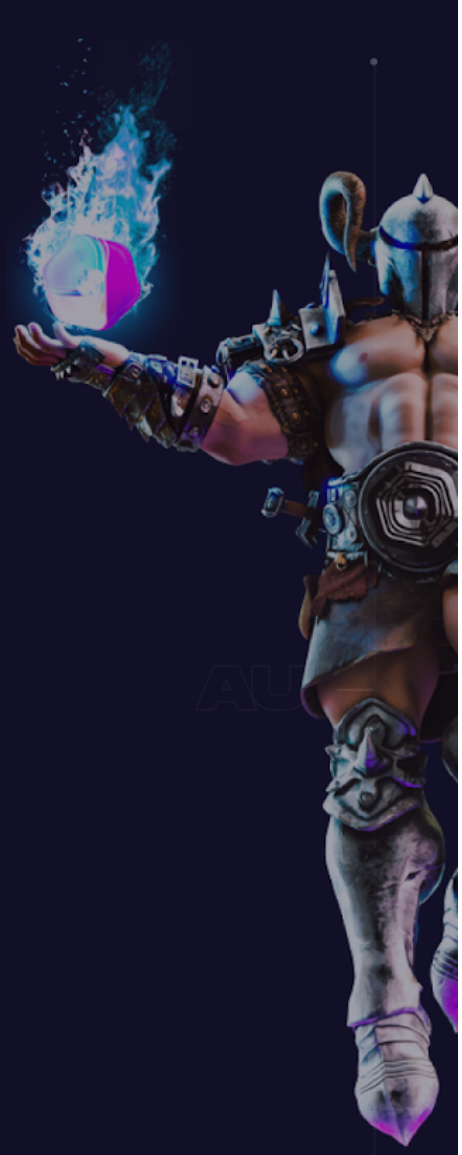
UnoRe

Reinsure Tech OU
Harju Maakond, Tallin Kesklinna
Linnaosa Torminae 7-26 10145, Estonia

Insured Audit: Code Review & Protocol Security Report



MAHALEND

Protocol
MahaLend

Date
31st May 2023

The UnoRe security research team has completed an initial time-boxed security review of a part of the **Mahalend** protocol and **LP Oracle** contract code, with a focus on the security aspects of the application's implementation.

Disclaimer

This report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all the vulnerabilities are fixed - upon the decision of the Customer.

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

Document Changelog:

24th April 2023	**Initial Pre-Triage Insured Audit Report**
20th May 2023	**Active Monitoring Notes**
31st May 2023	**Maha Post-Triage Insured Audit Report**

Technical Overview

MahaLend is a decentralized non-custodial liquidity protocol (meaning your funds are yours and yours alone) where users can either provide liquidity and receive interest, or borrow ARTH, and pay interest on their loan. MahaLend is a fork of AAVE V3. You can read more about the protocol in the full documentation here.

In scope for this audit we have the following contracts:

- MasterchefAToken.sol is responsible for managing and distributing rewards to users who provide liquidity to a certain liquidity pool on the platform.
- FeeBase.sol is responsible for the fees logic such as the rewardFeeDestination and rewardFeeRate.
- ChainlinkLPOracleGMU.sol is a Chainlink oracle for LP tokens

Threat Model

Roles & Actors

1. poolAdmin - sets rewardFeeDestination and rewardFeeRate via setRewardFeeRate and setRewardFeeAddress
2. pool - can mint and burn aTokens on behalf of users
3. rewardFeeDestination - receives the fees from the protocol

Internal Security QA

1. Q: What in the protocol has value in the market? The tokens held as collateral, as well as any interest accrued on the loan. Also the fees which are basically also tokens.
2. Q: What is the worst thing that can happen to the protocol? If the protocol is put into DoS state or locked/reward tokens are stolen.
3. Q: In what case can the protocol/users lose money? If an attacker is able to drain a pool or is able to claim the rewards of other users because of miscalculations.

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

Security Review Summary

Review commit hash LP oracle - af2b5e441449a942867647c41c9c59e2754399d9
Review commit hash Mahalend - 6e546fe20c78d4b7b93c33238080c9690e9bb4c5

Audit Scope

The following smart contracts were in scope of the audit:

- MasterchefAToken.sol - (Latest commit f29ac7c) - https://github.com/mahalend/contracts-core/blob/feat/aave-fork/contracts/protocol/tokenization/MasterchefAToken.sol
- ChainlinkLPOracleGMU.sol - (Latest commit c9a146f) - https://github.com/MahaDAO/gmu-oracle-contracts/blob/master/contracts/chainlink/ChainlinkLPOracleGMU.sol
- FeeBase.sol - (Latest commit 54e722d) - https://github.com/mahalend/contracts-core/blob/feat/aave-fork/contracts/misc/FeeBase.sol

The following number of issues were found, categorized by their severity:

- Critical & High: 2 issues
- Medium: 5 issues
- Low: 3 issues
- Informational: 9 issues

Note: The above summary of report findings at the Pre-Triage stage, most of these issues were addressed/fixed in consecutive stages.

Summary Table of Our Findings

ID	TITLE	Severity	Status
[C-01]	Oracle price manipulation attack can be executed on ChainlinkLPOracleGMU	Critical	Fixed
[H-01]	Insufficient Chainlink price feed validation	High	Fixed
[M-01]	feeRate validations are insufficient	Medium	Acknowledged
[M-02]	Fees are paid by the wrong contract	Medium	
[M-03]	Consider implementing timelock for onlyPoolAdmin functions	Medium	Acknowledged
[L-01]	Front-runnable initializer	Low	
[L-02]	Division before multiplication in _accumulatedRewardsForAmount	Low	Fixed
[L-03]	Change harvest() to internal	Low	Acknowledged
[I-01]	Use stable pragma statement	Informational	
[I-02]	Use custom errors instead of require statements	Informational	
[I-03]	Unused import	Informational	
[I-04]	SafeMath is not needed in the protocol	Informational	

[I-05]	Storage variable should be a constant	Informational	
[I-06]	Use OpenZeppelin's battle tested code instead of implementing it yourself	Informational	
[I-07]	Method should be inlined	Informational	
[I-08]	Redundant getter method	Informational	
[I-09]	Use concrete types in constructor directly instead of address and casting later	Informational	

Triage Fix Comments

[C-01] Oracle price manipulation attack can be executed on ChainlinkLPOracleGMU

https://github.com/MahaDAO/gmu-oracle-contracts/blob/7dc028732012cb55ca683bcedfb67987e860beac/contracts/lp-oracles/ChainlinkUniswapLPOracle.sol

[H-01] Insufficient Chainlink price feed validation

added validations for the price feed
https://github.com/MahaDAO/gmu-oracle-contracts/blob/master/contracts/lp-oracles/ChainlinkUniswapLPOracleWithSequencer.sol

[M-01] Insufficient Chainlink price feed validation

feeRate is set by governance

[M-02] feeRate validations are insufficient

The FeeBase is inherited by the aToken contract

[M-03] Consider implementing timelock for onlyPoolAdmin functions

By default pool admins are only the timelock contracts

[L-01] Front-runnable initializer

The initializer is called at the same time of the contract creation or update

[L-02] Division before multiplication in _accumulatedRewardsForAmount

https://github.com/mahalend/contracts-core/blob/master/contracts/protocol/tokenization/MasterchefAToken.sol#L177-L181

[L-03] Change harvest() to internal

Harvest can be public because it is meant to be called on a daily basis so that governance can collect revenue daily.

Centralization Risk Areas:

We have also identified several key areas within the protocol which contains centralization risks which needs to be made aware to the community and have highlighted them below:

1) MasterchefAToken
    a) setRewardFeeRate
    b) setRewardFeeAddress
2) IncentivizedERC20
    a) setIncentivesController

Active Monitoring and Realtime Tracking Scope:

1) AccessControl.sol
   (https://arbiscan.io/address/0xeE56fb1E3c274dE5F2a066C4A7A1fE7d5BEC07Ab#code)
   1) RoleRevorked
   2) RoleGranted
   3) RoleAdminChanged

2) BorrowLogic.sol (Pool | Address 0x8f8da3b85d854b1b4210e30c3118ca7e7b0ead70 | Arbiscan)
   1) FlashLoanLogic.sol
      1) Flashloan

2) LiquidationLogic.sol
    1) LiquidationCall
3) SupplyLogic.sol
    1) supply
    2) withdraw
4) Pool.sol
    1) updateBridgeProtocolFee
    2) proxy monitoring - monitors if the underlying contract changes

3) usdc/usdt atoken.sol
(https://arbiscan.io/address/0xdf69edd3a4807ff925d304dec185eb6ddf95c107#code)
    1) minting (above a certain token)
    2) proxy monitoring - monitors if the underlying contract changes

8) poolconfigurator.sol (PoolConfigurator | Address 0x363A1080535001993fD7058F334FaaE9Ed83D520 | Arbiscan) - monitoring key risk factors on the borrowing lending market in the protocol

1. setBorrowcap
2. setDebtCeiling
3. setReserveFactor
4. setReserveInterestRateStrategyAddress
5. setResereStableRateBorrowing
6. setSupplyCap
7. updateAToken

9) Arbitrum Sequencer(ArbitrumSequencerUptimeFeed | Address 0xC1303BBBaf172C55848D3Cb91606d8E27FF38428 | Arbiscan)

1. latestRoundData - real time monitoring of uptime sequencer feed to check if the Oracle price feeds are upto date in the arbitrum network

(https://docs.chain.link/data-feeds/l2-sequencer-feeds#arbitrum)

Initial Report Detailed Findings

[C-01] Oracle price manipulation attack can be executed on ChainlinkLPOracleGMU

Severity

Impact: High, as this is the most important feature of the contract that is used to calculate prices
Likelihood: High, as a common attack vector (flash loans) can be used to manipulate the oracle

Description

The formula that ChainlinkLPOracleGMU uses in fetchPrice goes through the _fetchPrice, tokenAGMUInLP and tokenBGMUInLP methods, where they have calls to lp.totalSupply and lp.getReserves. The problem is that the result of those functions is easily manipulatable by taking a flash loan and providing liquidity or buying/selling either of the pair's tokens. This means that any malicious user can easily manipulate the fetchPrice answer, which can have devastating consequences for the protocols that use the oracle.

Recommendations

Pricing LP tokens is a tough problem to solve, here is a solution: link

[H-01] Insufficient Chainlink price feed validation

Severity

Impact: High, as the protocol can operate with a stale price
Likelihood: Medium, as there are multiple ways that a problem happens

Description

In ChainlinkLPOracleGMU there are multiple problems with the Chainlink price feed input validation. The _getCurrentChainlinkResponse method calls the Chainlink price feed aggregator's latestRoundData method, but it does no input validation on it - the answer is not checked if it is actually a positive number and also the timestamp or updatedAt property is also not checked if it isn't too old. Another problem is the try-catch approach, where the cached ChainlinkResponse object has a success field that holds if the call to the aggregator failed or succeeded. This value is properly set, but not actually used anywhere, meaning in the other methods it is possible that the ChainlinkResponse objects that others operate with has a success = false property, but is still used as the response is valid.

- https://docs.chain.link/docs/historical-price-data/#historical-rounds
- https://docs.chain.link/docs/faq/#how-can-i-check-if-the-answer-to-a-round-is-being-carried-over-from-a-previous-round

The final issue here is that one of the chains the contract will be deployed on is the Arbitrum chain, where the Chainlink feeds have special properties. They operate with a sequencer, which mandatorily has to be checked if it is up and if it isn't the transaction should revert or set success = false, otherwise the protocol might operate with a stale/invalid price.

Recommendations

Instead of using try-catch just revert on failed external calls to the Chainlink aggregator, since there is no clean way to continue from a failed message. When it comes to the Arbitrum sequencer validation, follow the guide in Chainlink's L2 sequencer feeds docs. For the input of latestRoundData make sure to validate the input values returned from the call properly.

[M-01] feeRate validations are insufficient

Severity

Impact: High, as it can result in 100% rewards loss for users
Likelihood: Low, as it requires a malicious or compromised owner

Description

There are two problems with the feeRate property in FeeBase.

1. There are validations in _setRewardFeeRate that are missing in initializeFeeBase, so the feeRate can initially be set to any uint256 number
2. The pool admin can set the fee to 100% and steal all new rewards from users

Recommendations

Make sure to implement the same validations in initializeFeeBase that are in _setRewardFeeRate (you can remove the >= 0 validation, as it is always true for a uint256 value). Also make the max fee to be a smaller percentage, for example 10%.

[M-02] Fees are paid by the wrong contract

Severity

Impact: Low, as no funds are at risk
Likelihood: High, as the transaction may fail every time

Description

The burn method calls the harvest function which collects the rewards and is intended to pay the fees to the governance contract. This is done by calling the _chargeFee functions inside the FeeBase contract.

```
 function _chargeFee(IERC20 token, uint256 earnings) internal {
    ...

    if (feeToCharge > 0 && rewardFeeDestination != address(0)) {
      token.safeTransfer(rewardFeeDestination, feeToCharge);
      emit RewardFeeCharged(earnings, feeToCharge, rewardFeeDestination);
    }
```

The problem is that safeTransfer is used which means that the feeToCharge is send from the FeeBase contract which is not designed to store any tokens. Thus, the transaction will fail. The _chargeFee function is called by the MasterchefAToken contract which is the msg.sender in this case.

Recommendations

The fees are calculated on the earnings of the MasterchefAToken contract which collects the rewards. Make sure the correct contract is paying the fees.

```
  function _chargeFee(IERC20 token, uint256 earnings) internal {
    ...
    if (feeToCharge > 0 && rewardFeeDestination != address(0)) {
-       token.safeTransfer(rewardFeeDestination, feeToCharge);
+       token.safeTransferFrom(msg.sender, rewardFeeDestination, feeToCharge);
      emit RewardFeeCharged(earnings, feeToCharge, rewardFeeDestination);
    }

```

[M-03] Consider implementing timelock for onlyPoolAdmin functions

Severity

Impact: High, as it will result in monetary loss for the PoolAdmin and users
Likelihood: Low, because it requires a malicious or a compromised PoolAdmin

Description

In MasterchefAtoken.sol, the PoolAdmin can call setRewardFeeAddress which changes the address where the fees(rewards) are received. If there is a compromised owner or a malicious one he can change this address and there is no functionality that can stop this. Also the setRewardFeeRate function can be called and the fees can be changed to 100%. This will have a direct financial or trust impact on users who should be given an opportunity to react to them by exiting / engaging without being surprised when changes initiated by such functions are made.

Recommendations

A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate. Consider adding a timelock to both setRewardFeeAddress and setRewardFeeRate.

[L-01] Front-runnable initializer

The initialize method in MasterchefAToken is front-runnable. The impact is not a high one, since the admin's transaction to initialize will be reverted if it was front-runned, so it will be well-known that the contract should not be used. Still, an attacker can easily do this and create problems for the contract deployer/initializer. Add access control to the initialize method.

[L-02] Division before multiplication in _accumulatedRewardsForAmount

We have the following piece of code in _accumulatedRewardsForAmount:

```
    uint256 perc = (bal * 1e18) / (total);
    return (accRewards * perc) / (1e18);
```

We can see that when perc is calculated there is division, but then perc is used in multiplication on the next line. Change the code so it is just:

```
    return accRewards * bal / total;
```

This way there is no division before multiplication, which can result in a loss of precision.

[L-03] Change harvest() to internal

The harvest function is used to collect the rewards inside the contract and pay the fees to the governance. There is no point this to be a public function because anyone can call it which may not be desirable. Change it to internal or add onlyPool modifier.

[I-01] Use stable pragma statement

It's a best practice to lock the pragma statement in your contracts so you get deterministic bytecode compilation. Lock the pragma (use a stable one, removing the ^) in FeeBase and ChainlinkLPOracleGMU.

[I-02] Use custom errors instead of require statements

To improve the protocol's gas efficiency and interoperability, replace all require statements with revert statements with Solidity custom errors.

[I-03] Unused import

The FeeBase contract imports Ownable but does not inherit the contract or use it in any way. Remove the import statement.

[I-04] SafeMath is not needed in the protocol

Since the protocol is using a Solidity compiler version that is later than 0.8.0, this means that the compiler has built-in underflow & overflow checks. This means that the SafeMath library is redundant and should be removed from each contract as it wastes gas.

[I-05] Storage variable should be a constant

The pct100 variable in FeeBase can never be changed and its value is hardcoded in the contract anyway. Make the pct100 variable a private constant one and hardcode its value directly.

[I-06] Use OpenZeppelin's battle tested code instead of implementing it yourself

Instead of copy-pasting the code from OpenZeppelin's ReentrancyGuard in MasterchefAToken, just import the library and directly use it. Copy-pasting code is error-prone and a bad practice. Also, use IERC20Metadata from OpenZeppelin instead of the IERC20WithDeciamls interface, which also has a typo in its name, plus the decimals method should return uint8 instead of uint256.

[I-07] Method should be inlined

The _accumulatedRewardsFor method is only used in accumulatedRewardsFor so it should be inlined for simplicity.

[I-08] Redundant getter method

The getDecimalPercision method duplicates the TARGET_DIGITS getter (it is automatically generated since TARGET_DIGITS is a public constant). The best solution here is to make TARGET_DIGITS private instead of public.

[I-09] Use concrete types in constructor directly instead of address and casting later

The ChainlinkLPOracleGMU's constructor takes 4 address typed arguments and casts them each to an interface type in the constructor's body. Instead of doing those casts explicitly, just change the argument types from address to the interface type expected.