arbitrary./execution

MilkomedaM1

MILKOMEDA SECURITY ASSESSMENT

October 14, 2022

Prepared For:
Nico Arqueros
Prepared By:
Chris Masden, Jasper Clark

Changelog:

September 29, 2022	*Initial report delivered*
October 14, 2022	*Final report delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's assessment of the Milkomeda Canonical Bridge Asset system. The system allows for standardization of token transfer interoperability between different chains and bridges.

Two Arbitrary Execution engineers conducted this review over a 1-week period, from September 19, 2022 to September 23, 2022. The audited branch was `main` (commit hash `450ae7a3081fcbc781f4ce4637700013fbbb7b31`) in the `dcSpark/milkomeda-token-merger` repository. The Solidity files in scope for this audit included all contracts in the `contracts/` directory with the exception of contracts in the `dev/` directory. The complete list of files is in Appendix B.

A detailed manual review of the codebase was performed with a focus on protocol-specific security issues. Static analysis was also performed using Slither.

The assessment resulted in findings ranging in severity from medium to note (informational). The finding with a severity level of medium identified a scenario where it's possible to unintentionally lock funds in the contract without the ability to withdraw them. The note findings contain observations regarding coding best practices and opportunities for gas optimizations.

FIX REVIEW UPDATE

The Milkomeda team has fixed all major issues identified in the engagement. The full breakdown of fixes can be found in the Fixes Summary section.

FIX REVIEW PROCESS

After receiving fixes for the findings shared with Milkomeda, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the *Update* section for a finding.

OBSERVATIONS

The core logic of the protocol is in the `TokenMerger` contract. The other contracts crucial to the system involve the `CanonicalToken`, an ERC20 token built on top of the well established OpenZeppelin ERC20 library, and an ERC1967 proxy contract that is also built on top of OpenZeppelin libraries. All the contracts in scope were straightforward to understand and had logically grouped functions that made understanding the different components in the contracts easy to understand.

There are three undesirable issues the Milkomeda team is attempting to address with this system: * The fragmentation of liquidity by having various bridge assets can lead to a greater amount of slippage when performing a token swap. This would result in the user paying a higher fee. * Having multiple assets can be very confusing to users and leads to a bad user experience. * The protocol is meant to lower risk to a user when using a bridge. The way it's able to do this is by spreading the risk across multiple bridges

instead of picking the “best” bridge as it’s very difficult if not impossible to determine which is the best bridge.

A commission is collected by the protocol and can help cover lost funds in the event of a bridge exploit that results in stolen funds.

SYSTEM OVERVIEW

CANONICAL TOKEN

A standard ERC20 token built using OpenZeppelin libraries. A public `mint` function was added with an `onlyOwner` function modifier that only lets the owner of the `TokenMerger` proxy mint tokens via the `mintCanonicalToken` function.

ERC1967 PROXY

ERC1967 compliant proxy that uses OpenZeppelin libraries. All function calls to the proxy get delegated to the implementation address with the exception of the `transferOwnership`, `getAdmin`, `upgradeTo`, and `getImplementation` functions.

TOKEN CONVERSION

This component of the system is focused around the conversion of fragmented tokens to canonical tokens and vice versa. Canonical tokens are the tokens that are minted by the protocol. Fragmented tokens are the tokens issued by individual bridges.

VULNERABILITY STATISTICS

Severity	Count
Critical	0
High	0
Medium	1
Low	1
Note	4

FIXES SUMMARY

Finding	Severity	Status
M01	Medium	Fixed in pull request #10
L01	Low	Fixed in pull request #14
N01	Note	Fixed in pull request #11
N02	Note	Fixed in pull request #12
N03	Note	Fixed in pull request #15
N04	Note	Fixed in pull request #13

FINDINGS

MEDIUM SEVERITY

[M01] UNCHECKED TRANSFER

The `withdrawCommission` function performs an ERC20 token transfer to collect the fees accumulated from users of the protocol. If an ERC20 token fails to transfer funds from one address to another, there are two common outcomes. The first is that an exception will be raised and will result in a reverted transaction. The second is that the `boolean` return value from `transfer` will return `false`. In `withdrawCommission`, the return value from the `transfer` is not checked and can lead to a specified amount of tokens being subtracted from the `commissionTreasury` mapping, even if a transfer fails. This will result in locked funds that will be unable to be withdrawn from the protocol.

RECOMMENDATION

Consider checking the return value of the `transfer` function and raise an exception if `false` is returned.

UPDATE

Fixed in pull request #10 (commit hash `7f845bc0a1ead0ec6df9f58417c69e22410fa87b`), as recommended.

LOW SEVERITY

[L01] ACTIVE TOKEN PAIRS CAN BECOME INACTIVE

The `blocksToWait` storage variable is used in the `initialDelayPassed` modifier to determine when a pair of fragmented and canonical tokens can be converted. A limitation of the current implementation is that `blocksToWait` is used globally and effects all fragmented and canonical pairs. This can cause already active pairs to become inactive and would prevent users of the protocol from converting between the token pairs until after a specified numbers of blocks has occurred. This can result in a negative user experience as this behavior is unexpected.

RECOMMENDATION

Consider storing the height at which the conversion protocol is active on each `FragmentedToken`.

UPDATE

Fixed in pull request #14 (commit hash `7f52e4dd34c46f84c6b314ce92668f817643cdbc`), as recommended.

NOTE SEVERITY

[N01] `COMMISSIONDECIMALS` CAN BE SET INCORRECTLY

The `commissionDecimals` storageVariable is used in conjunction with `commissionPercentage` to calculate the commission. The current approach can lead to an invalid combination of `commissionDecimals` and `commissionPercentage` values. Additionally, the code can be simplified by choosing a fixed decimal value.

RECOMMENDATION

Consider making `commissionDecimals` a `constant` variable with a value of `1e18` and changing the type of the `commissionPercentage` variable to `uint256`. This is a common practice that will allow for easier calculations when determining the amount of commission to collect on a token conversion.

UPDATE

Fixed in pull request #11 (commit hash `942b67133c2af8a03efff4bdd42322257022f8b5`), as recommended.

[N02] USE OF FLOATING COMPILER VERSION PRAGMA

All contracts in this audit float their Solidity compiler versions (e.g. `pragma solidity >=0.8.4`). Locking the compiler version prevents accidentally deploying the contracts with a different version than what was used for testing. It is best practice to deploy contracts with the same compiler version that is used during testing and development.

RECOMMENDATION

Consider locking the compiler pragma to the specific version of the Solidity compiler used during testing and development.

UPDATE

Fixed in pull request #12 (commit hash `a6492f3c09db8f6112cb0c04ec375d9f409069ca`), as recommended.

[N03] GAS SAVINGS

Within the code base, several opportunities for gas optimization have been identified: * Line 242 in the `withdrawCommission` function contains an unnecessary require check as the compiler specified in `hardhat.config.ts` (`0.8.15`) will insert underflow guards around the subtraction on line 244. * The `onlyAllowedPair` function modifier can be made more efficient by using a storage pointer. * Solidity compiler optimization is not enabled. * Line 170 in the `removeFragmentedToken` function clears a map entry by writing `0's` to the location of the `ConversionProtocol` in storage instead of using the `delete` keyword. * The `fragmentedToken` member of the `ConversionProtocol` `struct` is being used to determine if a `ConversionProtocol` struct has been initialized. However, this can be determined by checking to see if the `conversionProtocol` mapping struct has a value of 0.

RECOMMENDATION

Consider making the suggested changes to reduce gas usage.

UPDATE

Fixed in pull request #15 (commit hash `5ad7ed17a8e9dcd61be9eb7c2f184c453b94bc61`), as recommended.

[N04] REENTRANT FUNCTIONS

The `convertToCanonical` function and the `convertToFragmented` function do not follow the checks-effects-interactions pattern that is recommended for functions that can be reentered. The `_addToTreasury` internal function call happens after both ERC20 external function calls. The potential consequence of a malicious actor exploiting this behavior is minimal due to both functions transferring tokens from `msg.sender` to the contract before transferring the matching token to the specified receiver. At worst, this would cause the contract to hold additional tokens that have not be added to the `commissionTreasury` and would not allow withdrawing of the additional tokens.

RECOMMENDATION

Consider following the checks-effects-interactions pattern by calling `_addToTreasury` before the external function calls.

UPDATE

Fixed in pull request #13 (commit hash `c5eccc173aeefa08d75a13a812c35c13b7c4d29c`), as recommended.

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	Definition
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
CanonicalToken.sol
ERC1967Ownable.sol
FragmentedToken.sol
TokenMerger.sol
TokenMergerProxy.sol
```