Gloop Finance Audit Report

Version 1.0

Conducted by: Kiki

March 2025

Table of Contents

1 Introduction

1.1 About Kiki

Kiki is a Security Researcher who has conducted dozens of security reviews with the top security firm Guardian Audits, as well as through private engagements. View their previous work here, or reach out via Twitter or Telegram.

1.2 Disclaimer

Security Reviews are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

1.3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

1.3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

1.3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

2 Executive Summary

2.1 Overview

Project Name	Gloop Finance
Codebase	gm-lending-protocol
Operating platform	Arbitrum
Language	Solidity
Initial commit	[1] c6b2ea92d18a2d6a9efb95ea5abf947e3bda0c67
Remediation commit	55a6f06a92374696659ae0b4734e16c85fbfc132
Audit methodology	Manual Review

2.2 Scope

Files and folders in scope

- src/LendingPool.sol
- src/GMVault.sol
- src/GMUSDCVault.sol
- src/GMPriceOracle.sol
- src/GMPoints.sol
- src/GMInterestRateModel.sol
- src/GMIncentives.sol
- src/AddressStore.sol
- src/token/GloopStaking.sol

2.3 Issues Found

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	1	1	0	0
High risk	2	2	0	0
Medium risk	7	7	0	0
Low risk	11	11	0	0

2.4 Findings & Resolutions

ID	Title	Severity	Status
C-01	DoS via Unlimited Referral Spam	Critical	Resolved
H-01	Interest Rate Miscalculation Due to L1 Block Usage	High	Resolved
H-02	Missing Interest Accrual in disableAsset function	High	Resolved
M-01	Withdrawal check missing borrow factor application	Medium	Resolved
M-02	Profitable Self-Liquidation Due to Bonus Overlap	Medium	Resolved
M-03	Borrowing Enabled During Liquidation Grace Period	Medium	Resolved
M-04	Reward Removal Prevents Future Claims	Medium	Resolved
M-05	Precision Loss in Points Calculation	Medium	Resolved
M-06	Exploitable Referral Boost Mechanism	Medium	Resolved
M-07	Incorrect Reward Timestamp Update Skips Periods	Medium	Resolved
L-01	Interest Accrues During Vault Pause Period	Low	Resolved
L-02	Rounding Direction for Debt Values	Low	Resolved
L-03	Missing Health Factor Check Post-Borrow	Low	Resolved
L-04	Missing Admin Function for Bad Debt Liquidation	Low	Resolved
L-05	Incorrect Block Number Comparison	Low	Resolved
L-06	Exceeding Maximum GM Tokens in Collateral	Low	Resolved
L-07	Missing Validation for Oracle Price	Low	Resolved
L-08	Unbounded GM Token Array Creates DoS Risk	Low	Resolved
L-09	Misleading Revert Message in Rewards Claim	Low	Resolved
L-10	Misleading Comment in Debt Update Logic	Low	Resolved
L-11	Centralization Risks	Low	Resolved

3 Findings

3.1 Critical Risk

3.1.1 DoS via Unlimited Referral Spam

Severity: *Critical risk (Resolved)*

Context: ***GMPoints.sol***:133

Description:

In the `GMPoints` contract, there is no limit on the number of referrals that can be added to a user's `userRefs[_referrer].referrals` array. This creates a potential Denial of Service (DoS) vulnerability:

1. An attacker can create multiple addresses and set a victim's address as the `_referrer`.
2. Each time this happens, the victim's referral array grows via:

```
userRefs[_referrer].referrals.push(msg.sender);
```

3. When the victim tries to perform actions that iterate through their referrals array (such as withdrawing or repaying), the transaction will fail due to exceeding the block gas limit.

The issue is particularly severe because referrals cannot be removed once added, which could permanently lock the victim out of withdrawing their funds or using the protocol.

Recommendation: Implement the following two solutions to prevent the DoS attack as well as any griefing vectors.

- A pull method for users to accept being a referrer. This would give users control over how many referrals are tied to their account, preventing this DoS attack.
- Introduce a maximum referral limit per account. Since a max bonus already exists, there is no need for any user to exceed that set amount.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.2 High Risk

3.2.1 Interest Rate Miscalculation Due to L1 Block Usage

Severity: *High risk (Resolved)*

Context: ***LendingPool.sol***:940

Description:

The `LendingPool` contract incorrectly uses L1 block numbers (`block.number`) as the `blockDelta` while interest rate is based on Arbitrum's much faster block production. This creates a significant discrepancy in interest accrual since L1 blocks are produced every 12 seconds while Arbitrum

blocks are produced every ~0.25 seconds. This means that for a year's worth of interest, it is expected that 126,144,000 blocks will pass, since this is roughly Arbitrum's expected block production over the course of a year. But when the interest is actually accrued, only about 2,628,000 blocks will be produced, resulting in only about ~2.1% of the expected interest being accrued. This results in significantly reduced interest accrual and diminished yields for liquidity providers.

Recommendation:

Replace block number-based time tracking with timestamp-based calculations to ensure consistent interest accrual.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.2.2 Missing Interest Accrual in disableAsset function

Severity: *High risk (Resolved)*

Context: ***LendingPool.sol***:*719*

Description:

The `disableAsset` function fails to accrue interest before allowing users to disable their collateral assets. This creates an issue where users can disable collateral when they shouldn't be allowed to, giving an attacker the opportunity to put their account in an insolvent state.

The function checks if disabling collateral would bring the user's health factor below 1:

```
require(
    calculateHFAfterCollChange(asset, msg.sender, balanceOf(asset, msg.sender),
        false) >= 1e18,
    "Disabling asset would bring Health Factor below 1"
);
```

However, since interest isn't accrued first, the user's borrow balance is stale and doesn't include pending interest. Allowing a user to disable collateral when their actual health factor (after interest) would be < 1 leads to users removing enough collateral that their position becomes either undercollateralized or potentially insolvent, effectively stealing from LPs and leaving the protocol with bad debt.

Recommendation:

Add interest accrual at the start of the `disableAsset` function.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.3 Medium Risk

3.3.1 Withdrawal check missing borrow factor application

Severity: *Medium risk (Resolved)*

Context: ***LendingPool.sol***:*402*

Description:

In the `_withdraw` function, the withdrawal check for users with active borrows is overly restrictive due to inconsistent factor application. The issue occurs because `withdrawValue` only applies the lend factor while being compared against `maxBorrowableValue` which applies both lend and borrow factors.

Consider the following scenario with GM tokens at $1/token and factors of 0.5:

- Initial collateral: 100 GM ($100 USD)
- Max borrowable value: $100 * 0.5 * 0.5 = $25 USD
- Current borrowed: $5 USD
- Remaining borrowable: $20 USD

Current implementation limits withdrawals to 40 GM:

```
withdrawValue = 40 GM * $1 * 0.5 = $20 USD

$20 USD + $5 USD = $25 USD (equals maxBorrowableValue)
```

After this withdrawal, the position has:

- Remaining collateral: 60 GM ($60 USD)
- maxBorrowableValue: 60 * $1 * 0.5 * 0.5 = $15 USD
- Current borrowed: $5 USD

Despite only having $5 USD borrowed against a $15 USD borrowing capacity, the user cannot withdraw more collateral because `withdrawValue` is not reduced by the borrow factor like `maxBorrowableValue`.

The missing borrow factor application in `withdrawValue` calculation artificially restricts users from withdrawing collateral even when their position would remain healthy after the withdrawal.

Recommendation:

Apply both lend and borrow factors when calculating `withdrawValue` to match the factor application in `maxBorrowableValue`:

```
uint256 withdrawValue = amount

.mulDivDown(oracle.getUnderlyingPrice(asset), baseUnits[asset])

.mulDivDown(configurations[asset].lendFactor, 1e18)

.mulDivDown(configurations[borrowAsset].borrowFactor, 1e18); //=> Add borrow factor*
```

This ensures consistent factor application and allows users to withdraw the maximum amount of collateral while maintaining a healthy position.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.3.2 Profitable Self-Liquidation Due to Bonus Overlap

Severity: *Medium risk (Resolved)*

Context: ***LendingPool.sol***:*614*

Description:

In the `LendingPool` contract, there is no validation ensuring the `liquidationBonus` cannot exceed the factors of collateral assets. This creates a profitable self-liquidation vulnerability that can be exploited to drain the protocol. At its core, this is possible because users can repay an amount less than they borrowed, but due to the bonus, receive the full collateral amount for the underpayment.

To demonstrate the severity, consider the following scenario with the product of `borrowFactor` and `lendFactor` being **90%** and a `liquidationBonus` of **125%:**

Initial State:

- User has 100 GM tokens ($100 value)
- Protocol configuration: `lendFactor` = 90%, `liquidationBonus` = 125%

Attack Steps:

1. Deposit 100 GM tokens as collateral ($100 value)
2. Borrow 90 USDC (90% of collateral value per `lendFactor`)
3. Any amount of fees accrue allowing Self-liquidate position:
    - Required repayment = $100/1.25 = 80 USDC (due to `liquidationBonus`)
    - Collateral seized = 100 GM tokens ($100 value)

Final State:

- User now has: 100 GM tokens + 10 USDC (90 borrowed - 80 repaid)
- Net profit: $10 value per cycle

This cycle can be repeated to continuously extract value from the protocol, as there are no safeguards preventing this configuration.

Recommendation:

Add configuration bounds and validation in both `updateLiquidationBonus` and `configureAsset` functions:

1. Cap `liquidationBonus` to be less than 10% (1.1e18)
2. Require GM token `lendFactor` configurations to be at most 90% (0.9e18)

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.3.3 Borrowing Enabled During Liquidation Grace Period

Severity: *Medium risk (Resolved)*

Context: ***LendingPool.sol**:459*

Description: The `borrow` function in the `LendingPool` contract allows users to take out loans even during the grace period when liquidations are disabled. When a vault is unpaused after an emergency, there is a configurable delay before liquidations resume. During this grace period, users can still borrow assets while being temporarily protected from liquidation. Since any liquidation attempt would fail due to the following check:

```
if (assetVault.paused() || block.timestamp < assetVault.liquidationsResumeTimestamp
    ()) {
        revert("Liquidations currently disabled");
 }
```

This creates a serious risk vector where malicious users could: 1. Take out maximum loans during the grace period on riskier assets 2. During the protected time account health moves from unhealthy (but solvent) to insolvent 3. Once liquidated, the protocol will be left with bad debt that cannot be fully recovered.

This would impact both LP's as well as the protocols overall health. Leaving the protocol in an insolvent state.

Recommendation: Add a check in the `borrow()` function to ensure liquidations are currently enabled:

```
function borrow(ERC20 asset, uint256 amount) external nonReentrant {
    IGMVault vault = vaults[asset];
    require(block.timestamp >= vault.liquidationsResumeTimestamp(), "Borrowing
        disabled during grace period");
    // ... rest of borrow logic
}
```

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.3.4 Reward Removal Prevents Future Claims

Severity: *Medium risk (Resolved)*

Context: ***GMIncentives.sol***:138

Description:

The `removeReward` function in the `GMIncentives` contract allows the owner to remove a reward token once the reward period has ended. However, this implementation has a flaw: users who last claimed their rewards before the end of the reward period will be unable to claim any accumulated rewards for the removed token.

Once a token is removed, the contract no longer recognizes it, effectively locking any unclaimed rewards. This oversight can lead to a loss of yield, as users are prevented from accessing rewards they have rightfully earned.

Recommendation:

Before removing a reward token, ensure that all users have claimed their rewards. Alternatively, implement a grace period allowing users to claim rewards for a limited time after a token is removed, ensuring they can access their accumulated rewards.

Resolution: Resolved. The recommended fix was implemented in **5939e4e**.

3.3.5 Precision Loss in Points Calculation

Severity: *Medium risk (Resolved)*

Context: ***GMPoints.sol***:249-271

Description:

In the `GMPoints` contract, multiple instances of division operations are performed before multiplication when calculating points, leading to unnecessary precision loss. This issue occurs in both lending and borrowing points calculations:

1. For lending points:

```
floatingLendPoints = ((prevBalance / 1e6) * pointParams.lendingUSDCPPD *
    timeSinceLastUpdate * totalBoost) / PRECISION;
```

2. For borrowing points:

```
floatingBorrowPoints = ((currentBorrowBalance / 1e6) * pointParams.
    borrowingUSDCPPD * timeSinceLastUpdate * totalBoost) / PRECISION;
```

In both cases, dividing by `1e6` first causes precision loss in the final points calculation, leading to understated rewards for users.

This issue is particularly impactful because the precision loss can be up to 0.99 USDC per calculation. Since points are calculated frequently, this loss compounds over time, potentially leading to a significant understatement of earned points.

Recommendation:

Rearrange the calculations to perform all multiplications before divisions to maintain maximum precision.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.3.6 Exploitable Referral Boost Mechanism

Severity: *Medium risk (Resolved)*

Context: ***GMPoints.sol**:190-210*

Description:

The `calculateCurrentBoost` function in the `GMPoints` contract can be exploited to artificially inflate a user's referral boost. By creating multiple accounts with the minimum deposit and setting oneself as the referrer, a user can quickly achieve the maximum referral boost.

This manipulation undermines the integrity of the points system, allowing users to gain an unfair advantage and potentially distorting the distribution of rewards.

Recommendation:

- Introduce a mechanism to dynamically adjust the minimum deposit amount to counteract exploitation.
- Alternatively, implement a blacklist feature to nullify the referral count of malicious referrers, ensuring that only legitimate referrals contribute to the boost. This could be accomplished in the `calculateCurrentBoost` function.

Resolution: Resolved. The recommended fix was implemented in **deddb3e**.

3.3.7 Incorrect Reward Timestamp Update Skips Periods

Severity: *Medium risk (Resolved)*

Context: ***GMIncentives.sol***:275

Description:

In the `_updateRewardData` function, the `lastUpdateTimestamp` can incorrectly updated when both the emissions and reward index doesn't change. When unchanged (indicating no new rewards should be distributed), the function still updates `lastUpdateTimestamp` to the current block timestamp in the `else` branch:

This premature timestamp update causes the next call to `_getRewardIndex` to calculate an incorrect time delta, as it uses this artificially updated timestamp instead of the last actual reward distribution. This effectively "skips" periods where rewards should have accrued, leading to:

- Users missing rewards they should have earned
- Incorrect reward distribution calculations
- Potential manipulation of reward distributions by repeatedly calling functions that trigger this update

The bug affects all reward distributions in the protocol and could result in a loss of rewards for users.

3.4 Recommendation

Scale up the reward index to 1e18 so that small amount of index changes can still be accounted for. It will be important to also scale down the index when applying it to individual rewards.

Resolution: Resolved. The recommended fix was implemented in **deddb3e**.

3.5 Low Risk

3.5.1 Interest Accrues During Vault Pause Period

Severity: *Low risk (Resolved)*

Context: ***GMVault.sol***:44-55

Description:

The `pause` and `unpause` functions in the `GMVault` contract allow interest to incorrectly accrue during the pause period. When the vault is paused, the interest calculation continues to use the time delta between the last interest accrual and the current time, including the period when the vault was paused. This leads to users being charged interest for the pause period, potentially causing accounts to become unhealthy unfairly.

Recommendation:

Modify the functions to properly handle interest accrual around pause periods. The `pause` function should accrue all pending interest immediately before pausing, and the `unpause` function should update the last interest accrual timestamp to the current time before unpausing. This ensures no interest is accrued for the period when the vault was paused.

Resolution: Resolved. The recommended fix was implemented in **a967883**.

3.5.2 Rounding Direction for Debt Values

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:*24*

Description:

The contract incorrectly uses `mulDivDown` for debt-related calculations where it should use `mulDivUp`. When calculating debt values, rounding should always be against the user to ensure the protocol remains fully collateralized. The current implementation rounds down in incorrectly in these six locations: 1, 2, 3, 4, 5, 6

This incorrect rounding could lead to users borrowing slightly more than they should be allowed to, potentially leaving the protocol under-collateralized.

Recommendation:

Replace the cited instances of `mulDivDown` with `mulDivUp`. This ensures that debt amounts are always rounded up, maintaining proper protocol collateralization. Keep `mulDivDown` for asset/collateral calculations as these should round against the user.

Resolution: Resolved. The recommended fix was implemented in **a967883**.

3.5.3 Missing Health Factor Check Post-Borrow

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:*500*

Description:

In the `borrow` function of the `LendingPool` contract, there is no check to re-evaluate the borrower's health factor after the borrow operation is completed. The current implementation only checks the health factor before the borrow.

Although the function's calculations aim to ensure an account remains solvent, the complexity of debt addition introduces potential edge cases where a borrower could become liquidatable after the transaction completes. A post-borrow health factor check would add an extra layer of security.

This issue also applies to:

- `_withdraw` function
- `disableAsset` function

Recommendation:

Add a post-operation health factor check in the `borrow`, `_withdraw`, and `disableAsset` functions. If the borrower's health factor falls below the required threshold, the transaction should be reverted to prevent unintended liquidations or bad debt situations.

Resolution: Resolved. The recommended fix was implemented in **a967883**.

3.5.4 Missing Admin Function for Bad Debt Liquidation

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:*631*

Description:

The `LendingPool` contract lacks an admin function to liquidate bad debt. Once a user accrues bad debt, standard liquidation processes cannot resolve it, leaving the lending pool with unresolved bad debt. This situation can negatively impact the pool's overall health and potentially require the use of reserves in an inefficient manner.

Recommendation:

Implement an admin-only function to liquidate bad debt. This function should allow the admin to repay the borrower's debt using the pool's reserves, ensuring that bad debt is effectively managed and the pool's financial health is maintained.

Resolution: Resolved. The recommended fix was implemented in **a967883**.

3.5.5 Incorrect Block Number Comparison

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:*937*

Description:

In the `totalBorrows` function, there is an incorrect block number comparison that prevents the early return from occurring. The code checks if `(blockDelta == block.number)` when it should be checking if `blockDelta == 0` to determine if the cached value is from the current block.

This comparison will only evaluate to true if `blockDelta` happens to equal the current block number, which is almost never the intended logic. As a result, the function will rarely return the cached value even when it should, causing unnecessary recalculations.

The impact is that the contract will not properly utilize its caching mechanism, leading to Increased gas costs from redundant calculations.

Recommendation:

Consider using timestamps instead of block numbers for more precise timing.

Or alternatively, Change the condition to check if the cached value is from the current block:

```
if (blockDelta == 0) return cachedTotalBorrows[asset];
```

Resolution: Resolved. The recommended fix was implemented in **98413b4**.

3.5.6 Exceeding Maximum GM Tokens in Collateral

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:*702*

Description:

In the `LendingPool` contract, users can exceed the maximum number of GM tokens allowed as collateral if the GM token list changes.

If the token list is updated to include new GM tokens, a user could end up with a combined total from both the old and new lists, exceeding the expected maximum. This can cause:

- Impossible liquidations: The loop iterating through collateral assets may run out of gas before processing all collaterals.
- Denial of Service (DoS): Over-leveraged positions cannot be liquidated, affecting protocol solvency.

Recommendation:

Implement a validation check in the `enableAsset` function to ensure users do not exceed the maximum number of GM tokens allowed as collateral when the GM token list is updated.

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.5.7 Missing Validation for Oracle Price

Severity: *Low risk (Resolved)*

Context: ***GMPriceOracle.sol***:*72-89*

Description:

The `getPrice` function in the `GMPriceOracle` contract does not validate the price returned by the oracle. While GMX has safeguards in place to prevent unrealistic price values, GM tokens can theoretically have a negative or zero price.

By design, GM tokens decrease in value as traders make a profit, since LPs are responsible for covering these payouts. If trader profits exceed the LPs' ability to pay, the price of the GM token could turn negative. In such cases, transactions will revert as expected since casting a negative value to **`uint256`** will fail.

However, if the price is exactly zero, it will be accepted, leading to downstream issues where a zero price is used in calculations. Although these instances are extremely rare, proper precautions should still be implemented to prevent unintended behavior.

Recommendation:

Implement a validation check in the `getUnderlyingPrice` and `_getPriceFromChainlink` functions to ensure that the price is non-zero before being used.

Resolution: Resolved. The recommended fix was implemented in **5939e4e**.

3.5.8 Unbounded GM Token Array Creates DoS Risk

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol***:227-242

Description:

The `LendingPool` contract maintains an array of GM tokens but lacks a maximum limit on the number of tokens that can be added. This array is iterated over in multiple functions throughout the contract. Without an upper bound, an excessive number of GM tokens could be added through `setGmTokens` and `addNewGmToken`, potentially causing functions to run out of gas when iterating over the array.

If the array grows too large, these critical functions could become unusable due to exceeding block gas limits, effectively causing a denial of service. This would prevent users from:

- Taking new loans
- Repaying existing loans
- Getting liquidated when unhealthy
- Withdrawing collateral

Recommendation:

Add a maximum limit for the number of GM tokens that can be supported:

```
function addNewGmToken(address token) external onlyOwner {
    require(gmTokens.length < MAX_GM_TOKENS, "Max GM tokens reached");
    //...
}

function setGmTokens(address[] calldata tokens) external onlyOwner {
    require(tokens.length <= MAX_GM_TOKENS, "Too many GM tokens");
    //...
}
```

Resolution: Resolved. The recommended fix was implemented in **0e87f01**.

3.5.9 Misleading Revert Message in Rewards Claim

Severity: *Low risk (Resolved)*

Context: ***GloopStaking.sol***:326

Description:

In the `GloopStaking` contract, the revert message `NoRewardsToClaim` in the `_claimRewards` function is misleading.

- The message is triggered when a user attempts to claim rewards but the input amounts for GLOOP and USDC rewards are both zero.
- This does not necessarily mean the user has no rewards available—it simply means they haven't specified any rewards to claim.

- The current message may confuse users into believing they have no rewards, when in reality, they just haven't selected any to claim.

Recommendation:

Update the revert message from `NoRewardsToClaim` to `NoRewardsBeingClaimed` to accurately reflect the condition being checked.

Resolution: Resolved. The recommended fix was implemented in **5939e4e**.

3.5.10 Misleading Comment in Debt Update Logic

Severity: *Low risk (Resolved)*

Context: ***LendingPool.sol**:537*

Description:

In the `_repay` function, the comment `// Add to the asset's total internal debt` is misleading. The logic following this comment actually decreases the `totalInternalDebt` for the asset, as it subtracts the `debtUnits` from `totalInternalDebt`.

This discrepancy between the comment and the actual code behavior can lead to confusion for developers and auditors, potentially causing misunderstandings about the function's logic and how debt is managed within the protocol.

Recommendation:

Update the comment to accurately reflect the operation being performed. Change it to:

```
// Subtract from the asset's total internal debt
```

This ensures clarity and prevents any misinterpretation of the code's functionality.

Resolution: Resolved. The recommended fix was implemented in **a967883**.

3.5.11 Centralization Risks

Severity: *Low risk (Resolved)*

Context: ***c6b2ea92d18a2d6a9efb95ea5abf947e3bda0c67***

Description:

The protocol is subject to centralization risks stemming from powers granted to the admin. These powers include the ability to update oracles, interest rate models, funding contracts with reward tokens, and modify the list of GM tokens.

Recommendation:

Document these centralization risks thoroughly to ensure transparency. This documentation should clearly outline the extent of admin powers and the potential impacts of their misuse.

Resolution: Resolved. The recommended fix was implemented in .