MahaDAO

Smart Contract Audit Report

StakeRewards.sol

May 02, 2021

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Introduction

1. About MahaDAO

ARTH is a new type of currency designed to not be pegged to government-owned currencies (like US Dollar, Euro, or Chinese Yuan), but still remain relatively stable (unlike Gold and Bitcoin).

Without being influenced by government-owned currencies, ARTH will be immune to inflation. Through stability, ARTH also becomes a superior choice of currency for means of trade. This is unlike Gold or Bitcoin, which are used more as a store of value rather than a medium of exchange.

Visit http://mahadao.com/ to learn more about.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The MahaDAO team has provided documentation for the purpose of conducting the audit. The documents are:

1. https://docs.arthcoin.com/

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: **ARTH v2**
- Contract Name: StakeRewards.sol
- Languages: Solidity(Smart contract)
- Github commit hash for audit:**8bcf83f8d6a3d5675d400ec63acbf079ba638bed**
- GitHub link: https://github.com/MahaDAO/arthcoin-v2/blob/master/contracts/Staking/StakingRewards.sol
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

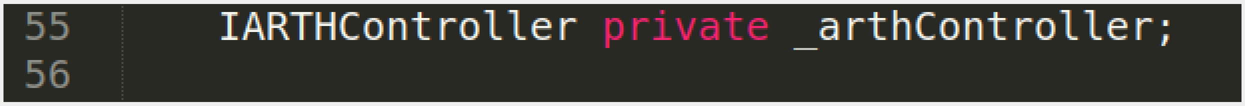
Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	1	2	6
Closed	-	-	-

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

High severity issues

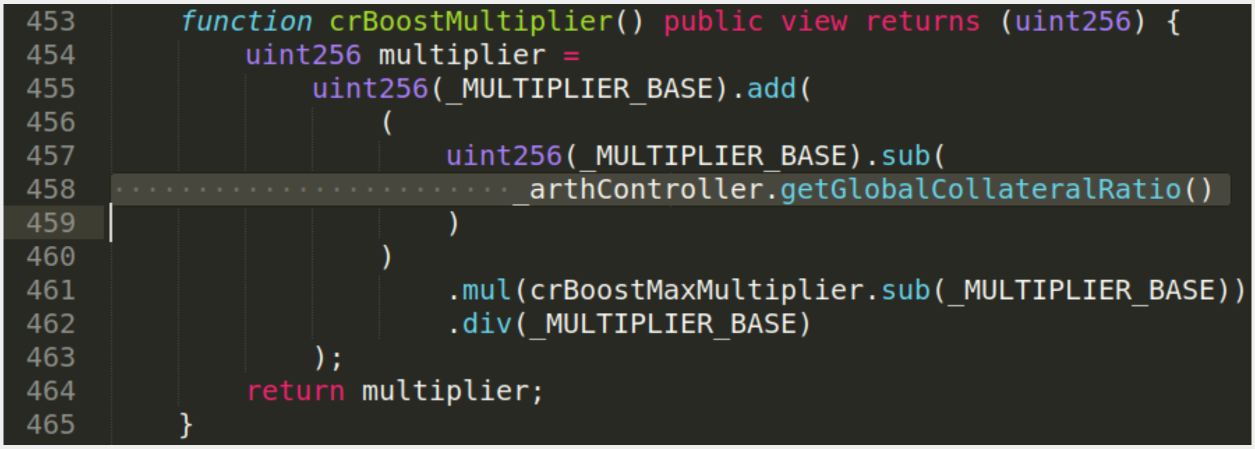
1. **_arthConroller is never initialized**
   Line no - 56, 458
   Description:
   The **_arthController** state variable is never initialized throughout the contract.



```
55        IARTHController private _arthController;
56
```

However, it is used in the **crBoostMultiplier** function at line **458** to call the **getGlobalCollateralRatio** function.

Since the **_arthController** is never initialized, it will lead to an unexpected scenario that will adversely affect the intended behaviour of the function.



```
453     function crBoostMultiplier() public view returns (uint256) {
454         uint256 multiplier =
455             uint256(_MULTIPLIER_BASE).add(
456                 (
457                     uint256(_MULTIPLIER_BASE).sub(
458                         _arthController.getGlobalCollateralRatio()
459                     )
460                 )
461                     .mul(crBoostMaxMultiplier.sub(_MULTIPLIER_BASE))
462                     .div(_MULTIPLIER_BASE)
463             );
464         return multiplier;
465     }
```

Recommendation:
_arthController must be initialized adequately before being used in a particular function.

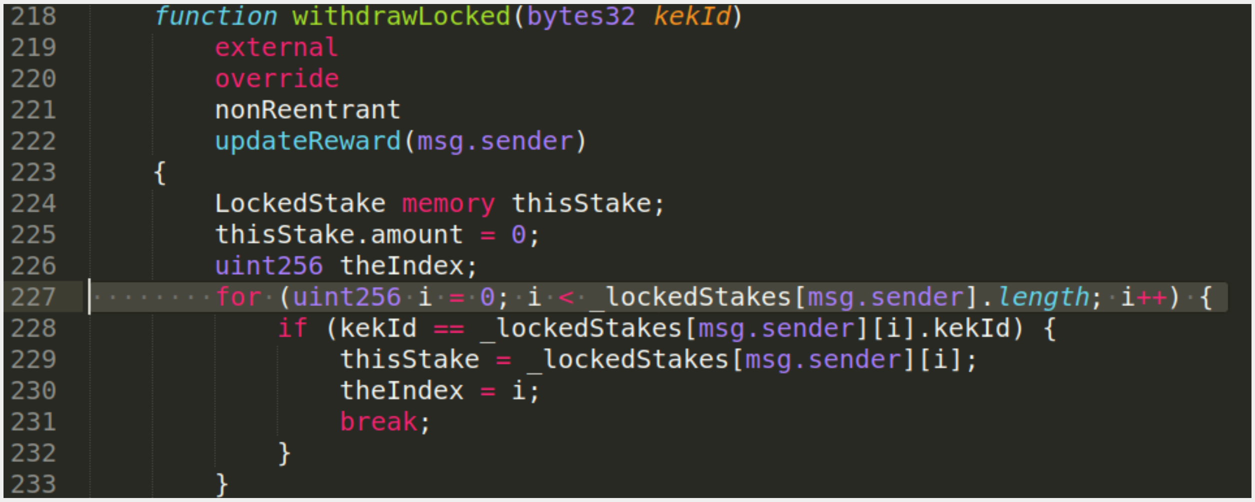
Medium severity issues

1. **State Variables Updated After External Call. Violation of Check_Effects_Interaction Pattern**
   Line no -524-539, 576-593,497-498, 277-279,208-209, 263-265
   Description:
   As per the Check_Effects_Interaction Pattern in Solidity, external calls should be made at the very end of the function. Event emission as well as any state variable modification must be done before the external call is made.

However, during the automated testing, it was found that some of the functions in the StakingRewards contract violate this **Check-Effects-Interaction** pattern at the above-mentioned lines.

Recommendation:
Modification of any State Variables must be performed before making an external call. Check Effects Interaction Pattern must be followed while implementing external calls in a function.

2. **for Loop in withdrawLocked function is extremely costly**
Line no - 227
Description:
The **for loop** in the **withdrawLocked function** includes state variables like **.length** of a non-memory array in the condition of the for loops.



```
218     function withdrawLocked(bytes32 kekId)
219         external
220         override
221         nonReentrant
222         updateReward(msg.sender)
223     {
224         LockedStake memory thisStake;
225         thisStake.amount = 0;
226         uint256 theIndex;
227         for (uint256 i = 0; i < _lockedStakes[msg.sender].length; i++) {
228             if (kekId == _lockedStakes[msg.sender][i].kekId) {
229                 thisStake = _lockedStakes[msg.sender][i];
230                 theIndex = i;
231                 break;
232             }
233         }
```

As a result, these state variables consume a lot more extra gas for every iteration of the loop.

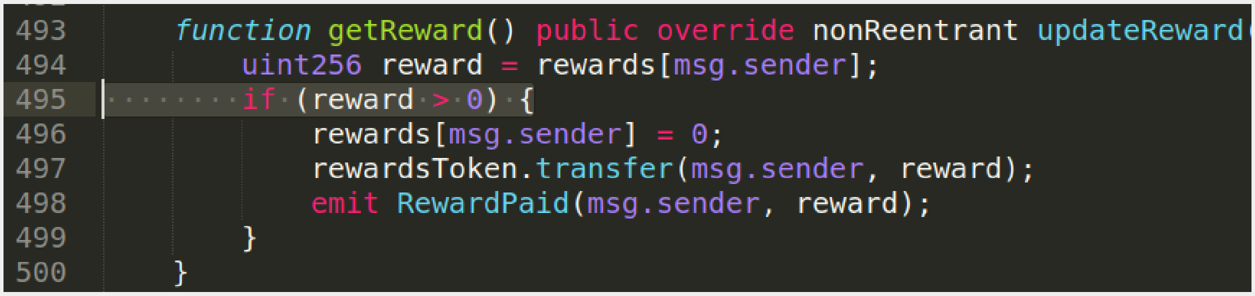
Recommendation:
It's quite effective to use a local variable instead of a state variable like **.length** in a loop. For instance,

```
local_variable = _lockedStakes[msg.sender].length;
 for (uint256 i = 0; i < local_variable; i++) {
        if (kekId == _lockedStakes[msg.sender][i].kekId) {
           thisStake = _lockedStakes[msg.sender][i];
           theIndex = i;
           break;
        }
    }
```

Low severity issues

1. **getReward function should include require statement instead of IF-Else Statement**
   Line no: 495-499
   Description:
   The **getReward** function includes an **if statement** at the very beginning of the function to check whether or not the **reward amount** for a particular user is more than Zero. Most importantly, this is a strict check and the function body is only executed if this **IF statement** holds true.

   In Solidity, in order to check for such strict validations in a function, **require statements** are considered more preferable and effective. While it helps in gas optimizations it also enhances the readability of the code.



```
493     function getReward() public override nonReentrant updateReward
494         uint256 reward = rewards[msg.sender];
495         if (reward > 0) {
496             rewards[msg.sender] = 0;
497             rewardsToken.transfer(msg.sender, reward);
498             emit RewardPaid(msg.sender, reward);
499         }
500     }
```

   Recommendation:
   Use **require statement** instead of **IF statement** in the above-mentioned function line. For instance,
   require(reward > 0,"Error MSG:Reward Amount for this address is ZERO");

2. **External Visibility should be preferred**
   Description:
   Those functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.
   Therefore, the following function must be marked as **external** within the contract:
   - **lockedBalanceOf**
   - **getReward**

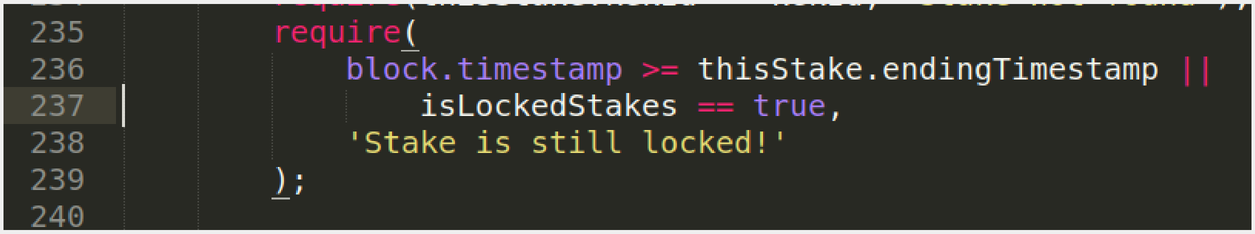
   Recommendation:
   If the public visibility of these functions is not intended, the visibility keyword must be modified to external.

3. **Comparison to boolean Constant**

   Line no: 237, 521, 549

   Description:

   Boolean constants can directly be used in conditional statements or require statements. Therefore, it's not considered a better practise to explicitly use **TRUE or FALSE** in the **require** statements.



```
235         require(
236             block.timestamp >= thisStake.endingTimestamp ||
237                 isLockedStakes == true,
238             'Stake is still locked!'
239         );
240
```

   Recommendation:

   The equality to boolean constants must be removed from the above-mentioned line.

4. **Return Value of an External Call is never used Effectively**

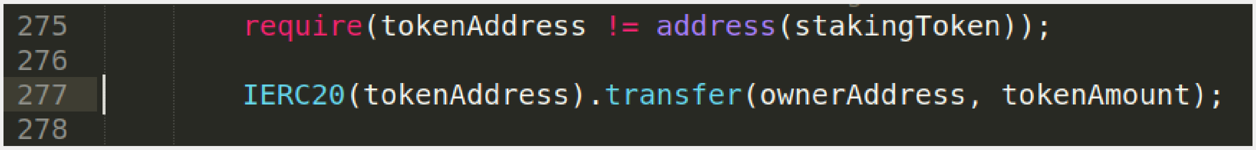
   Line no - 277, 497

   Description:

   The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.

   These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.

   However, the StakingRewards contract never uses these return values throughout the contract.



```
275         require(tokenAddress != address(stakingToken));
276
277         IERC20(tokenAddress).transfer(ownerAddress, tokenAmount);
278
```

   Recommendation:

   Effective use of all the return values from external calls must be ensured within the contract.

5. **No Events emitted after imperative State Variable modification**

   Line no - 356, 360

   Description:

   Functions that update an imperative arithmetic state variable contract should emit an event after the updation.

The following functions modify some crucial arithmetic parameters like **ownerAddress**, **timelockAddress**, **rewardRate** etc, in the StakingReward contract but do not emit an event after that:

- **setOwnerAndTimelock**
- **setRewardRate**

Since there is no event emitted on updating this variable, it might be difficult to track it off-chain.

Recommendation:

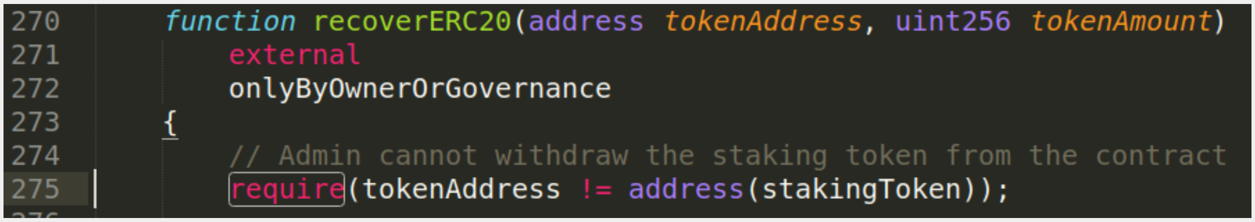
An event should be fired after updating the rewardRate variable.

6. Absence of Error messages in Require Statements

Line no - 275

Description:

The **recoverERC20** includes a **require** statement in the StakingRewards.sol contract that does not include an error message.



```
270     function recoverERC20(address tokenAddress, uint256 tokenAmount)
271         external
272         onlyByOwnerOrGovernance
273     {
274         // Admin cannot withdraw the staking token from the contract
275         require(tokenAddress != address(stakingToken));
```

While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:

Error Messages must be included in every require statement in the contract

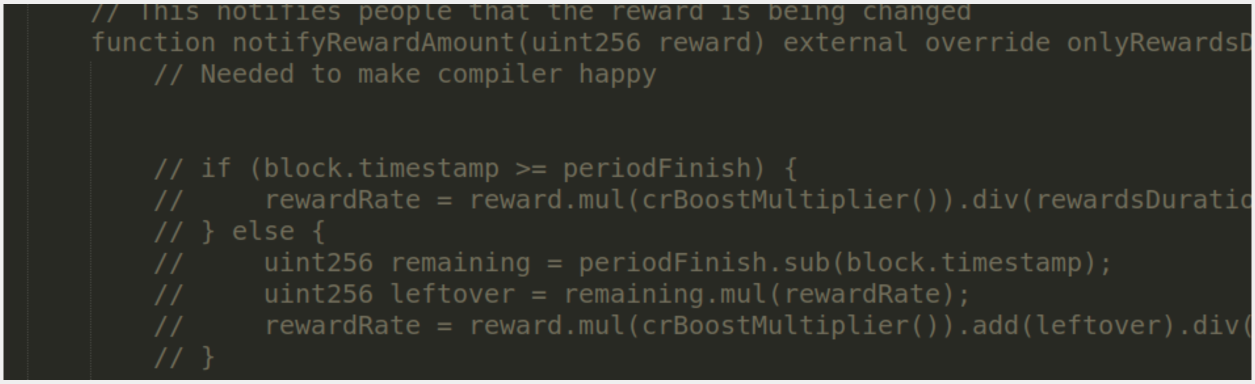
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Recommendations

1. **NatSpec Annotations must be included**
   Description:
   A smart contract does not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

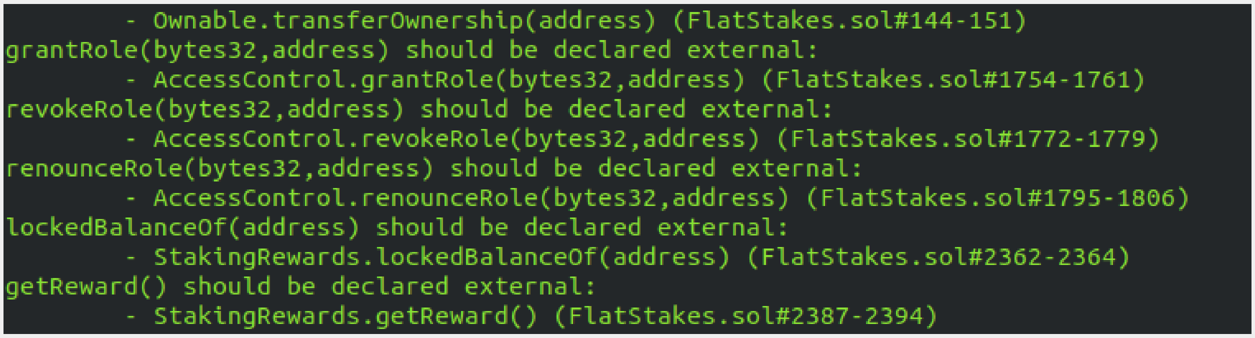
2. **Commented codes must be wiped out before deployment**
   Description:
   The StakingReward.sol contract includes quite a few commented codes at the end of the contract.
   This badly affects the readability of the code.



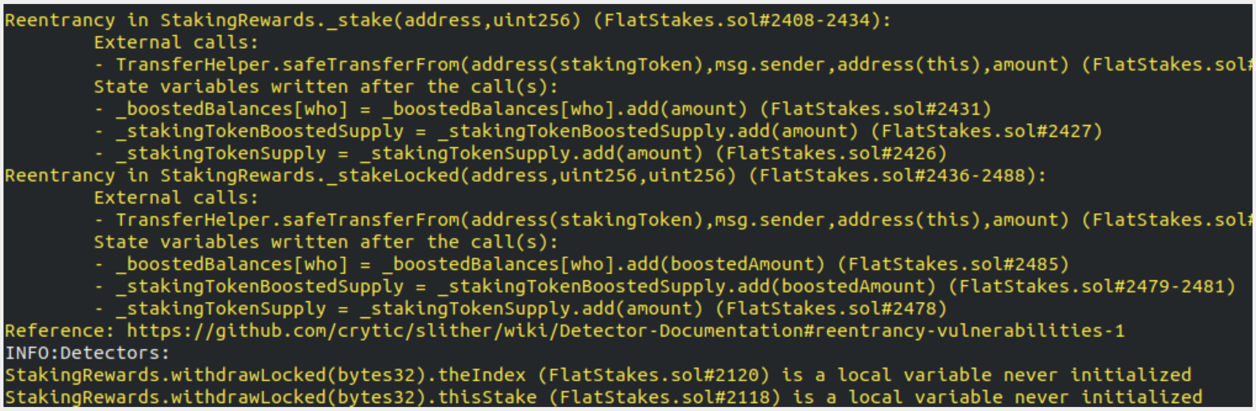
```
    // This notifies people that the reward is being changed
    function notifyRewardAmount(uint256 reward) external override onlyRewardsD
        // Needed to make compiler happy


        // if (block.timestamp >= periodFinish) {
        //     rewardRate = reward.mul(crBoostMultiplier()).div(rewardsDuratic
        // } else {
        //     uint256 remaining = periodFinish.sub(block.timestamp);
        //     uint256 leftover = remaining.mul(rewardRate);
        //     rewardRate = reward.mul(crBoostMultiplier()).add(leftover).div(
        // }
```

   Recommendation:
   If these instances of code are not required in the current version of the contract, then the commented codes must be removed before deployment.

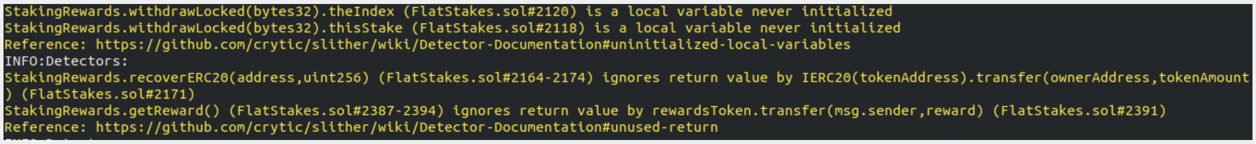
Automated Test Result



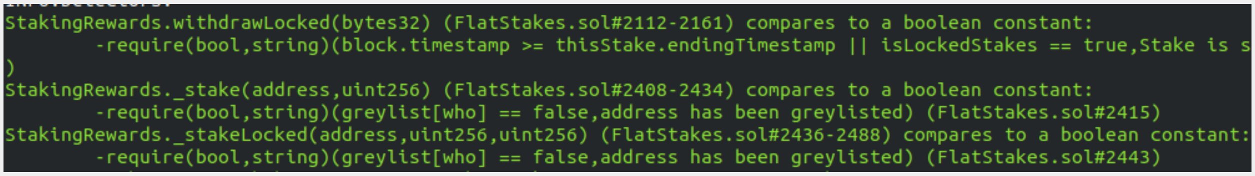
```
        - Ownable.transferOwnership(address) (FlatStakes.sol#144-151)
grantRole(bytes32,address) should be declared external:
        - AccessControl.grantRole(bytes32,address) (FlatStakes.sol#1754-1761)
revokeRole(bytes32,address) should be declared external:
        - AccessControl.revokeRole(bytes32,address) (FlatStakes.sol#1772-1779)
renounceRole(bytes32,address) should be declared external:
        - AccessControl.renounceRole(bytes32,address) (FlatStakes.sol#1795-1806)
lockedBalanceOf(address) should be declared external:
        - StakingRewards.lockedBalanceOf(address) (FlatStakes.sol#2362-2364)
getReward() should be declared external:
        - StakingRewards.getReward() (FlatStakes.sol#2387-2394)
```



```
Reentrancy in StakingRewards._stake(address,uint256) (FlatStakes.sol#2408-2434):
        External calls:
        - TransferHelper.safeTransferFrom(address(stakingToken),msg.sender,address(this),amount) (FlatStakes.sol#
        State variables written after the call(s):
        - _boostedBalances[who] = _boostedBalances[who].add(amount) (FlatStakes.sol#2431)
        - _stakingTokenBoostedSupply = _stakingTokenBoostedSupply.add(amount) (FlatStakes.sol#2427)
        - _stakingTokenSupply = _stakingTokenSupply.add(amount) (FlatStakes.sol#2426)
Reentrancy in StakingRewards._stakeLocked(address,uint256,uint256) (FlatStakes.sol#2436-2488):
        External calls:
        - TransferHelper.safeTransferFrom(address(stakingToken),msg.sender,address(this),amount) (FlatStakes.sol#
        State variables written after the call(s):
        - _boostedBalances[who] = _boostedBalances[who].add(boostedAmount) (FlatStakes.sol#2485)
        - _stakingTokenBoostedSupply = _stakingTokenBoostedSupply.add(boostedAmount) (FlatStakes.sol#2479-2481)
        - _stakingTokenSupply = _stakingTokenSupply.add(amount) (FlatStakes.sol#2478)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
StakingRewards.withdrawLocked(bytes32).theIndex (FlatStakes.sol#2120) is a local variable never initialized
StakingRewards.withdrawLocked(bytes32).thisStake (FlatStakes.sol#2118) is a local variable never initialized
```



```
StakingRewards.withdrawLocked(bytes32).theIndex (FlatStakes.sol#2120) is a local variable never initialized
StakingRewards.withdrawLocked(bytes32).thisStake (FlatStakes.sol#2118) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
StakingRewards.recoverERC20(address,uint256) (FlatStakes.sol#2164-2174) ignores return value by IERC20(tokenAddress).transfer(ownerAddress,tokenAmount
) (FlatStakes.sol#2171)
StakingRewards.getReward() (FlatStakes.sol#2387-2394) ignores return value by rewardsToken.transfer(msg.sender,reward) (FlatStakes.sol#2391)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```



```
StakingRewards.withdrawLocked(bytes32) (FlatStakes.sol#2112-2161) compares to a boolean constant:
        -require(bool,string)(block.timestamp >= thisStake.endingTimestamp || isLockedStakes == true,Stake is s
)
StakingRewards._stake(address,uint256) (FlatStakes.sol#2408-2434) compares to a boolean constant:
        -require(bool,string)(greylist[who] == false,address has been greylisted) (FlatStakes.sol#2415)
StakingRewards._stakeLocked(address,uint256,uint256) (FlatStakes.sol#2436-2488) compares to a boolean constant:
        -require(bool,string)(greylist[who] == false,address has been greylisted) (FlatStakes.sol#2443)
```

Concluding Remarks

While conducting the audits of MahaDAO smart contract - StakeRewards.sol, it was observed that the contracts contain High, Medium and Low severity issues, along with a few areas of recommendations.

Our auditors suggest that Low severity issues should be resolved by MahaDAO developers. Resolving the areas of recommendations are up to the team's discretion. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the MahaDAO platform or its product neither this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes Pvt Ltd.