abd7f21c5499740f6d8b597435	Final version

common

V	Date	Commit Hash	Note
1	08 Jan 2024	4a37119f2da946c6d8ad7b9a70dfdd219225115b	Initial version
2	01 Mar 2024	f4473275da04ab839e777c67122c00e495601bf5	Version with fixes
3	15 Apr 2024	e809402c4cc21f1fa8291f17ee0aee859f3b0d29	Final version

For the solidity smart contracts, the compiler version `0.8.23` was chosen.

For `TTG`, the following contracts are in the scope of the review:

```
DistributionVault.sol
EmergencyGovernor.sol
EmergencyGovernorDeployer.sol
PowerBootstrapToken.sol
PowerToken.sol
PowerTokenDeployer.sol
Registrar.sol
StandardGovernor.sol
StandardGovernorDeployer.sol
ZeroGovernor.sol
ZeroToken.sol
abstract:
    BatchGovernor.sol
```

```
    ERC5805.sol
    EpochBasedInflationaryVoteToken.sol
    EpochBasedVoteToken.sol
    ThresholdGovernor.sol
    interfaces:
        IBatchGovernor.sol
        IERC5805.sol
        IERC6372.sol
        IEpochBasedInflationaryVoteToken.sol
        IEpochBasedVoteToken.sol
        IGovernor.sol
        IThresholdGovernor.sol
interfaces:
    IDeployer.sol
    IDistributionVault.sol
    IEmergencyGovernor.sol
    IEmergencyGovernorDeployer.sol
    IPowerBootstrapToken.sol
    IPowerToken.sol
    IPowerTokenDeployer.sol
    IRegistrar.sol
    IStandardGovernor.sol
    IStandardGovernorDeployer.sol
    IZeroGovernor.sol
    IZeroToken.sol
libs:
    PureEpochs.sol
```

For `protocol`, the following contracts are in the scope of the review:

```
MToken.sol
MinterGateway.sol
abstract:
    ContinuousIndexing.sol
interfaces:
    IContinuousIndexing.sol
    IMToken.sol
    IMinterGateway.sol
    IRateModel.sol
    ITTGRegistrar.sol
libs:
    ContinuousIndexingMath.sol
    TTGRegistrarReader.sol
rateModels:
    MinterRateModel.sol
    SplitEarnerRateModel.sol
    StableEarnerRateModel.sol
    interfaces:
        IEarnerRateModel.sol
        IMinterRateModel.sol
        IStableEarnerRateModel.sol
```

For `common`, the following contracts are in the scope of the review:

```
ContractHelper.sol
ERC20Extended.sol
ERC3009.sol
ERC712.sol
StatefulERC712.sol
interfaces:
    IERC1271.sol
    IERC20.sol
    IERC20Extended.sol
    IERC3009.sol
    IERC712.sol
    IStatefulERC712.sol
libs:
    SignatureChecker.sol
    UIntMath.sol
```

In (Version 2), the scope has been modified as follows:

- The file and contract `ERC712` in `common/` has been renamed `ERC712Extended`.
- A new interface `common/interfaces/IERC712Extended.sol` has been added.
- The rate model `protocol/rateModels/SplitEarnerRateModel.sol` was removed from the codebase.

In (Version 3), the following file was renamed:

- The file `StableEarnerRateModel` has been renamed `EarnerRateModel`.

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Third-party libraries, like `solmate`, are out of the scope of this review. Furthermore, the soundness of the financial model was not evaluated. Finally, the setup of the special-purpose vehicle (SPV) that holds the collateral was not in scope of this review, and it is assumed that the integration with smart contracts works always according to the specifications.

3 System Overview

This system overview describes the initially received version ((Version 1)) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

M^ZERO Labs offers a stablecoin (`MToken`) backed by real-world assets, like T-bills, along with a Two-Tokens Governance system (TTG). The protocol considers two main actors, Minters and Validators. The first have collateral in an SPV and can mint stablecoin against their collateral, the latter are trusted by the system and bear the responsibility of verifying Minters' offchain collateral value. The Governance is composed of two tokens, `PowerToken` and `ZeroToken`. Token holders can make proposals and vote on them to change some of the parameters of the system.

3.1 Protocol

3.1.1 MToken

The MToken contract is a stable coin backed by collateral locked in an off-chain entity. Minters can mint and burn MToken through the MinterGateway contract. Allowlisted users can earn interest by calling the startEarning function and their balance will increase over time. Any change in the total amount of MTokens earning interest triggers an update to the earner interest rate. The earner interest rate is calculated in external rate model contract set by Governance. The total supply of the token consists of all earning balances (incl. interest) and non-earning balances and is inflationary. A user can stop earning interest by calling the stopEarning function and the interest earned up to that point will be added to their balance. Governance can remove a earner from the whitelist and anyone can stop them earning interest. The whitelisting feature can be disabled by Governance, which would allow any M holder to switch freely between earning and non-earning states.

For non-earner the contract stores their real balance, while for earners the contract stores the principal amount:

$$principalAmount = \frac{realBalance}{index}$$

The index is calculated using the formula for Continuous compounding interest:

$$index_t = index_{t-1} * e^{earnerRate * \frac{\Delta t}{365days}}$$

The exponential function is approximated with the $R(4,4)$ Padé approximant around point 0.

The transfer and transferFrom function take the present amount as an argument and rounds it down to the principal when needed. Transfers can be divided into two types: in-kind transfers and out-of-kind transfers. In-kind transfers happen when the sender and the recipient are both earners, or both non-earners. In this situation, the rounded-up principal is exchanged for the former and the token amount is exchanged for the latter. Out-of-kind describes transfers where only one of the participants is earner. If the sender is the earner, its balance is reduced by the rounded-up principal amount and the recipient's balance is increased by the token amount. If the sender is the recipient, its balance is increased by the rounded-down principal amount and the sender's balance is decreased by the token amount. Note that integrators must be aware of unexpected balance values after mint and transfers, see Effects of Roundings in MToken.

M token has 6 decimals and supports EIP-2612 permits, via EIP-712 or EIP-1271 signatures, and transfers with authorization according to EIP-3009 (via ERC-712 signatures / ERC-1271 support).

3.1.2 MinterGateway

The MinterGateway contract is the access point for the minting or burning of MToken, it also stores the accounting of the debt. There are 2 types of actors that interact with the contract: Validators and Minters. Minters keep collateral in an off-chain entity (i.e. SPV) and mint MToken against their collateral. Validators periodically verify the state of the off-chain collateral and sign it.

In order to mint new M, minters have to:

1. update their collateral with updateCollateral() by providing enough EIP-712 UpdateCollateral signatures, signed by validators. The signatures count threshold is set by governance. The recorded collateral updated timestamp is set to the oldest timestamp of the signatures batch. Minters are expected to refresh their collateral value once a day (decided by governance). If a Minter does not refresh their collateral within the update interval, their collateral value is considered to be 0, making them undercollateralized if they had any debt.
2. open a minting request with proposeMint. The proposals have a unique ID and can be cancelled by any validator. Their future position must be over-collateralized for the call to succeed.
3. mint M for the open request ID, if their request is not cancelled or expired. The position must be over-collateralized for the call to succeed.

Minters can receive penalties in two cases:

- if they miss update intervals. The penalty is imposed upon interaction with `updateCollateral()`, and `burnM()` for active Minters. It is computed as $(nbrMissedUpdateIntervals * principalAmount) * penaltyRate$. The system charges a penalty at most once per missed interval.
- if they are undercollateralized. The penalty is imposed upon interaction with `updateCollateral()`. It is computed as $(\frac{maxAllowedDebt}{currentIndex} - principalAmount) * penaltyRate$, with $maxAllowedDebt = collateralValue * mintRatio$. The `mintRatio` is set by governance and is $0\% \leq mintRatio \leq 10'000\%$. Note that `collateralValue = 0` if a collateral update is missed.

The validators are trusted parties and are responsible for verifying the amount of collateral locked in the SPV and signing the update request. The whitelisted validators list is managed directly by the governance. The update requests are verified in the `updateCollateral` function and the collateral value is stored along with the update timestamp. All the collateral values must match in the batch, and the recorded update timestamp is the smaller of the batch. Minters should submit signatures ordered by validators' addresses.

Minters have to pay interest on their debt. The minter interest rate is calculated in an external rate model contract set by Governance. The contract stores the principal amount of the minted `MToken`:

$$principalAmount = \frac{realBalance}{index}$$

The index is defined analogously to the `MToken` contract:

$$index_t = index_{t-1} * e^{minterRate * \frac{\Delta t}{365days}}$$

The exponential function is approximated with the $R(4,4)$ Padé approximant around point 0. Any change to the amount of `MTokens` or collateral will trigger an update to the minter interest rate. The earner interest rate is assumed to depend on the minter interest rate, so it is updated with the minter index. With each update, the difference of the interest paid by minters and the interest owed to earners is calculated and minted to the governance's `DistributionVault`.

Minters are whitelisted entities. They must first be `activated` by governance vote. Validators can `freeze` Minters to stop them from minting `MTokens` for some period.

When Minters exit the system or as an ultimate punishment, Minters can be deactivated by governance. Once deactivated, a Minter cannot be reactivated. Deactivated minters do not accrue any interest, so the contract stores their balance in M token. After Minters are removed by governance anyone can `deactivate` them: Their owed `M` token balance stops accruing the minter interest and their collateral balance is deleted. As a kind of liquidation, `M` holders can `burn` their tokens to repay the `deactivated` Minters debt. The protocol assumes that burners can then start negotiations with the SPV to retrieve the underlying collateral.

Any `M` holders can burn tokens to lower the debt of a Minter at any point in time. For deactivated minters, that burns a part of their real balance. For active minters, their principal balance is lowered.

Minters can propose to retrieve collateral from the SPV when they have a high enough collateralization ratio. Collateral that is pending retrieval is not counted towards the collateralization ratio. Validators monitor on-chain the retrieval requests which emit event `RetrievalCreated` and include them in signature when they are processed. The collateral retrieval is then finalized by the `updateCollateral` function and collateral value is updated accordingly.

3.1.3 RateModels

The protocol uses two types of rate models: The *minter rate model* determines the interest paid by minters and the *earner rate model* determines the interest paid to earners. The contracts implement a `rate` function that returns the yearly Annual Percentage RATE (APR) of a market and is consumed by the `MToken` and `MinterGateway` contracts without further checks. M^ZERO Labs implements 3 rate models:

The `MinterRateModel` is the only interest rate model available for the Minters. It reads the minter interest rate from the `TTG Registrar` and returns it. An update to the minter interest rate in the registrar alone does not have an effect on the interest paid by minters. The minter interest rate is only updated once `updateIndex()` is called in the `MinterGateway` contract.

The `Earners` have two different kind of earner interest rate models: The `StableEarnerRateModel` and the `SplitEarnerRateModel`. Both models aim to make sure that the interest paid to earners will not exceed the interest paid by minters.

The `SplitEarnerRateModel` defines the earner interest as:

$$earnerRate = \min(baseRate, 90\% * minterRate * \frac{totalActiveOwed}{totalEarningSupply})$$

Here the `baseRate` is an interest rate set by governance, `totalActiveOwed` is the amount of `MTokens` owed to minters and `totalEarningSupply` is the total supply of `MTokens` that are earning interest.

We can define

$$totalInterestEarnedByMinters = totalActiveOwed * (e^{minterRate * \frac{\Delta t}{SECONDS\_PER\_YEAR}} - 1)$$

$$totalInterestPaidToEarners = totalEarningSupply * (e^{earnerRate * \frac{\Delta t}{SECONDS\_PER\_YEAR}} - 1)$$

and show with a linear approximation that for small values of $minterRate * \frac{totalActiveOWed}{totalEarningSupply} * \frac{\Delta t}{T}$:

$$totalInterestPaidToEarners \simeq 90\% * totalInterestEarnedByMinters \leq totalInterestEarnedByMinters$$

The `StableEarnerRateModel` derives a *safe rate* for earners s.t. the interest paid by minters is equal to the interest paid to earners over a fixed time period.

$$totalActiveOwed * (e^{(minterRate * \frac{\Delta t}{SECONDS\_PER\_YEAR})} - 1) = totalEarningSupply * (e^{(safeRate * \frac{\Delta t}{SECONDS\_PER\_YEAR})} - 1)$$

With

$$safeRate = \ln(1 + (e^{minterRate * \frac{\Delta t}{SECONDS\_PER\_YEAR}} - 1) * \frac{totalActiveOwed}{totalEarningSupply}) * \frac{SECONDS\_PER\_YEAR}{\Delta t}$$

Then

$$earnerRate = \min(baseRate, 90\% * safeRate)$$

As the time Δt that will pass until the next update is unknown they calculate the safe rate for an `confidenceInterval`. For this timeframe the mathematical equation takes into account the *compound effect* of interest. The initial value for the `confidenceInterval` is set to 30 days.

When $totalActiveOwed \leq totalEarningSupply$, the `confidenceInterval` is set to 1 second. In this case, the safe earner interest rate will be smaller than the minter interest rate. For less than Δt seconds passed, a `confidenceInterval` of 30 days would overestimate the compounding effect of the interest paid by minters and the resulting interest rate for earners would be too high. A `confidenceInterval` of 1 second is a safe choice as it will underestimate the compounding effect.

When more than 30 days have passed the equations become imprecise. When $totalActiveOWed > totalEarningSupply$ the *safe interest rate* will be larger than the interest paid by minters and the compounding effect of the interest paid to earners will be underestimated. Hence the `updateIndex()` function is assumed to be called at least once every 30 days. The case where $totalActiveOWed > totalEarningSupply$ can arise when Minters are deactivated.

The configuration of the model implicitly assumes that the earner rate depends on the minter rate. Note that, any change on the interest rate by governance is applied only after `updateIndex()` is triggered.

3.2 Governance

The role of the governance is to update values in the `Registrar`, from where the protocol and governance will read their parameters. This is done through proposals in the `StandardGovernor`, `EmergencyGovernor`, and `ZeroGovernor`. `PowerToken` holders can vote in the `StandardGovernor` and `EmergencyGovernor`. Voters are rewarded with `ZeroToken` when voting in all proposals in the `StandardGovernor`. `ZeroToken` holders can vote in the `ZeroGovernor` and they have the power to redeploy the `PowerToken`, `StandardGovernor`, and `EmergencyGovernor` as a bundle.

3.2.1 Registrar

The `Registrar` contract is a key-value store for the parameters of the system. Its values can only be changed by the `StandardGovernor` or the `EmergencyGovernor` through proposals. Here is a non-exhaustive list of the stored values:

- list of approved Minters/Validators/Earners
- addresses of the minters and earners rate models
- freeze time for the minters
- collateral update interval

3.2.2 StandardGovernor

This contract manages proposals and voting process by voting with the vote token (`PowerToken`) on those proposals. Anyone can open a proposal provided they pay a `proposalFee` as set by governance in the form of `cashToken`. The proposals can: add/remove addresses from lists and set key-value pairs in the `Registrar`, and set a new `proposalFee`. The vote token delegatees can cast their votes on a proposal between the starting epoch and the ending epoch of the proposal, in this setting $startEpoch == endEpoch$, so delegatees can only cast their vote in a single voting period. An epoch is 15 days and proposals can be voted only during odd-numbered epochs.

The success of the vote is solely determined by $yesVotes > noVotes$ once the voting epoch ended, and no quorum is required. Votes on proposal always open at the start of the next voting epoch, voters have one epoch to vote, and two epochs to execute the proposal. The proposers of accepted proposals see their `proposalFee` refunded upon execution. When the first proposal for a voting epoch is proposed, the target supply of the vote token is set to be inflated at the start of that voting epoch. Proposals cannot be cancelled. If a proposal is defeated or expired, the associated fee can be sent to the `DistributionVault`. If the proposal gets executed successfully, the proposer is refunded with the proposal fee.

The voters participating in all proposals of a voting epoch are rewarded with their share of `ZeroToken`. The delegators receive no `ZeroToken` reward. The maximum amount of `ZeroToken` distributed during an active voting epoch is 5 million units. The voters can either call the contract to cast their votes directly, or they can submit signatures. The contract implements ERC-712 with ERC-1271 support.

3.2.3 PowerToken

The `PowerToken` is an ERC-20 token with 0 decimals and the following extensions: ERC-3009, ERC-712 with ERC-1271 support, ERC-5805. The token is an epoch-based vote token used in the `StandardGovernor` and `EmergencyGovernor`, and is inflationary (10% per active voting epoch in the `StandardGovernor`). Epochs are 15 days long and start at the Merge timestamp, and all odd epochs are potentially voting epochs. `PowerToken` does not allow minting, transfers or delegations during potential voting epochs. A snapshot of the total supply, balances, voting powers and delegatees is taken at the end of each epoch. Inflation happens only during a voting epoch with at least one (Standard) Proposal. Token holders can either vote themselves or delegate their voting power. If their voting power has been used in every proposals of a voting epoch, their balance will grow with inflation, so token holders are incentivized to vote on all the proposals. The token cannot be transfer/delegated/minted/bought during odd epochs (corresponding to voting epochs).

If not all the voting power was used in a voting epoch, there is a discrepancy between the theoretical inflationary total supply and the actual total supply of `PowerToken`. To fill this gap, it is possible to `buy()` `PowerTokens` up to the discrepancy amount. The pricing takes the form of a Dutch auction and is as follows: the 15 days are divided into 100 periods and decreases linearly during each period. The price halves from start to the end of the period, meaning that the slope of the linear price subfunction is divided by two after every period. Mathematically, with the time remaining in the epoch Δt_E, the time remaining in the period Δt_P, and fixed $buyAmount$ and $totalSupply_{currentEpoch-1}$:

$$cost(\Delta t_P, \Delta t_E) = \frac{\Delta t_P * 2^{\frac{\Delta t_E}{secondsPerPeriod}} + (secondsPerPeriod - \Delta t_P) * 2^{\frac{\Delta t_E}{secondsPerPeriod} - 1}}{secondsPerPeriod} * \frac{buyAmount}{totalSupply_{currentEpoch-1}}$$

The price depends on the percentage of tokens auctioned versus the total supply. The `cashToken` paid by traders is sent to `DistributionVault` and later claimed by `ZeroToken` holders. Therefore, `ZeroToken` holders have an advantage in the Dutch auction as described in ZERO holders buy PowerToken with a discount.

When deployed, the `PowerToken` is provided with an arbitrary bootstrap token from which it can read the balances that apply at the bootstrap epoch. Total supply is set to `10_000` and balances are scaled down to sum up to `10_000`. `ZeroGovernor` can redeploy at any time `PowerToken` and respective governor contracts, to either remove governance rights from existing `PowerToken` holders, or just to reset the accounting variables which continuously grow due to the inflation. When redeploying, the bootstrap token can either be the existing `ZeroToken`, or the previous `PowerToken`. Note that after redeployment, the old `PowerToken` loses its value in the system.

3.2.4 ZeroGovernor

This contract manages proposals and voting process with the vote token (`ZeroToken`) on those proposals. Anyone can open a proposal, at no cost. The proposal can: redeploy the standard and emergency governance systems with the current `PowerToken` or `ZeroToken` as bootstrap token for the new `PowerToken`, set a new `cashToken`, update the threshold ratio for proposals to be accepted in the `ZeroGovernor` or `EmergencyGovernor`. The vote token delegatees can cast their votes on a proposal between the starting epoch and the ending epoch of the proposal, in this setting $endEpoch = startEpoch + 1$. The success of the vote is determined by $\frac{yesVote}{totalSupply} \geq quorum$. Votes on proposal always open in the epoch they are proposed, voters have the remaining time in the current epoch, plus one epoch to vote and execute the proposal, i.e, proposals are executable as soon as the quorum is reached and must be executed until the end of `endEpoch`. Proposals cannot be cancelled.

3.2.5 ZeroToken

The `ZeroToken` is an ERC-20 token with 6 decimals and the following extensions: ERC-3009, ERC-712 with ERC-1271 support, ERC-5805. The token is an epoch based vote token used in the `ZeroGovernor`. Epochs are 15 days long and start at the Merge timestamp. A snapshot of the total supply, balances, voting powers and delegatees is taken at the end of each epoch. Token holders can either vote themselves or delegate their voting power. `ZeroTokens` can only be minted by the `StandardGovernor` as a reward for participation on voting.

3.2.6 EmergencyGovernor

This contract behaves exactly like `ZeroGovernor`, but the set of possible proposals and the vote token are different. The proposals can: add/remove addresses from lists and set key-value pairs in the `Registrar`, and set a new `proposalFee` in the `StandardGovernor`. The vote token is the `PowerToken`.

3.2.7 DistributionVault

The `DistributionVault` receives the excess `MToken` generated by Minters that was not distributed to earners, as well as the `cashTokens` amounts paid upon buying `PowerTokens` or un-refunded `proposalFee` sent to the vault. Users can call `distribute` on the vault in order to account for additional tokens. The difference with the last recorded balance is stored and a snapshot of the token balance is taken for the epoch where `distribute` is called. `ZeroToken` holders can claim their share of token for an epoch, based on their `ZERO` balance at the time of the snapshot.

3.3 Trust Model

- Validators: fully trusted. Validators are responsible to verify the off-chain collateral and produce only valid signatures. They should actively monitor on-chain updates and cancel incorrect minting proposals, freeze or deactivate misbehaving minters. Validators also should be always reachable and issue valid signatures for minters.
- Minters: fully trusted. Minter's have their collateral off-chain and the smart contracts assume that they will always pay their debt to the system. Minters are expected to update their collateral at every interval and pay any imposed penalty.
- `MToken` holders: not trusted. Anyone can hold `M` tokens and use them as stablecoin.
- `PowerToken` and `ZeroToken` holders: trusted to always act in the best interest of the protocol. We assume they have the incentives to always execute successful proposals after the vote, and call public functions that apply new parameters such as `updateIndex()`.
- The `cashTokens` are expected to be `WETH` and `M` as stated in the Whitepaper, but nothing prevents M^ZERO Labs to deploy the system with other `cashTokens`. We assume third-party tokens, if used as `cashToken`, are fully trusted and they are ERC20-compliant without special behavior (e.g., having transfer hooks, charging fees on transfer, rebasing or inflationary/deflationary tokens).

3.4 Changes in Version 2

- The functionality allowing to start/stop earning on behalf of an address in the `MToken` has been removed.
- The functions `ZeroToken.getPastVotes(address,uint256,uint256)` and `ZeroToken.pastDelegates(address,uint256,uint256)` have been removed.
- The `SplitEarnerRateModel` have been removed. The `StableEarnerRateModel` now calculates the earner interest rate with the formula of the `SplitEarnerRateModel` when

$totalActiveOwed \leq totalEarningSupply$ (previously the `confidenceInterval` was set to 1 second):

$$earnerRate = \min(baseRate, 90\% * minterRate * \frac{totalActiveOwed}{totalEarningSupply})$$

The safety threshold of 90% has been removed from the stable earner model. If the interest rate of earners is larger than the interest rate of minters, the interest paid to earners will start to outgrow the interest paid by minters after 30 days. The interest rate index of the minters is now rounded up and the interest rate index of the earners is rounded down.

3.5 Changes in Version 3

- The penalty for missing intervals has been revised to charge a higher penalty to minters. Previously, failing to call `updateCollateral()` for 1.5 intervals would incur a penalty as having zero collateral for 1 interval, while in **Version 3** minters pay a penalty for missing collateral update for 1 interval first, and then a penatly for 0.5 interval on the updated amount for beeing undercollateralized.

4 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	3

- Side-effects of Resets Acknowledged
- Effects of Roundings in PowerToken Code Partially Corrected
- Standard Proposal Fee Has Ambiguous Denomination Risk Accepted

Low-Severity Findings	7

- Sync of Address 0 During Delegation Acknowledged
- Vote Weight Has Ambiguous Interpretation Acknowledged
- Contract ERC3009 Inherits StatefulERC712 Acknowledged
- Earner Interest Can Exceed Minter's Interest Acknowledged
- Excess Owed M Can Be Larger Due to Rounding Acknowledged
- Incomplete Interfaces Code Partially Corrected
- Remaining Dust in Distribution Vault Acknowledged

6.1 Side-effects of Resets

Correctness Medium Version 2 Acknowledged

CS-MZEROCORE-019

Zero holders can redeploy a new POWER token through the ZeroGovernor. As ZeroGovernor is based on a threshold voting, the proposal is executed as soon as it receives the required votes. If reset is triggered in epoch `N`, the new POWER token records as bootstrap epoch `N-1`, hence all accounts are bootstrapped according to their last snapshot in epoch `N-1`, ignoring any activity on old POWER token during epoch `N`. A reset has the following has side-effects depending on the epoch `N` when it happens:

- `N` is even (transfer epoch): Bootstrap balances in the new Power token will be based on snapshots from epoch `N-1`. Therefore, any transfer before redeployment during epoch `N` in the old token is ignored. Similarly, POWER tokens bought during the auction are lost even though users paid for them in the cash token. Moreover, any potential inflation from the previous voting epoch is lost although the total supply had been inflated.
- `N` is odd (voting epoch): Users voting in all standard proposals lose their inflation.

Furthermore, reset events affect the historical values for epochs before the bootstrap. Consider the following scenario:

- Governance is deployed at epoch `N`.
- A reset event happens at epoch `N+20`. Old POWER is used as bootstrap token.
- Querying historical balance of an account at epoch `N+1` depends on the old POWER token.
- A reset event happens at epoch `N+30`. ZERO is used as bootstrap token this time.
- Querying again the balance of an account at epoch `N+1` returns a different balance as now it depends at ZERO token.

Acknowledged:

M^ZERO Labs is aware of this behavior and consider it to be inline with their design decisions.

6.2 Effects of Roundings in `PowerToken`

Correctness Medium Version 1 Code Partially Corrected

CS-MZEROCORE-004

Some rounding errors in the `PowerToken` can happen in two cases:

1. Accounts with less than 0.01% of the `totalSupply` in the `bootstrapToken` get rounded down to zero when bootstrapped. The rounding error potentially happens also for other accounts even if they do not round down to zero. Therefore the sum of all bootstrap amounts might not match the initial supply.
2. Holders of `PowerToken` with less than `10 PowerTokens` do not get any balance inflation although they (or their delegatee) vote in all proposals. This could happen due to the rounding down in function `_getInflation()`. In case the holder delegates to another account which has a voting power more than 10, its voting power inflates, but the balance of the delegator will not inflate.

Note that the difference between the total supply and the actual sum of all account balances grows over time, and might have implications in the quorum-based governance if threshold is high (quorum might be unreachable).

Code partially corrected:

1. The total supply has been increased from `10_000` to `1_000_000`, decreasing the effect of rounding by a factor 100 as well.
2. This issue still holds.

6.3 Standard Proposal Fee Has Ambiguous Denomination

Correctness Medium Version 1 Risk Accepted

CS-MZEROCORE-005

The Standard Proposal Fee can be changed in two ways:

- ZeroGovernance: `setCashToken(newCashToken_, newProposalFee_)`

- Standard or Emergency governances: `setProposalFee()`

The second option changes the proposal fee and keeps the current cash token in place. Voters (PowerToken holders) will vote on whether the proposal makes sense in consideration of the current cash token. Yet, changes to the cash token (voted by ZeroToken holders) before execution of the proposal can drastically skew the proposal's intent. For instance, a fee of `1000 * 1e6` is reasonable with M (1000 USD) but could be very cheap if the cash token is switched to WETH by zero governance. This would enable griefing attacks in the standard governor as making new proposals has negligible cost, while rejecting them is costly in terms of gas (majority should vote `no`). Similarly, a fee of `2e18` is reasonable with WETH (2 Ether) but could become exorbitantly high if the cash token is MToken (2e14 USD or 200 trillion USD).

Risk accepted:

M^ZERO Labs is aware of this issue but has decided to keep the code unchanged, providing the following reasoning:

```
Zero holder can, with the same threshold of yes votes, nuke current Power holders and replace
them with themselves, so there is no real reason for them to “play with their food” by abusing
setCashToken(newCashToken_, newProposalFee_) to make Power holders’ lives difficult. Nevertheless,
errors can occur, and new emergency proposals or Zero proposals can be proposed passed,
and executed quickly to set the proposal fee to something sensible.
```

6.4 Sync of Address 0 During Delegation

Design | Low | Version 2 | Acknowledged

CS-MZEROCORE-033

Delegating to `address(0)`, which is used as an alias for delegating to self, triggers a sync on `address(0)`, hence pushing snapshots in storage for the bootstrap and sync:

```
function _delegate(address delegator_, address newDelegatee_) internal override {
    if (delegator_ != newDelegatee_) _sync(newDelegatee_, _clock());
    ...
}
```

Acknowledged:

M^ZERO Labs is aware of this behavior and consider it to be inline with their design decisions.

6.5 Vote Weight Has Ambiguous Interpretation

Correctness | Low | Version 2 | Acknowledged

CS-MZEROCORE-034

BatchGovernor._castVotes returns the value `weight_`, which is defined as the number of votes cast for each proposal. However, if multiple proposals are voted on, the function returns the number of votes cast for the *last proposal*.

Note that a voter has the same number of votes on all proposals started in the same epoch. However, Emergency and Zero Governors proposals can be voted on in two epochs. If a user votes on proposals created in two different epochs, the function returns the number of votes the user had in one of the epochs, depending on the order of the proposals, and not the votes cast. This is a problem for external integrations that could misrepresent the voting weight.

Acknowledged:

Client is aware of this behavior, but has decided to keep the code unchanged.

6.6 Contract ERC3009 Inherits StatefulERC712

Design | Low | Version 1 | Acknowledged

CS-MZEROCORE-006

The contract `ERC3009` extends the abstract contract `StatefulERC712` which keeps track of used nonces in the public mapping `nonces`. However, `ERC3009` does not use any functionality of this contract.

Furthermore, `ERC3009` uses random nonces of type `bytes32` and the standard explicitly avoids sequential nonces. On contrary, `StatefulERC712` is designed to use sequential nonces. Hence, extending `ERC712` is enough.

Acknowledged:

M^ZERO Labs has acknowledged the issue but has decided to keep the code unchanged.

6.7 Earner Interest Can Exceed Minter's Interest

Design | Low | Version 1 | Acknowledged

CS-MZEROCORE-003

In the `SplitEarnerRateModel` the interest rate of earners is computed based on the minter's interest and the ratio of total supply of active minters and the total supply of earners:

```
UIntMath.min256(
    baseRate(),
    (_EARNER_SPLIT_MULTIPLIER * (IMinterGateway(minterGateway).minterRate() * totalActiveOwedM_)) /
        totalEarningSupply_ /
        _ONE
);
```

The multiplier is set to a constant(90%). The formula computes a higher interest rate for earners than minters if `totalActiveM > totalEarningSupply_`. Due to the continuous compounding effect, the cashflow becomes negative over time, i.e., earners receive more MTokens than paid by minters. Therefore, `baseRate()` has to be chosen carefully to prevent this scenario from happening in practice.

A similar behavior manifests in the `StableEarnerRateModel` if minter's supply of M tokens is larger than earners' supply and `updateIndex()` does not get called for longer than 30 days (confidence interval). The interest of earners exceeds the interest paid by minters due to compounding, hence netting a negative cashflow for the system.

Acknowledged:

Contract `SplitEarnerRateModel` has been removed from codebase (Version 2). However, the issue is still present in the revised `StableEarnerRateModel` if the following conditions hold:

- The supply of owed M in minterGateway is larger than the total supply of earners.
- `MToken.updateIndex()` is not triggered for more than 30 days.

Note that `MToken.updateIndex()` is always triggered when a minter updates their collateral. Minters are incentivized to update their collateral with a frequency enforced by the governance. However, the governance can change the frequency in the future to any duration and raise the issue if the two conditions above are met.

In Version 3, the earners' interest rate is is set to 90% of the value computed in function `getSafeEarnerRate()`, hence reducing the likelihood of M overprinting. However, if `MToken.updateIndex()` is not triggered for a long time (+30 days), overprinting can happen.

6.8 Excess Owed M Can Be Larger Due to Rounding

Design Low Version 1 Acknowledged

CS-MZEROCORE-008

Function `MinterGateway.updateIndex()` mints the difference between total owed M and M token total supply to the TTG vault. The difference is computed in function `excessOwedM()` which queries the total supply from MToken:

```
uint240 totalMSupply_ = uint240(IMToken(mToken).totalSupply());

uint240 totalOwedM_ = _getPresentAmountRoundedDown(principalOfTotalActiveOwedM, currentIndex()) +
    totalInactiveOwedM;

unchecked {
    if (totalOwedM_ > totalMSupply_) return totalOwedM_ - totalMSupply_;
}
```

Function `MToken.totalSupply()` rounds down the total supply of earners, therefore the excess M amount is computed slightly larger than the real value. In this case, the gateway will mint more tokens to the vault.

Acknowledged:

The M^ZERO Labs team is aware of this behavior and has provided the following description:

```
This is fine so long as the invariant that MToken.totalSupply() <= MinterGateway.totalOwedM() is
maintained. Further, this excess is limited to the max rounding error (so it's always just a
rounding error) and in any case, the TTG vault results in larger dust as it tries to divide the
minted M tokens across all Zero holders.
```

6.9 Incomplete Interfaces

Design Low Version 1 Code Partially Corrected

CS-MZEROCORE-010

1. The contract `MinterGateway` is `ContinuousIndexing` and `ERC712`, but `IMinterGateway` only extends `IContinuousIndexing`. For completeness, `IMinterGateway` should also inherit `IERC712`.

2. The interface IERC3009 should declare the functions TRANSFER_WITH_AUTHORIZATION_TYPEHASH() and RECEIVE_WITH_AUTHORIZATION_TYPEHASH() to match the ERC-3009 interface standard.

Code partially corrected:

1. IMinterGateway has been updated to inherit IERC712, but in Version 2 the function eip712Domain is added in IERC712Extended.
2. IERC3009 now declares the functions listed above.

6.10 Remaining Dust in Distribution Vault

Correctness Low Version 1 Acknowledged

CS-MZEROCORE-029

Function DistributionVault.getClaimable() rounds down when computing the amount of cash token that can be claimed by an account, hence dust remains in the vault:

```
claimable_ += (distributionOfAt[token_][startEpoch_ + index_] * balance_) / totalSupply_;
```

The dust of cash tokens (including MToken) accumulates in the vault and cannot be withdrawn. In case of MToken, the locked dust has implications for last minters, who might be unable to fully repay their debt and close their positions.

Acknowledged:

M^ZERO Labs answered:

```
There is no real way to prevent this from happening, just as there is no real way to prevent M
from being sent to addresses to which no one has the private key (or control of), which would
also result in last minters not being able to fully repay. This means that M tokens are worth
more than the nominal debt (since M tokens will be lost). Ideally this encourages minting. A
full wind down would likely involve social layers, in which remaining Minters can be allowed or
aided in exiting. After all, their collateral is off-chain.
```

However, a mechanism to reduce the dust amount has been implemented, by adding 9 decimals of precision in the computation.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	3

- Hashing of String Arrays Is Not Compliant With EIP-712 Code Corrected
- Code Restricts Execution of Proposal to 1 Epoch Code Corrected
- EIP-712 Dynamic Types Code Corrected

Low-Severity Findings	15

- Outdated Dependencies Code Corrected
- UI Interpretation of COUNTING_MODE Code Corrected
- EIP5805 DelegateChanged Not Always Emitted Code Corrected
- Inconsistent Collateral and Penalty at Expiry Boundary Code Corrected
- Incorrect Specifications Code Corrected
- Missing Input Sanitization Code Corrected
- No Expiry in Buy Function Code Corrected
- Possible Overflow When Syncing Accounts Code Corrected
- Possible Overflow in convertToBasisPoints Code Corrected
- Possible Rounding to 0 in getSafeEarnerRate Specification Changed
- Reentrancy in PowerToken Re-Buy Code Corrected
- Remaining ToDos in Codebase Code Corrected
- Resubmission of Signatures and Staleness Code Corrected
- Timestamp 0 in Signatures Code Corrected
- Wrong Condition in StableEarnerRateModel Code Corrected

Informational Findings	4

- Dead Code Code Corrected
- Misleading Error Name Code Corrected
- Reason Ignored in BatchGovernor Code Corrected
- ERC712 Does Not Implement Extension EIP-5267 Code Corrected

7.1 Hashing of String Arrays Is Not Compliant With EIP-712

Correctness Medium Version 2 Code Corrected

CS-MZEROCORE-040

The standard EIP712 specifies that dynamic types (i.e. arrays) must be encoded as the `keccak256` hash of their contents. The function `_getReasonListHash()` does not follow the standard when hashing `reasonList` which is an array of `string` (dynamic type):

```
function _getReasonListHash(string[] calldata reasonList_) ... {
    bytes memory reasonBytes_;

    for (...) {
        reasonBytes_ = abi.encodePacked(reasonBytes_, bytes(reasonList_[index_]));
    }

    return keccak256(reasonBytes_);
}
```

Furthermore, using `encodePacked` for an array of strings enables hash collisions as it is possible to craft different strings that produce the same hash.

Code corrected:

The function has been updated to comply to EIP712.

7.2 Code Restricts Execution of Proposal to 1 Epoch

Correctness Medium Version 1 Code Corrected

CS-MZEROCORE-001

The function `StandardGovernor.execute()` tries to execute all proposals voted in the last two epochs. However, the function `StandardGovernor.state()` returns the status `Succeeded` only for proposals voted in the previous epoch. The state `Expired` is returned for older proposals, hence stopping them from being executed. This behavior conflicts the whitepaper and the code comments in function `execute()` which state that a successful proposal can be executed during the next 2 epochs:

```
// Proposals have voteStart=N and voteEnd=N, and can be executed only during epochs N+1 and N+2.
```

Code corrected:

The function `StandardGovernor.execute` has been updated to execute only the proposals voted in the last epoch, and the comment was update:

```
// Proposals have voteStart=N and voteEnd=N, and can be executed only during epoch N+1.
```

The condition for `ProposalState.Active` in `StandardGovernor.state` has been made stricter (from <= to ==), as `voteStart == voteEnd`.

7.3 EIP-712 Dynamic Types

Correctness Medium Version 1 Code Corrected

CS-MZEROCORE-002

The standard EIP712 specifies that dynamic types like arrays must be encoded as the `keccak256` hash of the concatenated `encodeData` of their contents. The following functions do not hash the arrays according to the standard:

- `BatchGovernor.getBallotsDigest()`
- `MinterGateway._getUpdateCollateralDigest()`

Code corrected:

The functions listed above now hash the arrays according to the standard.

7.4 Outdated Dependencies

Design Low Version 3 Code Corrected

CS-MZEROCORE-039

The `common` library referenced in the `TTG` is an old version of the codebase and should be updated.

Code corrected:

The commit reference to `common` library has been updated in Version 4.

7.5 UI Interpretation of `COUNTING_MODE`

Correctness Low Version 2 Code Corrected

CS-MZEROCORE-038

In the `IBatchGovernor`, the `enum VoteType` binds a "no" to `0` and a "yes" to `1`. Abstain votes are not supported in governance contracts. This behavior should be reflected in the view function `COUNTING_MODE()` which returns `support=for,against`. However, the `enum VoteType` and the value returned by `COUNTING_MODE` are not aligned.

Code corrected:

The function `COUNTING_MODE()` was updated to return `support=against,for`.

7.6 EIP5805 `DelegateChanged` Not Always Emitted

Correctness Low Version 1 Code Corrected

CS-MZEROCORE-007

The EIP-5805 specs requests the `DelegateChanged` event to be emitted when `delegator` changes the delegation of its assets from `fromDelegate` to `toDelegate`. The function

`EpochBasedVoteToken._setDelegatee` does not fully adhere to the standard as only emits the event when it is not the first change of delegation. I.e., if `_delegatees[delegator_].length==0` the function starts a snapshot for the account with the new delegatee and returns without emitting the event.

Code corrected:

The function `EpochBasedVoteToken._setDelegatee` has been updated to emit the `DelegateChanged` event in the special case where `_delegatees[delegator_].length==0`.

7.7 Inconsistent Collateral and Penalty at Expiry Boundary

Correctness Low Version 1 Code Corrected

CS-MZEROCORE-011

The penalty and collateral calculation are not consistent with each other when `block.timestamp == updateTimestamp + updateCollateralInterval`. The collateral will still be the non-zeroed collateral value, but a penalty for missed collateral update will still be charged for 1 period.

Consider the following example:

- `interval = 3`
- minter updated its collateral at `updateTimestamp = 1` with `collateral = 42`
- we are now at `block.timestamp == 4`:
  - `collateralOf` will return `42` since `block.timestamp == updateTimestamp + interval`
  - a penalty will be charged for 1 period because `missedIntervals_ = (block.timestamp - lastUpdate) /  = 1`

Code corrected:

The collateral value is now zero when `block.timestamp == updateTimestamp + updateCollateralInterval`.

7.8 Incorrect Specifications

Correctness Low Version 1 Code Corrected

CS-MZEROCORE-028

1. The natspec description of `IRateModel.rate` states that the return value is `APY in BPS`. However, `rate()` returns the yearly interest rate does not consider the compounding.
2. The natspec description for `principalAmount` in event `PenaltyImposed` is incorrect.
3. The natspec `@return weight_` of `BatchGovernor._castVote()` indicates `The type of support to cast for each proposal`, but it should be the voting power of the voter.

Code corrected:

The incorrect natspec descriptions listed above have been fixed in Version 2.

7.9 Missing Input Sanitization

Design | Low | Version 1 | Code Corrected

CS-MZEROCORE-012

- Some of the functions accept an `epoch=0` as input, which is an invalid input as $epoch > 0$. Non-exhaustive list of such functions:
  - `ZeroToken`: `getPastVotes`, `pastBalancesOf`, `pastDelegates`, `_getDelegateesBetween`, `_getValuesBetween`
  - `EpochBasedVoteToken`: `pastBalanceOf`, `pastDelegates`, `getPastVotes`, `pastTotalSupply`
- Function `MinterGateway.proposeMint()` does not perform any sanity check on `destination` address.
- Function `BatchGovernor.castVotes()` does not check that the length of input arrays matches.

Code corrected:

Various inputs sanitization have been added across the codebase, fixing the issue.

7.10 No Expiry in Buy Function

Security | Low | Version 1 | Code Corrected

CS-MZEROCORE-013

The function `PowerToken.buy()` does not allow users to specify an expiry timestamp, which would prevent a transaction to execute at a later time. Currently, it is possible that user's transaction gets executed at a future transfer epoch and potentially buys tokens with a price higher than originally intended.

Code corrected:

The `buy()` function has been modified to accept an additional argument `expiryEpoch_` and the transaction reverts when the current epoch is greater than the expiry epoch.

7.11 Possible Overflow When Syncing Accounts

Correctness | Low | Version 1 | Code Corrected

CS-MZEROCORE-014

The function `_sync()` in `EpochBasedInflationaryVoteToken` computes the unrealized inflation of an account by iterating through all epochs since last sync. The `for-loop` is implemented in `_getUnrealizedInflation()` and in each iteration, except the last one, it checks that the new balance does not exceed the limits:

```
// Cap inflation to `type(uint240).max`.
if (inflatedBalance_ >= type(uint240).max) return type(uint240).max;
```

However, if the inflation from the last iteration causes the final balance of an account to exceed the limit (`type(uint240).max`), function `_sync()` updates the balance with the full inflation amount via `_addBalance()`. The later uses `unchecked` block, hence an overflow happens.

Code corrected:

The function `_getUnrealizedInflation` has been revised to ensure that the resulting balance of an account does not overflow. If the inflated balance exceeds the maximum value of `uint240` to return `type(uint240).max - balance_`:

```
for (...) {
    ...

    unchecked {
        inflatedBalance_ += _getInflation(uint240(inflatedBalance_));

        // Cap inflation to `type(uint240).max`.
        if (inflatedBalance_ >= type(uint240).max) return type(uint240).max - balance_;
    }
}
```

7.12 Possible Overflow in convertToBasisPoints

Design | Low | Version 1 | Code Corrected

CS-MZEROCORE-015

The function `ContinuousIndexingMath.convertToBasisPoints()` uses `unchecked` block to convert a `uint64` input into a `uint32` type. The computation can overflow for large values of `input`, i.e., `input > type(uint32).max * 10**8`.

This issue is unlikely to happen in the current codebase as the function is called only with inputs representing interest rates which are capped.

Code corrected:

The output has been changed to `uint40`, which can hold the result of the conversion up to the maximum value of the input.

7.13 Possible Rounding to 0 in getSafeEarnerRate

Design | Low | Version 1 | Specification Changed

CS-MZEROCORE-016

The function `getSafeEarnerRate` in `StableEarnerRateModel` computes the value `lnArg_` as follows:

```
int256 lnArg_ = int256(
    1e12 + ((((uint256(totalActiveOwedM_) * (deltaMinterIndex_ - 1e12)) / 1e12) * 1e12) / totalEarningSupply_)
);
```

Note that `deltaMinterIndex_` is usually close to 1 (`10**12`) for short time intervals, hence `deltaMinterIndex_ - 1e12` is a value close to 0. Therefore the intermediary result `(uint256(totalActiveOwedM_) * (deltaMinterIndex_ - 1e12)) / 1e12` rounds down to 0 for values of `totalActiveOwedM` below a certain threshold.

Specification changed:

The specifications of function `getSafeEarnerRate()` have changed in Version 2, hence a confidence interval of 1 second is not used anymore. Furthermore, `deltaMinterIndex_` is no longer rounded down by performing a division with `1e12`:

```
RATE_CONFIDENCE_INTERVAL = 30 days

...

uint48 deltaMinterIndex_ = ContinuousIndexingMath.getContinuousIndex(
    ContinuousIndexingMath.convertFromBasisPoints(minterRate_),
    RATE_CONFIDENCE_INTERVAL
);

int256 lnArg_ = int256(
    1e12 + (((uint256(totalActiveOwedM_) * (deltaMinterIndex_ - 1e12))) / totalEarningSupply_)
);
```

7.14 Reentrancy in PowerToken Re-Buy

Security | Low | Version 1 | Code Corrected

CS-MZEROCORE-017

In the function `PowerToken.buy()` the `cashToken` in transferred from the buyer before the `totalSupply` of the token is increased by `mint()`. If the `cashToken` implements callbacks (ERC777-like), this enables a reentrancy issue that allows an attacker to mint arbitrary amounts of `PowerToken`, as the `amountToAuction` would only be decreased after `mint()` is called.

Code corrected:

Function `buy()` has been revised to perform the transfer of the cash token at the end of the function, following check-effect-interaction (CEI) pattern.

7.15 Remaining ToDos in Codebase

Design | Low | Version 1 | Code Corrected

CS-MZEROCORE-018

The following `ToDo` comments are present in the following contracts:

- `BatchGovernor`
- `ThresholdGovernor`

Addressing remaining notes help improve the quality and readability of the code.

Code corrected:

ToDo comments listed above have been removed from the codebase.

7.16 Resubmission of Signatures and Staleness

Design | Low | Version 1 | Code Corrected

CS-MZEROCORE-041

The function `_updateCollateral()` reverts only if the new timestamp is strictly smaller than the current one. But if the exact same batch of signatures is used, the two timestamps will be equal and the check will pass, even though the result is stale.

Code corrected:

The function `_updateCollateral()` has been updated to revert also if the new timestamp equals the one already stored:

```
 uint40 lastUpdateTimestamp_ = _minterStates[minter_].updateTimestamp;

// MinterGateway already has more recent collateral update
if (newTimestamp_ <= lastUpdateTimestamp_) revert StaleCollateralUpdate(newTimestamp_, lastUpdateTimestamp_);
```

7.17 Timestamp 0 in Signatures

Correctness | Low | Version 1 | Code Corrected

CS-MZEROCORE-030

Function `MinterGateway.verifyValidatorSignatures()` currently allows signatures with a timestamp set to `0`. Although it is anticipated that validators will not typically generate signatures with a timestamp of `0`, in the event that such signatures occur, there is a risk of replaying them, given that `minTimestamp` will always be `block.timestamp`.

Code corrected:

Function `_verifyValidatorSignatures()` has been updated to revert with the error `ZeroTimestamp` if a signature has a timestamp of zero:

```
// Check that the timestamp is not 0.
if (timestamps_[index_] == 0) revert ZeroTimestamp();

// Check that the timestamp is not in the future.
if (timestamps_[index_] > uint40(block.timestamp)) revert FutureTimestamp();
```

Note, whitelisted validators are considered fully trusted by the system and they should only provide signatures with timestamps that match the off-chain verification of the collateral.

7.18 Wrong Condition in `StableEarnerRateModel`

Correctness | Low | Version 1 | Code Corrected

The function `StableEarnerRateModel.getSafeEarnerRate()` implements the check `expRate_ > type(uint64).max` to return early if the rate is too big, otherwise the value returned by `ContinuousIndexingMath.convertToBasisPoints(uint64(expRate_))` will be returned.

As mentioned in the issue Possible Overflow in ConvertToBasisPoints, the function `ContinuousIndexingMath.convertToBasisPoints` will overflow if its input is greater than `type(uint32).max`. The currently implemented check leaves a hole for the values of the rate between `type(uint32).max` and `type(uint64).max` where the function will overflow. The function `StableEarnerRateModel.getSafeEarnerRate()` should return early if `expRate_ > type(uint32).max` instead.

Code corrected:

The issue Possible Overflow in ConvertToBasisPoints has been fixed and the function `StableEarnerRateModel.getSafeEarnerRate()` returns the maximum value of `uint32` when the expRate is greater than `type(uint32).max`:

```
function getSafeEarnerRate(...) public pure returns (uint32) {
    ...

    uint40 safeRate_ = ContinuousIndexingMath.convertToBasisPoints(uint64(expRate_));
    return (safeRate_ > type(uint32).max) ? type(uint32).max : uint32(safeRate_);
}
```

7.19 Dead Code

Informational **Version 1** **Code Corrected**

CS-MZEROCORE-021

The libraries `PureEpochs` and `ContinuousIndexingMath` implement some unused functions. The unused functions will be ignored by the compiler, but unused code can increase the difficulty of understanding the codebase. The functions are:

- `ContinuousIndexingMath.exponentAssembly`
- `PureEpochs.getTimeUntilEpochStart`
- `PureEpochs.getTimeUntilEpochEnds`
- `PureEpochs.getTimeSinceEpochStart`
- `PureEpochs.getTimeSinceEpochEnd`
- `SignatureChecker.isValidECDSASignature(address,bytes32,uint8,bytes32,bytes32)`

The function `EpochBasedVoteToken._subUnchecked` is never used.

Code corrected:

All the functions listed above have been removed from the codebase.

7.20 ERC712 Does Not Implement Extension EIP-5267

Informational | Version 1 | Code Corrected

CS-MZEROCORE-022

The abstract contract ERC712 does not implement the extension EIP-5267 which aims to improve the integration of EIP-712 signatures with third-party tools.

Code corrected:

M^ZERO Labs team has implemented the *EIP-5267* extension in the abstract contract *ERC712Extended*.

7.21 Misleading Error Name

Informational | Version 1 | Code Corrected

CS-MZEROCORE-025

The error `ReusedNonce` emitted in `ERC5805._checkAndIncrementNonce()` is misleading, as this error will be emitted for nonce that are $> currentNonce$, which haven't been used yet by definition.

Code corrected:

The error has been renamed as `InvalidAccountNonce`.

7.22 Reason Ignored in BatchGovernor

Informational | Version 1 | Code Corrected

CS-MZEROCORE-027

The function `BatchGovernor.castVoteWithReason()` takes as input a `string` parameter that represents the reason. This parameter is ignored by the function and the event `VoteCast` is always emitted with an empty string as reason.

Code corrected:

The function `castVoteWithReason()` has been revised to include the `reason` variable in the event `VoteCast`. Furthermore, new functionalities that allow users to provide reasons when submitting votes for a list of proposals, or when voting with a signature, are implemented.

8 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

8.1 Gas Optimizations

Informational | Version 1 | Code Partially Corrected

CS-MZEROCORE-023

1. In the function `MinterGateway._verifyValidatorSignatures()`, the check for approved validators can be moved higher up in the loop to save gas in the case the validator is not approved.
2. In the functions `PowerToken._getBalance()` and `PowerToken._getVotes()`, `_getUnrealizedInflation()` is called if the length of `_balances[account_]` or `_votingPowers[account_]` is 0. This is not needed as `_getUnrealizedInflation()` will always return `0` in that case. I.e., as delegation are reset with the bootstrap, the only way for `_getUnrealizedInflation()` to be non-zero is for the account to have its voting power used after `bootstrapEpoch`. This can be done only if the account either delegated or voted after the redeployment, both of those actions trigger `bootstrap()`, which updates `_balances[account_]` and `_votingPowers[account_]`.
3. The decrement of `latestVoteStart_` can be unchecked in `BatchGovernor._tryExecute()`.
4. In the `ThresholdGovernor`, since `_votingDelay = 0` the state `Pending` cannot be reached and the check `currentEpoch_ < voteStart_` in `ThresholdGovernor.state()` can be dropped.
5. The function `SignatureChecker.isValidSignature()` will try to ECDSA-verify the signature even if the signature is following the EIP-1271. Checking the codesize of the target to decide whether to ECDSA-verify or to call `isValidSignature()` will save gas.
6. The check `error_ == SignatureChecker.Error.InvalidSignature` in `ERC712._revertIfError()` can be avoided as the function throws the same error as default.
7. Local variables `error` in functions `SignatureChecker.validateECDSASignature(*)` are not used.
8. The first condition `expiry_ != type(uint256).max` in `ERC712._revertIfExpired()` is redundant as such timestamp cannot be expired due to `block.timestamp` being smaller than `type(uin256).max`.
9. Function `ERC3009.transferWithAuthorization()` could be more gas efficient to revert early if `validBefore_` and `validAfter_` are checked first.
10. Function `ERC3009.receiveWithAuthorization()` could be more gas efficient to revert early if `validBefore_`, `validAfter_` and `to_ == msg.sender` are checked first.
11. The internal function `ContinuousIndexing._getPrincipalAmountRoundedUp(uint240, uint128)` could be avoided if `divideUp()` is called by caller.
12. The field `quorumRatio` in the struct `Proposal` is unused.
13. The check `latestPossibleVoteStart_ > 0` in `ThresholdGovernor.execute()` is redundant as `currentEpoch` is already checked to be non-zero.
14. The check `participationInflation_ > ONE` in `EpochBasedInflationaryVoteToken` is redundant as the variable is set to `1000` (10%) when PowerToken is deployed.

15. The sanity checks in `EmergencyGovernor.constructor()` are redundant as they are checked by the deployer.
16. Similarly, several checks in `StandardGovernor.constructor()` are redundant.
17. The sanity check in `PowerToken.setNextCashToken()` is redundant as `nextCashToken_` is validated already in ZeroGovernor.

Version 2:

18. Functions `_subtractEarningAmount()` and `_subtractNonEarningAmount()` in MToken perform redundant SLOADs when updating `rawBalance`.
19. Function `StableEarnerRateModel.rate()` performs a multiplication and a division with the same value.
20. The local variable `digest_` in `ERC20Extended._permitAndGetDigest()` is unused.
21. The local variables `weight_` in variations of function `castVote*BySig()` in BatchGovernor are unused.
22. The checks `delegator_ != newDelegatee_` in `PowerToken._delegate()` and `recipient_ != sender_` in `EpochBasedInflationaryVoteToken._transfer()` can be avoided.
23. Function `_revertIfInvalidCalldata` in `EmergencyGovernor`, `StandardGovernor` and `ZeroGovernor` could be more gas efficient to return early if calldata is matched and revert if no match is found.

Code partially corrected:

The optimization points 1, 3, 4, 7, 8, 13, and 19 have been implemented in the updated codebase. The optimization in point 5 is partially addressed as the function has been revised to use less gas for ECDSA signatures, while making ERC1271 signatures more expensive.

8.2 Inconsistent Error on Transfer Reverts

Informational Version 2

CS-MZEROCORE-035

Transferring M tokens between accounts of different states (earner to non-earner, or vice-versa) throws the error `InsufficientBalance` when sender has not enough balance. However, when transferring M tokens between two accounts with same state reverts due to underflow in the internal function `_transferAmountInKind()`:

```
function _transferAmountInKind(address sender_, address recipient_, uint240 amount_) internal {
    _balances[sender_].rawBalance -= amount_;
    ...
}
```

Similarly, POWER and ZERO token revert due to an underflow when the sender does not have enough balance.

8.3 Inconsistent Events

Informational Version 1 Code Partially Corrected

CS-MZEROCORE-024

1. In the function `MinterGateway.updateCollateral()`, the order of events does not match the order of changes on-chain. For example, the events would be trigger in the following order: update-penalty-penalty, but the order of execution on-chain is penalty-update-penalty. It is in general good practice to emit the events to match the changes on-chain.
2. Anyone can call the function `MinterGateway.activateMinter()` for an existing active minter and emit the respective event, although no state changes.
3. The event `MintCanceled` is emitted if calling the function `MinterGateway.cancelMint()` with `mintId_ = 0` although no such proposal can exist.
4. Functions `startEarning()` and `stopEarning()` can be called multiple times for an address to emit the respective events.
5. Functions `allowEarningOnBehalf()` and `disallowEarningOnBehalf()` can be called multiple times with no state changes.
6. Function `PowerToken.buy()` can be called at any time with `minAmount_` set to 0, so events `Buy` and `Transfer` would be emitted even during voting epochs.

Version 2:

7. The event `Sync` is emitted only by function `EpochBasedInflationaryVoteToken.sync()`, although a sync happens in other call paths. Futhermore, `sync()` can be called with arbitrary `epoch_` (e.g., older than the last sync's epoch) and the event is always emitted.

Code partially corrected:

1. No change.
2. No change.
3. The function now reverts if `mintId_ == 0`.
4. Both functions `startEarning()` and `stopEarning()` have been updated to return early and do not emit an event when they do not trigger a state change.
5. The functions have been removed from the codebase.
6. The function now reverts if `minAmount_ == 0` or `maxAmount_ == 0`.
7. M^ZERO Labs informed us they only want the `Sync` event to be emitted from the public `sync()` function. The function has been updated and now only takes an address as parameter.

8.4 Metadata of PowerToken

Informational Version 1 Acknowledged

CS-MZEROCORE-031

The name and symbol of `PowerToken` is hardcoded in its constructor:

```
constructor(
    ...
) EpochBasedInflationaryVoteToken("Power Token", "POWER", 0, ONE / 10) {
    ...
}
```

Therefore, name and symbol will be the same for new tokens if redeployed by governance.

Acknowledged:

M^ZERO Labs answered:

```
There can only be one Power Token at a time.
```

8.5 Misleading Natspec Description for _divideUp

Informational Version 1

CS-MZEROCORE-032

The natspec description of `PowerToken._divideUp()` is misleading as the function actually rounds up the ratio `x/y` in BPS. Therefore, the function should not be used with arbitrary inputs as the description might suggest:

```
/**
 * @dev Helper function to calculate `x` / `y`, rounded up.
 ...
 */
function _divideUp(uint256 x, uint256 y) internal pure returns (uint256 z) {
    ...
}
```

8.6 Past Balances From Bootstrap Token Are Scaled Down

Informational Version 2

CS-MZEROCORE-036

The internal function `_getInternalOrBootstrap` relies on `_getBootstrapBalance()` to compute past balances or votes for an account according to an epoch which is older than the bootstrap epoch:

```
if (epoch_ <= bootstrapEpoch) return _getBootstrapBalance(account_, epoch_);
```

Function `_getBootstrapBalance()` implements the following formula:

```
(IEpochBasedVoteToken(bootstrapToken).pastBalanceOf(account_, epoch_)
    *   INITIAL_SUPPLY) / _bootstrapSupply
```

Note that the original balance of `account_` in the `bootstrapToken` gets scaled down by the factor `INITIAL_SUPPLY/_bootstrapSupply`.

In Version 3, the `INITIAL_SUPPLY` has been increased from `10_000` to `1_000_000`, improving the loss by a factor 100.

8.7 Possible Griefing With Governance Proposals

Informational Version 1 Acknowledged

CS-MZEROCORE-026

ZeroGovernor and EmergencyGovernor do not implement any measure to prevent attackers from proposing a large number of malicious proposals. Although such proposals do not get executed, assuming they do not receive the threshold of yes votes, they might be used to spam the system and make harder for users to find legit proposals.

Acknowledged:

M^ZERO Labs answered:

```
Intended behavior, zero and emergency governors proposals are optional to vote,
requiring thresholds of votes. Possible filtering can be done on a social level.
Gas fees on mainnet eventually should be spam prevention against such attacks.
```

8.8 Snapshots Remain in Storage

Informational **Version 1**

CS-MZEROCORE-037

Multiple contracts within the governance module maintain a record of changes for each account in storage. Typically, this information is only extended when new activity happens, and the old data remains uncleared. For example, mapping _lastSyncs stores the full history, although smart contracts only use the last entry.

In Version 3, the mapping _lastSyncs has been removed from the codebase.

9 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

9.1 Arbitrary Tokens Can Be Transferred to the Distribution Vault

Note **Version 2**

DistributionVault primarily receives cash tokens from the auction in the POWER token and fees of defeated proposals in the standard governor. It also receives the excess `M` tokens from the minter gateway. Zero holder can claim all tokens owned by the DistributionVault according to their balance in a given epoch.

Note that anyone can transfer (donate) arbitrary tokens to the vault. If such tokens are malicious, the accounting of the distribution vault might not work as expected. Zero holders should only claim tokens that are trustworthy.

9.2 Effects of Roundings in `MToken`

Note **Version 1**

The protocol extensively uses rounding up or down when it comes to working with principal values in the `MToken`. This has the following effects:

- for earners, the minting of `X` tokens will effectively mint `Y` tokens, with $Y \leq X$.
- for earners with initial token balances `A` and `B`, in-kind transfers of `X` tokens will result in updated balances `A'` and `B'`, with $A' \leq A - X$ and $B' \geq B + X$, and $A + B \leq A' + B'$.
- for out-of-kind transfers of `X` tokens and initial balances `A` and `B`, the updated balances `A'` and `B'` yield $A + B \geq A' - B'$.
- as soon as earners are active in the system, the invariant:

  $$\sum_{i \in nonEarningBalances} balanceOf(i) = totalSupply$$

  is relaxed to

  $$\sum_{i \in nonEarners} balanceOf(i) + \sum_{j \in earners} balanceOf(j) \leq totalSupply$$

- when non earners become earners, their balances go from `A` to `A'`, with $A' \leq A$.
- when earners stop earning, their balances go from `A` to `A'`, with $A \leq A'$.

Functions `transfer()`, `transferFrom()`, `mint()` and `burn()` update balances with amounts that might be different from those specified by callers due to rounding errors. The lost balances due to rounding will be counted by `excessOwedM` and get minted to the distribution vault. These specifics of `MToken` should be taken in consideration when integrating with 3rd-party protocols.

9.3 Inactive Minters Can Be Frozen

Note **Version 1**

The function `MinterGateway.freezeMinter` allows approved validators to freeze arbitrary addresses. But nothing prevents a non-active minter to be frozen.

M^ZERO Labs provides the following argument for allowing this behavior:

```
This is intentional. Validators can freeze a minter before they become activated,
in order to have time to properly access/setup/etc before that minter can start
minting. Abuse of this functionality can result in governance removing the validator.
```

9.4 Incentives for Accumulating Governance Tokens

Note **Version 1**

There is a circular incentive in the system to collect more governance tokens over time. Having `Zero` token allows users to claim rewards from the distribution vault. `Zero` tokens are emitted to `Power` holders that vote regularly on standard proposals. Collecting more `Zero` tokens from standard governor, allows users to buy more `Power` token with a discount from the Dutch auction, which increases `Zero` token emission when voting.

The system inherently encourages the ongoing collection of more governance tokens. Having `Zero` tokens enables users to claim rewards from the distribution vault. `Zero` tokens are issued to `Power` holders who regularly participate in voting on standard governor. Accumulating more `Zero` tokens through the standard governor enables users to purchase additional `Power` tokens at a discounted rate during the Dutch auction. This, in turn, amplifies the emission of `Zero` tokens when participating in voting.

9.5 Minter's Wallet Is Continuously Used

Note **Version 1**

The function `updateCollateral()` requires minters to submit transactions to the smart contract on a daily basis. Considering that minter's account is valuable in the system and it should be carefully protected (e.g., as a cold wallet), this might cause inconvenience to minters.

M^ZERO Labs provides the following argument for the design:

```
Ideally, a Minter's wallet should only be used to interact with the MinterGateway.
M minted by the Minter, should be sent to another wallet by passing a different
destination address in proposeMint. When the Minter wants to retrieve their collateral,
they will either have to ask the owner of the mint M tokens to call the burnM function
or acquire M on the secondary market to burn them themselves. Of course, Minters should
first and foremost ensure that they have established good security practices to avoid
any issues.
```

9.6 Minters Can Overwrite Mint Proposals

Note Version 1

Minters can have only one mint proposal at a time which is subject to a delay before it can be executed. Each proposal gets a unique id. To cancel a proposal, validators should pass the minter address and `mintId_` to `cancelMint()`.

This creates a front-running possibility as the minter can make a new proposal which gets a new id, therefore the validator's transaction canceling the old proposal reverts. However, the new proposal is still subject to the delay and validators can cancel it during this time. It is important that validators correctly parse events and always act as expected. Validators should cancel the respective mint proposals after minters submit collateral retrieve proposals.

9.7 Proposals Can Be Reordered

Note Version 1

The execution order of the `Succeeded` proposals can be arbitrary, they are not enforced to be executed in the same order they were proposed. This could lead to unexpected behaviors if multiple proposals are targeting the same parameters.

9.8 Recovery When System Incurs Losses

Note Version 1

It is possible that the system could mint uncollateralized MTokens when more interest is paid to earners than collected from minters. Although unlikely, this could happen if `updateIndex()` is not called for a long time (e.g., longer than the confidence interval of 30 days). The system does not implement an explicit recovery mechanism for such situations.

9.9 Solmate Libraries

Note Version 1

The external libraries are outside the scope of this code assessment. However, we would like to highlight that the contract `StableEarnerRateModel` uses the function `wadln()` from solmate which has the following disclaimer in their repository:

```
This is experimental software and is provided on an "as is" and "as available" basis.

While each major release has been audited, these contracts are not designed with user safety in mind:

    * There are implicit invariants these contracts expect to hold.
    * You can easily shoot yourself in the foot if you're not careful.
    * You should thoroughly read each contract you plan to use top to bottom.

We do not give any warranties and will not be liable for any loss incurred through
any use of this codebase.
```

9.10 Threshold Governors Can Ignore Majority

Note Version 1

The governors implementing `ThresholdGovernor`, i.e. `ZeroGovernor` and `EmergencyGovernor`, consider a proposal `Succeeded` if the ratio $\frac{yesVotes}{totalSupply} \geq quorum$. This means that if $quorum < 50\%$, the majority is ignored, and a proposal can pass even if it gets more no votes than `yes` votes.

M^ZERO Labs provides the following argument for allowing this behavior:

```
Thresholds will be significantly higher than 50%. If some reason governance lower
thresholds below 50%, the described situation is intended.
```

9.11 Update Interval Should Consider Delays on Collecting Signatures

Note **Version 1**

The updated interval parameter in the registrar decides the frequency that minters should update their collateral without paying a penalty. An update interval of `24 hours`, in reality, does not translate into a requirement to call `updateCollateral()` daily. As there is some delay from the moment a minter receives the first signature from a validator until the transaction executes on-chain, minters should update their collateral more frequently than the update interval. For example, at day 1 minter initiates the process of collecting signatures at 10:00am and the transaction is finalized after 1 hour, next day the minter should initiate the process at 09:00am such that `updateCollateral()` is executed before 10:00am (assuming the whole process always takes 1 hour).

9.12 Use of Power and Zero Tokens in 3Rd-Party Protocol

Note **Version 1**

Governance tokens `Zero` and `Power` play two important roles in the system:

- Maintain the protocol by voting on proposals.
- Claim M token rewards from the vault that help minters close their positions.

However, both tokens are implemented as ERC20 tokens and can be deposited in 3rd-party protocols (such as DEXes or lending protocols). If this happens, there are severe consequences for the system, as attacks that overtake governance majority become feasible (e.g., borrowing large amount of tokens in the last block of an epoch). Also, parts of rewards in the distribution vault might get locked.

9.13 Varias in Penalty Aggregation

Note **Version 1**

When computing the penalty amount for a "block" of penalties for missed collateral updates, i.e. update interval missed back-to-back, the penalty is not compounding, but it is between two distinct penalty blocks.

Within a penalty block, for 2 missed intervals, the amount is computed as `principalAmount * 2 * penaltyRate = (principalAmount * penaltyRate) + (principalAmount * penaltyRate`

), and for two not back-to-back missed intervals the amount is `(principalAmount * penaltyRate) + (principalAmount * (1 + penaltyRate)) * penaltyRate`.

9.14 ZERO Holders Buy PowerToken With a Discount

Note **Version 1**

Since `ZERO` holders can claim the cash tokens in the `DistributionVault`, they partially get back the amount they pay when buying `PowerToken` from the Dutch auction. This gives an advantage to `ZERO` holders proportionally to their balance.