QuillAudits

# Audit Report
# March, 2022

For

## RuufPay

# Contents

# Overview

## RuufPay

RuufPay has launched its own web staking application for users to earn rewards for being RuufCoin supporters. Users can link up decentralized wallets such as Metamask to the RuufPay Web Staking protocol to begin earning rewards today.

## Scope of the Audit

The scope of this audit was to analyze RuufPay- RuufStakeFarm smart contract codebase for quality, security, and correctness.

**RuufPay Contract:**
https://github.com/RuufPay/smartcontracts/blob/main/contracts/RuufStakeFarm.sol

**Branch:** main

**Commit:** 2c3357a6f8d787202c3bdfbed52ce53c3488fc8e

**Fixed In:** 389b2eaa8a51cf9181827d03570bff0f88045231

## Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

## Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

**Structural Analysis**
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

**Static Analysis**
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

**Code Review / Manual Analysis**
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

**Gas Consumption**
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

**Tools and Platforms used for Audit**
Mythril, Slither, SmartCheck, Surya, Solhint.

# Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

| Risk-level | Description |
| --- | --- |
| **High** | A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment. |
| **Medium** | The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed. |
| **Low** | Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future. |
| **Informational** | These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact. |

## Number of issues per severity

| Type | High | Medium | Low | Informational |
| --- | --- | --- | --- | --- |
| Open | 0 | 0 | 0 | 0 |
| Acknowledged | 0 | 0 | 0 | 0 |
| Closed | 0 | 1 | 5 | 3 |

# Issues Found – Code Review/Manual Testing

## High severity issues

No issues found

## Medium severity issues

1. **Centralization issues**

    1. Owner can change the interest rate at any time with [#L61] changeIr() and can prevent everyone from earning reward amount.

    2. Owner can withdraw the stake of any user before the duration of months complete without the consent of the user, in this scenario the user will get back the staked amount without any reward.

    **Recommendation**
    Whenever certain significant privileged actions are performed within the contract, it is recommended to emit an event about it. Changing the IR in the contract is a significant action that affects the rewards of the users. Hence, it is recommended to emit an event for this action. Moreover, IR rates must be locked and must be changed after certain times. If this does not resonate with business logic, we would recommend you to emit an event and let users know that the IR rates have been changed and you have some amount of time to make a decision regarding staking of the funds.

    **Status:** Fixed
    **Auditors' Comment:** Ruufpay Team has fixed this issue by fixing the IR value at the time of staking. This simply means that once the user has staked and after that, if the owner updates the IR, the user will still receive the previously promised rewards.

## Low severity issues

2. **Missing zero check**

   Contracts lack zero address checks, hence are prone to be initialized with zero addresses.

   constructor lacks zero address check for _homeToken.

   [#l105] changeOwner lacks zero address check for _owner.

   **Recommendation**
   Consider adding zero address checks in order to avoid risks of incorrect contract initializations.

   **Status:** Fixed
   **Auditors' Comment:** changeOwner function is removed and now it's divided into two functions in the latest commit (commit: acf6a6412851ec7bc68ea18c76e609622540316f)

3. **Critical address change**

   When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no privileged role.

   [#105] changeOwner() function sets a new owner address with a one step process which may lead to accidental loss of access control over privileged roles.

   **Recommendation**
   Consider switching to a two-step process for transferring critical and privileged roles.

   **Status:** Fixed

4. **Not Emitting Events**

   [#L61] changeIr() changes the interest rate of the pool but doesn't emit events.

   **Recommendation**
   Whenever certain significant privileged actions are performed within the contract, it is recommended to emit an event about it. Changing the IR in the contract is a significant action that affects the rewards of the users. Hence, it is recommended to emit an event for this action

   **Status:** Fixed

5. **untilRewards return value of getUserData() would be negative if current timestamp amount is greater than staked date :**

   In [#L111] getUserData on [#L117] while calculating untilRewards there can be a scenario where int256(block.timestamp) would be greater than int256(stakeDate + (uint256(months) * 30 days)) in this case the function will return a negative value.

```
111     function getUserData(address _user) external view returns(uint256 homeTokens, uint256 stakeDate, uint2
112         homeTokens = balances[_user].amount;
113         stakeDate = balances[_user].stakeDate;
114         pendingRewards = _calculateRewards(_user);
115         months = balances[_user].months;
116         multiplier = _calculateMultiplier(stakeDate, months);
117         untilRewards = int256(stakeDate + (uint256(months) * 30 days)) - int256(block.timestamp);
118         if (months == 3) finalIr = ir[0];
119         else if (months == 6) finalIr = ir[0] + ir[1];
120         else if (months == 9) finalIr = ir[0] + ir[1] + ir[2];
121         else if (months == 12) finalIr = ir[0] + ir[1] + ir[2] + ir[3];
122     }
```

   **Recommendation**
   Consider adding a check statement where if the value of current block.timestamp is greater or equal to the value of stakeDate + (months * 30 days) then it will assign zero for untilRewards.

   **Status:** Fixed

6. **getUserData() reverts and fails to get data of users who have staked funds for 9 and 12 months.**

   [#L111] getUserData() function is supposed to fetch the data of the user's who have staked in RuufStake. However, the function is not working as intended and will revert if one tries to fetch data of the user who has staked for a period of 9 and 12 months. This is because the values are overflowing/underflowing due to inappropriate type conversion.

```
111     function getUserData(address _user) external view returns(uint256 homeTokens, uint256 stakeDate, uint256 pendingRewards, uint256 multiplier, uint16 months,
112         homeTokens = balances[_user].amount;
113         stakeDate = balances[_user].stakeDate;
114         pendingRewards = _calculateRewards(_user);
115         months = balances[_user].months;
116         multiplier = _calculateMultiplier(stakeDate, months);
117         untilRewards = int256(stakeDate + (months * 30 days)) - int256(block.timestamp);
118         if (months == 3) finalIr = ir[0];
119         else if (months == 6) finalIr = ir[0] + ir[1];
120         else if (months == 9) finalIr = ir[0] + ir[1] + ir[2];
121         else if (months == 12) finalIr = ir[0] + ir[1] + ir[2] + ir[3];
122     }
```

**Recommendation**

The solution to this issue is two folds.

1. If you are fine with the inappropriate type conversion and overflow/underflow, just wrap the L#117 with unchecked {}. Now the solidity will handle this and the function will fetch the data.
2. Another solution to this is to use uint64 for months instead of uint16.

**Status:** Fixed

## Informational issues

7. **Redundant functions**

   [#L73] withdraw() and [#L77] withdraw() functions are calling the same internal _withdraw() function with the change of user argument.

**Recommendation**

[#L73] withdraw() and [#L77] withdraw() can be merged into one function. Owner can call the same withdraw function and functionality can be added to the code if msg.sender is the owner then he can enter any user address. If msg.sender is a normal user then msg.sender will be used as a user who is withdrawing the balance.

**Status:** Fixed

8. **Unindexed event parameters**

[#L25] HomeTokenStaked, [#L26] WithdrawWithRewards, [#L27] WithdrawWithoutRewards events don't have any indexed parameters. Unindexed parameters make it difficult to track important data for off-chain monitoring tools.

```
25      event HomeTokenStaked(address _user, uint _amount, uint _date);
26      event WithdrawWithRewards(address _user, uint _homeAmount, uint _rewardsAmount);
27      event WithdrawWithoutRewards(address _user, uint _homeAmount);
28      event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
29
```

**Recommendation**
Consider indexing event parameters to avoid the task of off-chain services searching and filtering for specific events.

**Status:** Fixed

9. **Denial-Of-Service attack**

[#L35] stake() takes user address as a parameter and does not use msg.sender to know if the address entered is the original user who is calling the function.

Any other user can use anyone’s address and stake amount, later if the original user whose address is already used by the attacker tries to stake tokens then the transaction will revert with the UserAlreadyStaked error message.

```
35      function stake(address _user, uint256 _amount, uint16 _months) external {
36          require(_amount > 0, "InvalidAmount");
37          require(_months == 3 || _months == 6 || _months == 9 || _months == 12, "InvalidMonths");
38          require(IERC20(homeToken).balanceOf(msg.sender) >= _amount, "NoEnoughTokens");
39          require(balances[_user].amount == 0, "UserAlreadyStaked");
40
41          if (_months == 3) require(totalTokensStaked[0] + _amount <= maxStakes[0], "MaxLimitReached3Months");
42          else if (_months == 6) require(totalTokensStaked[1] + _amount <= maxStakes[1], "MaxLimitReached6Months");
43          else if (_months == 9) require(totalTokensStaked[2] + _amount <= maxStakes[2], "MaxLimitReached9Months");
44          else if (_months == 12) require(totalTokensStaked[3] + _amount <= maxStakes[3], "MaxLimitReached12Months");
45
46          balances[_user] = UserStake({
47              amount: _amount,
48              stakeDate: uint64(block.timestamp),
49              months: _months
50          });
51
52          IERC20(homeToken).safeTransferFrom(msg.sender, address(this), _amount);
53          if (_months == 3) totalTokensStaked[0] += _amount;
54          else if (_months == 6) totalTokensStaked[1] += _amount;
55          else if (_months == 9) totalTokensStaked[2] += _amount;
56          else if (_months == 12) totalTokensStaked[3] += _amount;
57
58          emit HomeTokenStaked(_user, _amount, block.timestamp);
59      }
```

**Recommendation**

1. Consider adding a require check statement which will check if msg.sender is _user to check the original address is staking the tokens and not any other address.
2. Or consider replacing msg.sender where _user parameter is used to restrict any user entering any other user's address while staking.

**Status:** Fixed

# Functional Testing Results

Some of the tests performed are mentioned below:

- ☑ User can withdraw staked amount.
- ☑ Owner can withdraw for any user.
- ☑ Owner should be able to change the interest rate.
- ☑ Owner should be able to change the max stake limits.
- ☑ Owner only can change the old owner's address.
- ☑ Should validate that appropriate rewards are provided.
- ☑ Should getUserData of users who have staked tokens.

# Issues Identified Via Automation Testing

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

# Closing Summary

Some issues of Medium and Low severity were found, which were fixed by the Ruufpay team. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

# Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of RuufPay. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the RuufPay team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

# Audit Report
# March, 2022

For

RuufPay

QuillAudits


Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com