Three Sigma Labs

Code Audit

Disclaimer

Code Audit

M^0 Labs Money middleware for the digital age

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

M^0 Labs Money middleware for the digital age

Table of Contents

Summary

Code Audit

M^0 Labs Money middleware for the digital age

Summary

Three Sigma Labs audited M^0 Labs in a 8 person week engagement. The audit was conducted from 08/01/2024 to 02/02/2024.

Protocol Description

The digital age demands open, decentralized, interoperable money technology. M^0 democratizes access to money issuance infrastructure. Based on a decentralized architecture and best-in-class collateral design, M^0 allows institutions to issue a cryptodollar.

Scope

Code Audit

M^0 Labs Money middleware for the digital age

Scope

Filepath	nSLOC
common/src/ContractHelper.sol	25
common/src/ERC20Permit.sol	87
common/src/ERC712.sol	55
common/src/SignatureChecker.sol	96
common/src/StatefulERC712.sol	10
protocol/src/ContinuousIndexing.sol	57
protocol/src/EarnerRateModel.sol	30
protocol/src/libs/ContinuousIndexingMath.sol	36
protocol/src/libs/SPOGRegistrarReader.sol	76
protocol/src/libs/UIntMath.sol	23
protocol/src/MinterRateModel.sol	15
protocol/src/MToken.sol	170
protocol/src/Protocol.sol	434
spog/src/abstract/BatchGovernor.sol	237
spog/src/abstract/EpochBasedInflationaryVoteToken.sol	137
spog/src/abstract/EpochBasedVoteToken.sol	215
spog/src/abstract/ERC5805.sol	38
spog/src/abstract/ThresholdGovernor.sol	100
spog/src/DistributionVault.sol	114
spog/src/EmergencyGovernor.sol	63
spog/src/EmergencyGovernorDeployer.sol	32
spog/src/libs/PureEpochs.sol	60
spog/src/PowerBootstrapToken.sol	22
spog/src/PowerToken.sol	170
spog/src/PowerTokenDeployer.sol	29
spog/src/Registrar.sol	78
spog/src/StandardGovernor.sol	233
spog/src/StandardGovernorDeployer.sol	48
spog/src/ZeroGovernor.sol	132
spog/src/ZeroToken.sol	105
SUM	2927

Methodology

Code Audit

M^0 Labs Money middleware for the digital age

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

M^0 Labs Money middleware for the digital age

Project Dashboard

Application Summary

Name	M^0 Labs
Commit	**common** 4a37119f2da946c6d8ad7b9a70dfdd219225115b **TTG** a8127901fa1f24a2e821cf4d9854a1aa6ac8088c **Protocol** 3499f50ff3382729f3e59565b19386ba61ef8e36
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	08-01-2024 to 02-02-2024
№ of Auditors	2
Review Time	8 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	1	0	0

Medium	3	0	0
Low	10	0	2
None	17	0	0

Category Breakdown

Suggestion	10
Documentation	0
Bug	16
Optimization	7
Good Code Practices	0

Code Maturity Evaluation

Code Audit

M^0 Labs Money middleware for the digital age

Code Maturity Evaluation

Code Maturity Evaluation Guidelines

Category	Evaluation
Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system.
Arithmetic	The proper use of mathematical operations and semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Code Stability	The extent to which the code was altered during the audit.
Upgradeability	The presence of parameterizations of the system that allow modifications after deployment.
Function Composition	The functions are generally small and have clear purposes.
Front-Running	The system's resistance to front-running attacks.
Monitoring	All operations that change the state of the system emit events, making it simple to monitor the state of the system. These events need to be correctly emitted.
Specification	The presence of comprehensive and readable codebase documentation.
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage.

Code Maturity Evaluation Results

Category	Evaluation
Access Controls	
Arithmetic	
Centralization	
Code Stability	
Upgradeability	
Function Composition	
Front-Running	
Monitoring	
Specification	
Testing and Verification	

Findings

Code Audit

M^0 Labs Money middleware for the digital age

Findings

3S-MZ-C01

Validator signature with zero timestamp can always be replayed

Id	3S-MZ-C01
Classification	Critical
Severity	Critical
Likelihood	Medium
Category	Bug
Status	Addressed in .

Description

When calling MinterGateway.updateCollateral, an array of timestamps gets passed. That is because the digest signed by a validator includes an updating timestamp.

The minimum timestamp between the signatures will be used as the new collateral update timestamp. This is used to check for staleness, which also has the side effect of preventing replaying a set of signatures:

```
    function _updateCollateral(address minter_, uint240 amount_, uint40
newTimestamp_) internal {
        uint40 lastUpdateTimestamp_ =
_minterStates[minter_].updateTimestamp;
        // MinterGateway already has more recent collateral update
        if (newTimestamp_ < lastUpdateTimestamp_) revert
StaleCollateralUpdate(newTimestamp_, lastUpdateTimestamp_);
        _minterStates[minter_].collateral = amount_;
        _minterStates[minter_].updateTimestamp = newTimestamp_;
    }
```

The issue arises when a timestamp signed by a validator is zero. This is explicitly allowed by the validator signature verification, which ignores zero values:

```
            // Find minimum between all valid timestamps for valid
```

```
signatures.
            minTimestamp_ = UIntMath.min40IgnoreZero(minTimestamp_,
UIntMath.safe40(timestamps_[index_]));
```

In the extreme scenario where every validator signature of the array has a zero timestamp, **`minTimestamp_`** will be **`block.timestamp`**, which will always pass the staleness check. This means that an array of signatures with zero timestamps can be replayed as many times as a minter wants, thus achieving a way for updating its collateral to that same value in the future, regardless of whether or not that's still true in the off-chain world. A minter could do the following:

1. Add a very large amount of real world collateral. Let's say $10M.

2. Call **`updateCollateral`** with the right amount. Let's assume validators will have signed with a zero timestamp, which is explicitly allowed by the protocol.

3. Redeem the entire real world collateral.

4. Call **`updateCollateral`** by replaying the original validator signatures. The protocol will still think the minter has all the collateral.

5. Mint the maximum allowed amount of **`MToken`**. Sell them somewhere at market price.

In a less extreme scenario where only 1 validator signs with a zero timestamp, we can still say that this specific signature can be reused anytime in the future, as long as the same collateral value is being used and that validator continues being approved by the TTG. The previous scenario can still be achieved if a minter rightfully calls **`updateCollateral`** multiple times with the same amount and rotates the validators until they finally get enough zero timestamp signatures from different validators.

Recommendation

Revert the signature verification if a timestamp is zero, by adding the following line to **`_verifyValidatorSignatures`**'s loop:

```
if (timestamps_[index_] == 0) revert ZeroTimestamp();
```

3S-MZ-H01

Lack of deadline in **`PowerToken.buy`** can lead to user's **`cashToken`** being distributed through **`ZeroToken`** holders

Id	3S-MZ-H01
Classification	High
Severity	High
Likelihood	Medium
Category	Bug
Status	

Description

The function PowerToken.buy is used to purchase tokens in auction. The price of those tokens decreases with time, following a dutch auction model. Because there's no deadline parameter in the function call, it's possible to retain this transaction and execute it in a subsequent auction at a steeper price, potentially resulting in a significant loss to the user, considering the price in an auction starts at **`2^99 wei`** per supply basis point.

Here's a likely scenario:

1. Auction A starts. Let's assume that **`cashToken`** is WETH.

2. Nobody buys the tokens until the price becomes more reasonable. For example, 1 ether per bps.

3. Alice max-approves the spending of her WETH for the **`PowerToken`** contract to transfer them. Alice holds 100 WETH in her wallet.

4. Alice tries to buy the tokens, but ends up paying a very low fee and the transaction ends up not being picked up for execution. In the meantime, market fees rise and the transaction gets stuck in the mempool. This is not uncommon, and there are thousands of mempool transactions with over a month.

5. A new auction B eventually starts. Starting price is the same.

6. A **`ZeroToken`** holder Bob submits Alice's transaction inside a flashbots bundle. Bob does this when the price is 100 ether per bps.

7. Alice ends up buying the desired amount of **`PowerToken`**. However, the price was much larger than she wanted, and she ended up spending 100 times more.

8. The WETH falls into the vault, where both Bob and the other **`ZeroToken`** holders will be able to claim their fair share.

It should be noted that, in theory, all **`ZeroToken`** holders have an economic incentive to be on the look for these buying attempts that get stuck on the mempool.

Recommendation

Add a deadline input to the function call, and check that **`block.timestamp`** is smaller than that deadline.

```
function buy(
    uint256 minAmount_,
    uint256 maxAmount_,
    address destination_,
    uint256 deadline_
) external returns (uint240 amount_, uint256 cost_) {
```

3S-MZ-M01

Creating a new proposal in **StandardGovernor** may reach a state of permanent DoS

Id	3S-MZ-M01
Classification	Medium
Severity	Medium
Likelihood	Medium
Category	Bug
Status	

Description

The function StandardGovernor.propose is used to create a new proposal to be executed in the **StandardGovernor**. If it's the first proposal for that epoch, it will call PowerToken.markNextVotingEpochAsActive to set the next target supply. Inside this function, there's a check to cap the next target supply at **type(uint240).max**:

```
        uint240 nextTargetSupply_ = _nextTargetSupply = UIntMath.bound240(
            uint256(_targetSupply) + (_targetSupply *
participationInflation) / ONE
        );
```

The issue is that the most likely scenario will be that the multiplication **_targetSupply * participationInflation** will overflow first, before any bound checks. Because **_targetSupply** is a **uint240** and **participationInflation** is a **uint16**, the multiplication will raise a panic overflow when it passes the maximum value of a **uint240**. This means that, in most cases, the transaction will revert with a panic overflow instead of just capping the next target supply.

Because the problematic function is called by **StandardGovernor.propose** for the first proposal of an epoch, this results in a permanent denial-of-service situation where it will no longer be possible to create new proposals in the **StandardGovernor**.

Recommendation

Cast **_targetSupply** to not allow a **uint240** overflow:

```
        uint240 nextTargetSupply_ = _nextTargetSupply = UIntMath.bound240(
            uint256(_targetSupply) + (uint256(_targetSupply) *
participationInflation) / ONE
        );
```

3S-MZ-M02

MToken's total principal invariant can be broken

Id	3S-MZ-M02
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	

Description

Because there's an unchecked total earning principal addition in MToken._addEarningAmount, the **MToken** contract would be subject to a potential silent overflow of **principalOfTotalEarningSupply**. However, there's a check in the MinterGateway.mintM function that's supposed to prevent this from ever happening.

But because **minterRate** and **earnerRate** can be different, it is possible that a silent overflow happening in **MToken** doesn't get caught by the check in **MinterGateway**. The transaction will not be reverted, and **principalOfTotalEarningSupply** will silently overflow, reaching a broken state. Because a silent overflow in **MToken** will lead to an additional mint of **MinterGateway.excessOwedM**, the function can still revert in a safe **uint112** cast within **MToken.mint**. But there's still possible mint amount values which will lead to a successful transaction.

Close to the point of overflow, a malicious minter could mint the appropriate amount to make **MToken** reach the broken state. The overflowed **principalOfTotalEarningSupply** might still be used to drive further impact to the protocol.

Recommendation

Use checked addition in **_addEarningAmount**.

```
    function _addEarningAmount(address account_, uint112 principalAmount_)
internal {
        unchecked {
            _balances[account_].rawBalance += principalAmount_;
```

```
        }
        principalOfTotalEarningSupply += principalAmount_;
    }
```

3S-MZ-M03

Validator signatures with greater timestamps can be reused in a subsequent **`updateCollateral`**

Id	3S-MZ-M03
Classification	Medium
Severity	Medium
Likelihood	Medium
Category	Bug
Status	

Description

When calling MinterGateway.updateCollateral, an array of timestamps gets passed. That is because the digest signed by a validator includes an updating timestamp. The minimum timestamp between the signatures will be used as the new collateral update timestamp. This is used to check for staleness, which also has the side effect of preventing replaying a set of signatures.

Provided the collateral amount stays the same, all signatures with a timestamp larger than the minimum timestamp between them can be reused for a subsequent update. This means that the minter may just need to get a signature from one validator and use all others. The new **`updateTimestamp`** may not be as large as if the minter had requested new signatures from all validators, but it still can allow the minter to delay the penalization time. Potentially, they can even misrepresent their on-chain collateral value during a small period, provided there's one malicious validator willing to forge a signature.

Recommendation

A gas-intensive solution would be to flag each new digest as used in storage. A less intensive mitigation would be to explicitly prevent a large deviation of the timestamps being used in an **`updateCollateral`** call.

3S-MZ-L01

Multiplication after division in **`StableEarnerRateModel`** leads to loss of precision

Id	3S-MZ-L01
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	

Description

In the StableEarnerRateModel.getSafeEarnerRate function, there's a log calculation detailed in a comment:

```
        // 6. ln(1 + (totalActive * (delta_minterIndex - 1) / totalEarning))
* SECONDS_PER_YEAR / dt = earnerRate
```

The code implementation of the log argument calculation follows the comment, but using fixed point arithmetic with 12 decimals. The problem is that a division by **`1e12`** is immediately followed by a **`1e12`** multiplication:

```
            1e12 + ((((uint256(totalActiveOwedM_) * (deltaMinterIndex_ -
1e12)) / 1e12) * 1e12) / totalEarningSupply_)
```

That division combined with the multiplication should cancel out in real world math. But because of integer division, this sets the first 12 decimals to zero.

Recommendation

Remove the division and multiplication by **`1e12`**.

3S-MZ-L02

A decrease in **`updateCollateralInterval`** will lead to unfair penalties

Id	3S-MZ-L02
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	Acknowledged

Description

The **`MinterGateway.updateCollateral`** function should be called once every **`updateCollateralInverval`**. If the minter fails to do this a penalty will be imposed on him next time he tries to update his collateral. This process is done through MinterGateway._imposePenaltyIfMissedCollateralUpdates function.

This function will in turn call MinterGateway._getMissedCollateralUpdateParameters to calculate the amount of intervals missed:

```
    function _getMissedCollateralUpdateParameters(
        uint40 lastUpdateTimestamp_,
        uint40 lastPenalizedUntil_,
        uint32 updateInterval_
    ) internal view returns (uint40 missedIntervals_, uint40 missedUntil_) {

        uint40 penalizeFrom_ = UIntMath.max40(lastUpdateTimestamp_,
lastPenalizedUntil_);
        // If brand new minter or `updateInterval_` is 0, then there is no
missed interval charge at all.
        if (lastUpdateTimestamp_ == 0 || updateInterval_ == 0) return (0,
penalizeFrom_);
        uint40 timeElapsed_ = uint40(block.timestamp) - penalizeFrom_;
        if (timeElapsed_ < updateInterval_) return (0, penalizeFrom_);
        missedIntervals_ = timeElapsed_ / updateInterval_;
        missedUntil_ = penalizeFrom_ + (missedIntervals_ * updateInterval_);
    }
```

The issue with this function arises when there's a decrease in **`updateCollateralInterval`**. It incorrectly assumes that the minter missed one or more intervals in the past based on the new **`updateCollateralInterval`**, even though this new value only became effective later on.

This will impose a penalty to the minter as if he missed updates to his collateral, when in reality he did not.

Here's an example:

- On January 1st, TTG executes a proposal that sets **`updateCollateralInterval`** to 1 month;

- On January 28th, Bob calls **`updateCollateral`**;

- On February 28th, Bob calls **`updateCollateral`** again;

- On March 27th, TTG executes a proposal that reduces **`updateCollateralInterval`** to 1 day;

- On March 28th, Bob calls **`updateCollateral`** so that he does not miss the new interval.

This last call, however, will impose an unfair penalty on Bob of 28 **`missedIntervals_`** when he did not missed any.

Recommendation

Change the logic in **`_getMissedCollateralUpdateParameters`** so that it checks if there was any decrease in **`updateCollateralInterval`** and, if so, calculate if there was any missed intervals before the proposal execution, using the old **`updateCollateralInterval`**. After that it should procede to calculate the amount of **`misssedIntervals_`** after the proposal execution using the new **`updateCollateralInterval`**.

3S-MZ-L03

Unhandled rounding error in **`DistributionVault.getClaimable`** leads to locked dust

Id	3S-MZ-L03
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	

Description

The DistributionVault.getClaimable function rounds down during the division operation used to calculate **`claimable`**, resulting in dust amounts being locked in **`DistributionVault`**.

```
    function getClaimable(
        address token_,
        address account_,
        uint256 startEpoch_,
        uint256 endEpoch_
    ) public view returns (uint256 claimable_) {
            // ...
            claimable_ += (distributionOfAt[token_][startEpoch_ + index_] *
balance_) / totalSupply_;
        }
    }
```

Every single token in the contract is meant to be retrieved by a specific user at a specific past epoch. This means that any rounding errors that might occur will lead to dust amounts being locked in the contract as **`DistributionVault`** does not have any mechanism that enables these funds to be retrieved.

The impact of this issue depends on a couple of points:

1. As token precision decreases, 1 unit of that token is more valuable than 1 unit of a token with higher precision. This will make rounding errors in the **`claimable`** calculation more

expensive for lower decimal tokens. So, the impact of this issue increases as the precision of the token claimed decreases.

2. The fact that the protocol makes a rounding down division for every past epoch will also increase the impact of this issue comparing to other vaults, as the dust amount will grow for every epoch iteration.

3. As the total supply of **`ZERO`** is more thinly distributed among holders, the amount of dust locked in the Vault will also increase:

- If no **`ZERO`** holder has more than 10% of the total supply, **`10 wei`** may be locked in the contract per epoch;

- If no **`ZERO`** holder has more than 5% of the total supply, **`20 wei`** may be locked in the contract per epoch;

- etc

3S-MZ-L04

MToken's total principal invariant doesn't hold without **MinterGateway**, leading to potential principal loss

Id	3S-MZ-L04
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	

Description

In function MToken._mint, when the account is an earner, the principal gets added to both its raw balance and to the global variable **principalOfTotalEarningSupply**. However, this addition is unchecked, allowing the total principle to silently overflow the maximum **uint112** amount. The final check assessing whether or not the total principal (earning and non earning supply) is greater than **type(uint112).max** will pass if the earning principal amount already overflowed before in **_addEarningAmount**.

This can further cause damage to earners if e.g. Alice is able to overflow Bob's balance:

1. Bob is going to call **stopEarning**.

2. Alice frontruns Bob's transaction and transfers him the exact principal amount to make Bob's balance equal **type(uint112).max + 2**.

3. When Bob calls **stopEarning**, there's an unsafe cast which will truncate his entire balance: **uint112 principalAmount_ = uint112(_balances[account_].rawBalance);**. Bob's new balance will be 1.

It should be noted that the **MinterGateway** prevents the invariant from easily breaking. But such an important **MToken** invariant should not be solely preserved by an external contract, as it can lead to potential problems in the future (e.g. changing the **MinterGateway** contract).

Recommendation

Consider removing the unchecked addition of **principalOfTotalEarningSupply** in **_addEarningAmount**:

```
    function _addEarningAmount(address account_, uint112 principalAmount_)
internal {
        unchecked {
            _balances[account_].rawBalance += principalAmount_;
        }
        principalOfTotalEarningSupply += principalAmount_;
    }
```

3S-MZ-L05

Unrealized inflation calculation returns wrong value when balance reaches cap

Id	3S-MZ-L05
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	

Description

The function EpochBasedInflationaryVoteToken._getUnrealizedInflation calculates a given account's inflation which has not been added to their balance yet. The inflation value gets incremented for each epoch the account/delegatee participated on.

The issue is that if the **`inflatedBalance_`** variable passes the **`type(uint240).max`** value, the function will return **`type(uint240).max`**, due to the following line:

```
                if (inflatedBalance_ >= type(uint240).max) return
type(uint240).max;
```

Let's imagine an unrealistic scenario where the current balance of an account is **`type(uint240).max - 1`**, that inflation from 2 participations is still unrealized, and that **`participationInflation`** is 10%.

1. In the first participation loop, **`inflation_`** is zero so **`inflatedBalance_`** will not reach the cap. The **`newInflation_`** value will be 10% of the initial balance, which also doesn't reach the cap.

2. In the second and last participation loop, **`inflatedBalance_`** will now be 110% of the initial balance, which will reach the cap. Thus, **`type(uint240).max`** will be returned as the unrealized inflation value.

It would have made more sense to force a cap on **`inflatedBalance_`** by setting it to **`type(uint240).max`** and continuing the loop execution. This way, the final inflation value

being returned would end up being about 20% of the initial balance, which does seem to make more sense. That being said, and considering the usage of **`_getUnrealizedInflation`** throughout the contract, it does seem like there's no meaningful impact associated to this.

Recommendation

Consider changing the line mentioned above:

```
                if (inflatedBalance_ >= type(uint240).max) {
inflatedBalance_ = type(uint240).max; }
```

3S-MZ-L06

Custom error for overflowing the total principal is not raised

Id	3S-MZ-L06
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	

Description

In the MToken._mint function, there's a check to make sure the total principal doesn't become larger than a **`uint112`**:

```
        if (
            principalOfTotalEarningSupply +
_getPrincipalAmountRoundedDown(totalNonEarningSupply) >= type(uint112).max
        ) {
            revert OverflowsPrincipalOfTotalSupply();
        }
```

Inside the condition, we're adding two **`uint112`** numbers, which will panic overflow if the result is greater than **`type(uint112).max`** (solidity 0.8 is being used). For this reason, the error inside the **`if`** statement will never actually be reached.

Recommendation

Cast one of the numbers to **`uint256`** to prevent the addition from panic overflowing:

```
        if (
            uint256(principalOfTotalEarningSupply) +
_getPrincipalAmountRoundedDown(totalNonEarningSupply) >= type(uint112).max
        ) {
            revert OverflowsPrincipalOfTotalSupply();
```

```
}
```

3S-MZ-L07

Proposals in the same voting period can have different ids but do the same

Id	3S-MZ-L07
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	

Description

In the BatchGovernor contract, a new proposal will generate a proposal id by using the calldata and the vote start. The **`_revertIfInvalidCalldata`** function makes sure the function selector is allowed (implemented in **`StandardGovernor`**, **`EmergencyGovernor`** and **`ZeroGovernor`**).

The issue is that the proposal calldata can have appended "dust" bytes in the end and it will still be considered valid. Anyone can create a proposal generating a different id but doing functionally the same thing as another existing proposal, because the "dust" bytes will be ignored on the execution moment but not on the id generation.

For example, to propose the call **`EmergencyGovernor.setStandardProposalFee(10)`**, the 2 following calldata values will generate 2 different proposal ids accomplishing the same thing:

- **`0xb5775125000000000000000000000000000000000000000000000000000000000000000a`**

- **`0xb5775125000000000000000000000000000000000000000000000000000000000000000affffffff`**

Allowing different proposal ids to accomplish the same thing in the same voting period can potentially force the voters to randomly decide between the 2 valid proposals, which might break the voting quorum. Because the **`EmergencyGovernor`** doesn't have a proposal fee (**`StandardGovernor`** has one), it is easier to spam with a number of different proposals doing the same thing as a legitimate one, which can cause confusion among voters and split the quorum.

Recommendation

Consider checking the calldata size in the function **`_revertIfInvalidCalldata`**. For example, the implementation in the **`StandardGovernor`** would be the following:

```
    function _revertIfInvalidCalldata(bytes memory callData_) internal pure
override {
        bytes4 func_ = bytes4(callData_);
        uint256 length = callData_.length;
        if (
            !(func_ == this.addToList.selector && length == 68) &&
            !(func_ == this.removeFromList.selector && length == 68) &&
            !(func_ == this.removeFromAndAddToList.selector && length ==
100) &&
            !(func_ == this.setKey.selector && length == 68) &&
            !(func_ == this.setProposalFee.selector && length == 36)
        ) revert InvalidCallData();
    }
```

3S-MZ-L08

Function **`cancelMint`** can be frontrunned to grief a validator

Id	3S-MZ-L08
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	Acknowledged

Description

In the MinterGateway.cancelMint function, an approved validator cancels a mint proposal by passing the minter address and **`mintId`**. The function will revert if the the minter's proposal doesn't correspond to the input id. Each **`mintId`** is generated from a nonce, and this is independent from the proposal details.

A minter can frontrun any **`cancelMint`** call by calling **`proposeMint`** with the exact same parameters as their mint proposal that was about to be cancelled. This will revert the validator's transaction, because the specified id will no longer be correct. A minter can keep doing this until they potentially get frozen by the validator.

3S-MZ-L09

StandardGovernor's implementation of **quorum** is incompatible with Tally

Id	3S-MZ-L09
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	

Description

The StandardGovernor contract inherits from **IGovernor**, which states the following:

```
/// @title Minimal OpenZeppelin-style, Tally-compatible governor.
interface IGovernor is IERC6372, IERC712 {
```

But the **StandardGovernor** contract implements **quorum** functions in the following way:

```
    /// @inheritdoc IGovernor
    function quorum() external pure returns (uint256) {
        return 0;
    }
    /// @inheritdoc IGovernor
    function quorum(uint256) external pure returns (uint256) {
        return 0;
    }
```

The documentation from Tally states that it "needs the quorum to calculate if a proposal has passed." Returning quorum as zero, even though that's not the quorum threshold's value, will not only break some compatibility with Tally, but it also might bring about unexpected outputs to other smart contracts within the Ethereum ecosystem expecting the **StandardGovernor** to follow full Tally compatibility, as stated in **IGovernor**, thus breaking modularity.

Recommendation

Consider properly implementing the **`quorum`** functions. Optionally, be explicit about this lack of compatibility.

3S-MZ-L10

A sufficiently large collateral may break the maximum owed M calculation

Id	3S-MZ-L10
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	

Description

In the MinterGateway contract, the **`updateCollateral`** function can set a collateral value to any value, provided the value is no larger than **`type(uint240).max`** and the data is signed by enough validators. The **`maxAllowedActiveOwedMOf`** function will use the collateral value to assess if a minter is undercollateralized:

```
    function maxAllowedActiveOwedMOf(address minter_) public view returns
(uint256) {
        // NOTE: Since `mintRatio()` is capped at 10,000% (i.e. 1_000_000)
this cannot overflow.
        unchecked {
            return _minterStates[minter_].isActive ?
(uint256(collateralOf(minter_)) * mintRatio()) / ONE : 0;
        }
    }
```

The comment note in **`maxAllowedActiveOwedMOf`** is incorrect, because the multiplication of a large **`mintRatio`** with a large collateral can overflow a **`uint256`**. In the unlikely scenario that validators allow a very odd and large collateral value, this multiplication will silently overflow due to the **`unchecked`** block, leading to a wrong calculation of the maximum allowed debt.

Recommendation

Remove the **`unchecked`** block in this situation to avoid a silent overflow. For better readability, consider raising a custom error instead of letting the compiler raise a panic error upon overflow.

3S-MZ-N01

Missing **`override`** keyword for interface inherited methods

Id	3S-MZ-N01
Classification	None
Category	Suggestion
Status	

Description

Most contracts are not implementing their interface methods with **`override`** keywords. For example, the DistributionVault contract implements all its interface functions, but it doesn't add the **`override`** keyword to any of the functions. This means the compiler won't check that the implementation is properly implementing what is defined on the interface, making it prone to spelling errors and function signature mismatching.

Recommendation

Add the **`override`** keyword to those contracts functions as a best practice and for compiler checks.

3S-MZ-N02

Some contracts don't implement their entire interface

Id	3S-MZ-N02
Classification	None
Category	Suggestion
Status	

Description

There are some contracts that don't fully implement the interface with the same name. For example, IERC5805 defines functions such as **`delegates`**, **`getPastVotes`** and **`getVotes`**. These functions are not implemented in **`ERC5805`**, only in **`EpochBasedVoteToken`**.

Recommendation

Implement all interface functions in the contract with the same name, even if said implementation remains empty and virtual.

3S-MZ-N03

No need to set **`isActive`** to **`false`** if that mapping entry was deleted

Id	3S-MZ-N03
Classification	None
Category	Bug, Optimization
Status	

Description

In MinterGateway.deactivateMinter the minter's entry in **`_minterStates`** is deleted using the **`delete`** keyword:

```
    function deactivateMinter(address minter_) external
onlyActiveMinter(minter_) returns (uint240 inactiveOwedM_) {
        // ...
        delete _minterStates[minter_];
        delete _mintProposals[minter_];
        _minterStates[minter_].isDeactivated = true;
        _minterStates[minter_].isActive = false;
        // ...
}
```

This action sets all types of that specific entry to its default value, and so, there is no need to set **`isActive`** to **`false`** as this is the default value set by the **`delete`** keyword.

Recommendation

Remove the following line from **`deactivateMinter`** function:

```
        _minterStates[minter_].isActive = false;
```

3S-MZ-N04

Unnecessary Recomputation of Storage Pointer in `MToken._startEarning`

Id	3S-MZ-N04
Classification	None
Category	Optimization
Status	

Description

In MToken._startEarning function, the **`_balances`** mapping is stored in the **`mBalance_`** variable.

```
MBalance storage mBalance_ = _balances[account_];
```

But a few lines below, instead of using **`mBalance_`** to retrieve **`rawBalance`**, **`_balances[account].rawBalance`** is used again spending unnecessary gas.

```
    function _startEarning(address account_) internal {
        emit StartedEarning(account_);
        MBalance storage mBalance_ = _balances[account_];
        if (mBalance_.isEarning) return;
        mBalance_.isEarning = true;
        // Treat the raw balance as present amount for non earner.
        uint240 amount_ = _balances[account_].rawBalance;
```

Recommendation

Change the **`amount_`** variable to:

```
uint240 amount_ = mBalance_.rawBalance;
```

3S-MZ-N05

Unnecessary check in `StandardGovernor.state`

Id	3S-MZ-N05
Classification	None
Category	Optimization
Status	

Description

Since StandardGovernor._votingPeriod function will always returns `0`, `voteStart` will always be equal to `voteEnd`.

This means that the following check in the `StandardGovernor.state` is unnecessary:

```
if (currentEpoch_ <= voteEnd_) return ProposalState.Active;
```

The use of `<` spends unnecessary gas by checking that `currentEpoch_` is less than `voteEnd_`.

If `currentEpoch_` is indeed less that `voteEnd_` then the transaction will stop a few lines above:

```
if (currentEpoch_ < voteStart_) return ProposalState.Pending;
```

Recommendation

The check for strict equality (`==`) will suffice.

3S-MZ-N06

Code should not panic underflow

Id	3S-MZ-N06
Classification	None
Category	Suggestion
Status	

Description

There's several places in the protocol where proper code validation is missing by, instead, relying on panic errors.

This is not recommended behaviour as per Solidity Docs:

Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix.

Here are the instances that rely on panic errors caught during the review:

- MToken._subtractEarningAmount

- MToken._subtractNonEarningAmount

- ERC20Extended.transferFrom

- DistributionVault.getClaimable

- PowerToken._divideUp

Recommendation

Raise proper errors instead of relying on panic.

3S-MZ-N07

StartedEarning event is emitted even if account is already earning

Id	3S-MZ-N07
Classification	None
Category	Optimization
Status	

Description

In MToken._startEarning the **`StartedEarning`** event will be emitted even if the calling account is already earning:

```
function _startEarning(address account_) internal {
    emit StartedEarning(account_);
    MBalance storage mBalance_ = _balances[account_];
    if (mBalance_.isEarning) return;

    // ...
}
```

The same happens in the MToken._stopEarning. The **`StoppedEarning`** event will be emitted even if the user isn't an earner.

Recommendation

Move the line that emits the event to the end of the function so that it will only fire when a new user starts/stops being an earner.

3S-MZ-N08

Wrong comment in Standard Governor's **`execute`** function

Id	3S-MZ-N08
Classification	None
Category	Suggestion
Status	

Description

The following comment is present in StandardGovernor's **`execute`** function:

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

This, however, is not the current behaviour of the contract. As confirmed by the client, approved proposals can only be executed 1 epoch after **`voteEnd`** and not 2 epochs as the comment suggests.

Recommendation

Update the comment so it explains the actual behaviour of epoch execution:

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

3S-MZ-N09

Unnecessary **currentEpoch** zero check in **StandardGovernor** and **ThresholdGovernor**

Id	3S-MZ-N09
Classification	None
Category	Optimization
Status	

Description

The **currentEpoch** value is derived from BatchGovernor._clock, which can never be zero. But the **execute** functions of both **StandardGovernor** and **ThresholdGovernor** check if **currentEpoch** is zero.

Recommendation

Since the epoch implementation being used doesn't allow for a zero value returned by **_clock**, consider removing these zero checks.

3S-MZ-N10

Overflow check in **`PowerToken._divideUp`** is unnecessary

Id	3S-MZ-N10
Classification	None
Category	Optimization
Status	

Description

The PowerToken._divideUp function checks if a result of a multiplication "wrapped" around the maximum number, i.e. if it silently overflowed:

```
z = (x * ONE) + y;
if (z < x) revert DivideUpOverflow();
```

Because calculation of `z` is not unchecked, the condition `z < x` will never be true, since solidity 0.8 checks and reverts on overflow.

Recommendation

Remove the line checking the `z < x` condition.

3S-MZ-N11

Unnecessary conditional check in **`ThresholdGovernance.execute`**

Id	3S-MZ-N11
Classification	None
Category	Optimization
Status	

Description

The function ThresholdGovernor.execute executes a successful proposal. Among other things, the function executes the following lines:

```
        if (currentEpoch_ == 0) revert InvalidEpoch();
        // Proposals have voteStart=N and voteEnd=N+1, and can be executed
only during epochs N and N+1.
        uint16 latestPossibleVoteStart_ = currentEpoch_;
        uint16 earliestPossibleVoteStart_ = latestPossibleVoteStart_ > 0 ?
latestPossibleVoteStart_ - 1 : 0;
```

The **`execute`** function will revert if **`currentEpoch_ == 0`**, and then it assigns this value to **`latestPossibleVoteStart_`**. Considering this last variable can never be zero, the conditional check in the following line is unnecessary.

Recommendation

Replace the last lined shown above with the following:

```
        uint16 earliestPossibleVoteStart_ = latestPossibleVoteStart_ - 1;
```

3S-MZ-N12

Inconsistent naming of function in **`PureEpochs`**

Id	3S-MZ-N12
Classification	None
Category	Suggestion
Status	

Description

The PureEpochs library implements a variety of functions centered on the definition of epochs. For example, it includes the following functions:

- **`getTimeSinceEpochStart`**

- **`getTimeSinceEpochEnd`**

- **`getTimeUntilEpochStart`**

It also includes the function **`getTimeUntilEpochEnds`**. The naming of this function in particular is inconsistent with the remaining ones.

Recommendation

Change the name of the **`getTimeUntilEpochEnds`** function to **`getTimeUntilEpochEnd`**.

3S-MZ-N13

`_checkAndIncrementNonce` in `ERC5805` raises a `ReusedNonce` error for non used nonces

Id	3S-MZ-N13
Classification	None
Category	Suggestion
Status	

Description

The ERC5805._checkAndIncrementNonce function makes sure the nonce being used in a signature is the right one. If it is, it increments it:

```
    function _checkAndIncrementNonce(address account_, uint256 nonce_)
internal {
        uint256 currentNonce_ = nonces[account_];
        if (nonce_ != currentNonce_) revert ReusedNonce(nonce_,
currentNonce_);
        unchecked {
            nonces[account_] = currentNonce_ + 1; // Nonce realistically
cannot overflow.
        }
    }
```

The error being raised when `nonce_ != currentNonce_` is misleading. The `if` statement doesn't necessarily mean that the `nonce_` value being used has been used in past signatures, rather it means that it is not the expected current nonce of the account. For example, if the current nonce is 1 and the user signs the data with nonce 3, this should revert, but nonce 3 hasn't been used yet.

Recommendation

Consider renaming the error to more accurately describe the issue. For example, `NotCurrentNonce`.

3S-MZ-N14

castVoteWithReason always fires a **VoteCast** event with an empty reason

Id	3S-MZ-N14
Classification	None
Category	Suggestion
Status	

Description

The BatchGovernor contract supports **castVoteWithReason** to be compatible with certain governors. Because the **VoteCast** event has a **reason** parameter, whoever uses this external function is expecting an event being emitted with the provided reason. Instead, the **reason** parameter is always empty in the event being emitted by **_castVote**:

```
    function _castVote(address voter_, uint256 weight_, uint256 proposalId_,
uint8 support_) internal virtual {
        // ...
        emit VoteCast(voter_, proposalId_, support_, weight_, "");
    }
```

Recommendation

Make sure that the **reason** input of **castVoteWithReason** is passed through to the **VoteCast** event.

3S-MZ-N15

`transferFrom` in **`ERC20Extended`** will always emit an **`Approval`** event if the allowance changes

Id	3S-MZ-N15
Classification	None
Category	Suggestion
Status	

Description

The ERC20Extended.transferFrom function is using **`_approve`**for changing allowance, and this internal function always fires up an **`Approval`** event.

Recommendation

The token should not be firing this event unless by calling the external **`ERC20.approve`** function. Other common implementations (OpenZeppelin, solmate) also don't fire this event when allowance changes due to a **`transferFrom`** call.

3S-MZ-N16

EIP712's **`_revertIfError`** should use all **`SignatureChecker.Error`** errors

Id	3S-MZ-N16
Classification	None
Category	Suggestion
Status	

Description

Function ERC712._revertIfError goes through the different error possibilities in **`SignatureChecker.Error`** and raises the right error accordingly. But some are missing: **`InvalidSignatureS`** and **`InvalidSignatureV`**.

Recommendation

Consider either using the missing ones for better error messaging instead of raising the more general **`InvalidSignature`** error, or removing the unused ones from the enum.

3S-MZ-N17

The **`IERC3009`** interface is not fully conforming to the standard

Id	3S-MZ-N17
Classification	None
Category	Suggestion
Status	

Description

The IERC3009 defines a token interface following the EIP3009 standard. The interface is not fully conforming to the standard and it's missing the following mandatory view functions:

- **`TRANSFER_WITH_AUTHORIZATION_TYPEHASH`**

- **`RECEIVE_WITH_AUTHORIZATION_TYPEHASH`**

Because the interface is also adding the optional canceling functions from EIP3009, it should also be defining the **`CANCEL_AUTHORIZATION_TYPEHASH`** view function.

Recommendation

Though the ERC3009 contract does implement these functions, it is still recommended that the interface implements them as well.