Moon Rabbit Defi Contracts Code Audit by Ambisafe Inc.
July, 2022

360 Pine Street, Suite 700, San Francisco, California 94104, USA

hello@ambisafe.com

1. **INTRODUCTION.** The Client requested Ambisafe to perform a code audit of the contracts implementing Moon Rabbit Defi 2.0. The contracts in question can be identified by the following git commit hash:

   `be82c969e60bf4c02e887c48eabfdcebd72f8dd5`

   The scope of the audit consists of 39 contracts, interfaces and libraries.

   After the initial code audit, Moon Rabbit DeFi applied a number of updates which can be identified by the following git commit hash:

   `af54cbf920406bd5b02d945c29eebd8df83cc3f5`

   Additional verification was performed after that.

2. **DISCLAIMER.** The code audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bugfree status. The code audit documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of Ambisafe.

3. **EXECUTIVE SUMMARY.** All the initially identified, minor and above, severity issues were **fixed** and are not present in the **final version** of the contracts. There are **no** known compiler bugs for the specified compiler version (0.8.3), that might affect the contracts' logic. There were 1 critical, 2 major, 4 minor, 25 informational and optimizational issues identified in the initial version of the contracts. The non-informational issues found in the contract were not present in the final version. They are described below for historical purposes.
   DSMath.rpow() and **MathHelper.mulDiv()** functions were compared with original implementations and confirmed to be equal.

4. **CRITICAL BUGS AND VULNERABILITIES.** One critical issue was identified that would allow a malicious actor to drain all the deposited liquidity from the protocol along with unlimited minting of the stable tokens. During the borrowing process of the native asset, a user address is called to transfer the funds, but user debt accounting happens only after that. In the initial version, this allowed a user to call the contract back and illegally withdraw their own liquidity making the debt uncollateralized. This critical issue has been fixed.

5. LINE BY LINE REVIEW. FIXED ISSUES.

5.1. DefiCore, line 84. Optimization, in the **updateCollateral()** function the **assetParameters** variable is read from storage twice.

5.2. DefiCore, line 94. Optimization, in the **updateCollateral()** function the **assetParameters.isAvailableAsCollateral(_assetKey)** is checked twice.

5.3. DefiCore, line 108. Note, the **updateCollateral()** function could emit an event.

5.4. DefiCore, line 133. Note, the **addLiquidity()** function allows a reentrancy before updating the user supply assets.

5.5. DefiCore, line 170. Note, the **withdrawLiquidity()** function allows a reentrancy before updating the user supply assets.

5.6. DefiCore, line 188. Note, the **approveToDelegateBorrow()** function could emit an event.

5.7. DefiCore, line 204. **Critical**, the **borrowFor()** function adds a borrowed asset to the user info after the assets are sent to the user. During this sending, a reentrancy is possible to take unlimited debt.

5.8. DefiCore, line 294. Note, in the **liquidation()** function revert reason, the word 'then' should be 'than'.

5.9. DefiCore, line 309. Note, in the **liquidation()** function revert reason, the word 'then' should be 'than'.

5.10. DefiCore, line 368. Minor, the **claimDistributionRewards()** function should use **safeTransfer()** when transferring the reward.

5.11. DefiCore, line 619. Note, in the **getAvailableLiquidity()** function, the local variable **_borrowedLimitInUSD** has a confusing name. Consider naming it **_borrowLimitInUSD**.

5.12. InterestRateLibrary, line 10. Optimization, the **InterestRateLibrary** could be redone to store all the values in the bytecode.

5.13. LiquidityPool, line 60. Optimization, the **lastLiquidity** pollutes storage, it would be cheaper to store a block and amount per user.

5.14. LiquidityPool, line 72. Optimization, the **withdrawLiquidity()** function does an excessive ‘enough liquidity’ check in case **_isMaxWithdraw** is true.

5.15. LiquidityPool, line 132. Minor, the **getAPY()** function should divide the interest by **getTotalLiquidity()**.

5.16. LiquidityPool, line 228. **Major**, the **_beforeTokenTransfer()** function doesn't verify if the transferred LP tokens were just added.

5.17. LiquidityPool, line 292. Minor, the **_ifNativePoolCheck()** function will silently lock **msg.value** sent with the transaction if the user has enough native token.

5.18. Registry, line 57. Minor, the **addProxyContract()** function should call a **setInjector(address(this))** on the newly deployed proxy to avoid setup interception.

5.19. RewardsDistribution, line 207. Optimization, the **_updateCumulativeSums()** function writes **LiquidityPoolInfo** to storage even if no values have changed.

5.20. UserInfoRegistry, line 378. Note, the **getMaxLiquidationQuantity()** could use **mulWithPrecision()**.

5.21. UserInfoRegistry, line 431. Optimization, the **_getUserAssets()** function could use **_userAssets.values()** instead of reading values manually.

5.22. AbstractPool, line 154. **Major**, the **repayBorrowFor()** function collects reserve funds from the same interest multiple times.

5.23. PureParameters, line 28. Optimization, the **Param** struct could instead encode all parameters as bytes32.

5.24. StablePermitToken, line 14. Optimization, the **registry** variable should be made **immutable**.

6. LINE BY LINE VERIFICATION. REMAINING AND ACKNOWLEDGED ISSUES.

6.1. CompoundRateKeeper, line 9. Optimization, the **CompoundRate** struct could be optimized to take a single storage slot.

6.2. DefiCore, line 414. Note, it would be helpful to add the **payerAddr** parameter to the **getMaxToRepay()** function.

6.3. LiquidityPool, line 52. Optimization, the **addLiquidity()** function excessively checks user balance.

6.4. LiquidityPool, line 83. Note, in the **withdrawLiquidity()** function, the **convertLPTokensToAsset(_userLPBalance) <= _liquidityAmount** case is always false.

6.5. Registry, line 158. Optimization, in the **getProxyUpgrader()** function, the **_proxyUpgrader** variable is read from storage twice.

6.6. ProxyUpgrader, line 42. Note, in the **getImplementation()** function could use **ImplementationGetter(what).implementation()**.

6.7. DecimalsConverter, line 20. Note, the **convert()** function will round down the result if the **destination** is smaller than the **base**.

6.8. DSMath, line 8. Note, misleading comment. It calculates ((x/b)^n)*b.

Ambisafe Inc.

360 Pine Street, Suite 700, San Francisco, California 94104, USA

hello@ambisafe.com