QuillAudits

Audit Report
April, 2022

For

KRIDA FANS

Table of Content

Executive Summary

Project Name	Kridafans
Overview	Krida Fans is a next-generation fantasy sports and social platform built on polygon blockchain. Krida Fans is powered by $KRIDA, which is a utility and governance token, enabling users to earn rewards and participate in community votes for important platform decisions.
Timeline	March 28,2022 to April 21,2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse Kridafans Staking codebase for quality, security, and correctness.
Source Code	*https://gist.github.com/kridafans/c546e0f86ec2c2fa0fdd1829afa890b9*
Fixed In	*https://gist.github.com/surajkathade/000057d8734bca34d3b6fc6f1afa4f7c*

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	1	3

Types of Severities

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.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility leve

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

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - StakingContract

High Severity Issues

No issues were found

Medium Severity Issues

No issues were found

Low Severity Issues

A.1 Use of Inefficient method

```
function _withdrawStake(uint256 index) internal returns(uint256){
    uint256 user_index = stakes[msg.sender];
    Stake memory current_stake = stakeholders[user_index].address_stakes[index];
    uint256 _epoch_period = _getLockingPeriod(current_stake.locking_period);
    require(
        (getCurrentTime().sub(current_stake.stake_time)) >= _epoch_period,
        "Staking: Cannot withdraw as stakes are locked for the locking period provided at the time of staking"
    );
    uint256 reward = calculateStakeReward(current_stake);
    delete stakeholders[user_index].address_stakes[index];//--->-use of Delete
    return current_stake.amount.add(reward);
}
```

Description

Using of “delete” to remove elements from an array is not recommended because it only removes the element but the length of array remains the same that could lead to unnecessary gas consumption while iterating over the array after several entries have been added and removed.

Remediation

Consider using whenNotPaused modifier to check and restrict _beforeTokenTransfer() from executing if the contract is paused.

Status

Fixed

Informational Issues

A.2 Unlocked pragma (pragma solidity ^0.8.11)

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same. Moreover, we strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor the current code base to only use stable features.

Status

Resolved

A.3 Missing event

```
function setRewardRate(
    uint256 _oneMonth,
    uint256 _sixMonth,
    uint256 _oneYear,
    uint256 _twoYear,
    uint256 _fourYear
)
public
onlyOwner{
    executeRewards();
    rewardRate.oneMonth = _oneMonth;
    rewardRate.sixMonth = _sixMonth;
    rewardRate.oneYear = _oneYear;
    rewardRate.twoYear = _twoYear;
    rewardRate.fourYear = _fourYear;
}
```

Description

In this function there is no event to be emitted after the reward rates are changed and on the front end it won't be reflected once this function call ends. Neither in this nor in the executeRewards() which will transfer the rewards to the users has an event to provide notification about a function call

Remediation

We recommend adding events in these functions.

Status

Resolved

A.4 Use of Inefficient method

```
function setRewardRate(
    uint256 _oneMonth,
    uint256 _sixMonth,
    uint256 _oneYear,
    uint256 _twoYear,
    uint256 _fourYear
)
public
onlyOwner{
    executeRewards();
    rewardRate.oneMonth = _oneMonth;
    rewardRate.sixMonth = _sixMonth;
    rewardRate.oneYear = _oneYear;
    rewardRate.twoYear = _twoYear;
    rewardRate.fourYear = _fourYear;
}
```

Description

This implementation could lead to a problem in a scenario where the tokenomics are not calculated properly and it's decided to call this function just within a few days of deployment. Use of this function may also induce the risk of this contract being used for rug pulls and could also raise questions about decentralization.

Remediation

From our audit we have concluded that the function should always be called with the accurate values and should not reach upto zero. Perhaps, the reward rate may be given in accordance to the timeline.

Status

Resolved

A.5 General Recommendations

In conclusion, we would like to mention that some of the error messages in the contract should be more detailed so that it will be more clear to the users of this contract about the reverts.

Status

Acknowledged

We would also like to recommend changing the statement in the require check on line[361] from "_amount < _token.balanceOf(msg.sender)" to "_amount <= _token.balanceOf(msg.sender)"

Status

Resolved

```
function stake(uint256 _amount, uint256 _lock) public {
    require(
        _amount < _token.balanceOf(msg.sender),
        "Token: Cannot stake more than you own"
    );
```

We have concluded that if the input of the locking period is being taken by the user and there are only 5 possible values then the use of an unsigned integer with specific(lower) bytes instead of 256 would be a good practice.

Status

Resolved

Functional Testing

Some of the tests performed are mentioned below

- should be able to stake and restake for certain locking periods
- Should be able to withdraw rewards after locking period is over
- Should be able to withdraw stake
- Should be able to set reward rate
- Should be able to compute supply
- Should be able to return values of rewards and stake for a staker
- Should be able to transfer rewards to users once setRewardRate function is called
- Should be able to calculate rewards according to the rate
- Should revert if withdraw functions are called before locking period is over
- Should revert if locking period is not correct
- Should revert if staking amount is zero
- Should revert if user wants to withdraw more than the amount (staked + rewards)

Automated Tests

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

In this report, we have considered the security of the Kridafans. We performed our audit according to the procedure described above.

Some issues of Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

At the end,the Kridafans Team resolved all of the issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Kridafans Platform. 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 KridafansTeam put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
April, 2022

For

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com