CYBERSEC CONTRACT AUDIT REPORT
Mith Cash - CTDSEC.COM

Introduction

During January of 2021, Mith Cash engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. Mith Cash provided CTDSec with access to their code repository and whitepaper.

Disclaimer

The 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 to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

I always recommend having a bug bounty program opened to detect future bugs.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Mith Cash contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Boardroom.sol
Bond.sol
Cash.sol
Distributor.sol
Migrations.sol
Oracle.sol
Share.sol
SimpleERCFund.sol
Timelock.sol
Treasure.sol
Voteproxy.sol

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

- Correctness of the protocol implementation [Result OK]
- User funds are secure on the blockchain and cannot be transferred without user permission [Result OK]
- Vulnerabilities within each component as well as secure interaction between the network components [Result OK]
- Correctly passing requests to the network core [Result OK]
- Data privacy, data leaking, and information integrity [Result OK]
- Susceptible to reentrancy attack [Result OK]
- Key management implementation: secure private key storage and proper management of encryption and signing keys [Result OK]
- Handling large volumes of network traffic [Result OK]
- Resistance to DDoS and similar attacks [Result OK]
- Aligning incentives with the rest of the network [Result OK]
- Any attack that impacts funds, such as draining or manipulating of funds [Result OK]
- Mismanagement of funds via transactions [Result OK]
- Inappropriate permissions and excess authority [Result OK]
- Special token issuance model [Result OK]

Vulnerabilities

ISSUES

MEDIUM SWC-000

Function could be marked as external.

The function definition of "owner" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.In public functions, Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Memory allocation is expensive, whereas reading from calldata is cheap.

Locations

```
108     // oracle
109     function getBondOraclePrice() public view returns (uint256) {
110         return _getCashPrice(bondOracle);
111     }

113     function getSeigniorageOraclePrice() public view returns (uint256) {
114         return _getCashPrice(seigniorageOracle);
115     }

167     function setFundAllocationRate(uint256 rate) public onlyOperator {
168         fundAllocationRate = rate;
169         emit ContributionPoolRateChanged(msg.sender, rate);
170     }
```

LOW SWC-115

Use of "tx.origin" as a part of authorization control.

Using "tx.origin" as a security control can lead to authorization bypass vulnerabilities. Consider using "msg.sender".

Tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.

Locations - ContractGuard.sol

```
7          return _status[block.number][tx.origin];
26         _status[block.number][tx.origin] = true;
```

LOW SWC-120

Potential use of "block.number" as source of randomness.

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. We recommend using a commitment scheme for example RANDAO or using external sources of randomness via oracles for example Oraclize.

Locations - Boardroom.sol

```
79          BoardSnapshot memory genesisSnapshot = BoardSnapshot({
80              time: block.number,
81              rewardReceived: 0,
82              rewardPerShare: 0
83          });
```

```
204          BoardSnapshot memory newSnapshot = BoardSnapshot({
205              time: block.number,
206              rewardReceived: amount,
207              rewardPerShare: nextRPS
208          });
```

LOW - SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.6.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Locations - Boardroom.sol

```
1  pragma solidity ^0.6.0;
2  //pragma experimental ABIEncoderV2;
3
```

```
1 pragma solidity >=0.6.2;
2
3 // Limit Order version 1.0
```

Summary of the Audit

During the audit the contract was manually reviewed and analyzed with static analysis tools. CTDSEC team have found some low to lowest security issues and the audit report contains all necessary information related to them. Because the vulnerabilities found are low level and medium one only affect gas consumption, the code is considered secure and no major fixes are required.