QuillAudits

Audit Report
June, 2021

Kangal

Contents

Audit Details and Target

1. Contract

Reward Token contract ($TEAK):
https://bscscan.com/address/0xba0f58179f5441d81d22402bd0183ffff130e243#code

Staking contract:
https://bscscan.com/address/0x222dc5cbc4d5082ac181532c01a57cc897ea4f15#code

2. Audit Target

- To find the security bugs and issues regarding security, potential risks and critical bugs.
- Check gas optimization and check gas consumption.
- Check function reusability and code optimisation.
- Test the limit for token transfer and check the accuracy in decimals.
- Check the functions and naming conventions.
- Check the code for proper checks before every function call.
- Event trigger checks for security and logs.
- Checks for constant and function visibility and dependencies.
- Validate the standard functions and checks.
- Check the business logic and correct implementation.
- Automated script testing for multiple use cases including transfers, including values and multi transfer check.
- Automated script testing to check the validations and limit tests before every function call.
- Check the use of data type and storage optimisation.
- Calculation checks for different use cases based on the transaction, calculations and reflected values.

Functions list and audit details

Major Functions

- mint()
- approveMinterRemoval()
- proposeMinterRemoval()
- approveProposedMinter()
- proposeMinter()
- updateConsensusState()
- removeMintingConsent()
- giveMintingConsent()
- pauseWithdrawals()
- pauseDeposits()
- setFeeVaultAddress()
- setMinimumStakeTime()
- setMinimumStakeAmount()
- calculateTotalPendingRewards()
- calculateLatestRewards()
- emergencyWithdraw()
- withdraw()
- claimRewards()
- deposit()

Overview of Contract

Kangal contracts are token contracts for rewards, token and staking contracts. Users can stake main tokens and get rewards based on the deposit time and rewards percentage. The contract is developed using the latest solidity version and updated contract standard functions using all security measures for token security and checks.
The main token contract follows ERC20 token contract functions and features for the development.
Reward token works on the logic of multiple owners and whitelisting for the miner to maintain maximum security for the contract for rewards.
This is the utility token that will be given for staking the Main token; it will be minted at the point of claiming reward or withdrawing by the staking contract. There are three admins, all of which are gnosis safe wallets.
Admins need to add a minter; to do this, they need to first propose a minter, and then another admin needs to approve this so that there is a 2/3 majority—the same process for removing a minter. Admins need to reach mintingConsensus of ⅔; otherwise, minting is not allowed. The token can be minted only by the operator that has a _minter role if minting consensus is reached.

Tokenomics

As per the information provided, the tokens generated will be initially transferred to the contract owner and then further division will be done based on the business logic of the application. Tokens cannot be directly purchased from the smart contract, so there will be an additional or third-party platform that will help users to purchase the tokens. Tokens can be held, transferred and delegated freely. Tokens are generated based on the supply, which can be checked by total supply.

Scope of Audit

The scope of this audit was to analyse Kangal smart contracts codebase for quality, security, and correctness.

Checked Vulnerabilities

The smart contract is scanned and checked for multiple types of possible bugs and issues. This mainly focuses on issues regarding security, attacks, mathematical errors, logical and business logic issues. Here are some of the commonly known vulnerabilities that are considered:

- TimeStamp dependencies.
- Variable and overflow
- Calculations and checks
- SHA values checks
- Vulnerabilities check for use case
- Standard function checks
- Checks for functions and required checks
- Gas optimisations and utilisation
- Check for token values after transfer
- Proper value updates for decimals
- Array checks
- Safemath checks
- Variable visibility and checks
- Error handling and crash issues
- Code length and function failure check
- Check for negative cases
- D-DOS attacks
- Comments and relevance
- Address hardcoded
- Modifiers check
- Library function use check
- Throw and inline assembly functions
- Locking and unlocking (if any)
- Ownable functions and transfer ownership
- checksArray and integer overflow possibility checks
- Revert or Rollback transactions check

Techniques and Methods

- Manual testing for each and every test cases for all functions.
- Running the functions, getting the outputs and verifying manually for multiple test cases.
- Automated script to check the values and functions based on automated test cases written in JS frameworks.
- Checking standard function and check compatibility with multiple wallets and platforms
- Checks with negative and positive test cases.
- Checks for multiple transactions at the same time and checks d-dos attacks.
- Validating multiple addresses for transactions and validating the results in managed excel.
- Get the details of failed and success cases and compare them with the expected output.
- Verifying gas usage and consumption and comparing with other standard token platforms and optimizing the results.
- Validate the transactions before sending and test the possibilities of attacks.

Structural Analysis
In this step we have analyzed 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.
SmartCheck.

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

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 security of smart contracts.

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 possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

High severity issues

Issues that must be fixed before deployment else they can create major issues.

Medium level severity issues

These issues will not create major issues in working but affect the performance of the smart contract.

Low level severity issues

These issues are more suggestions that should be implemented to refine the code in terms of gas, fees, speed and code accuracy

Informational

These are severity four issues which 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
Closed	0	0	2	2

Issues Found – Code Review / Manual Testing

High severity issues

No Issue found under this category

Medium severity issues

No Issue found under this category

Low level severity issues

1. **Contract:** StakingContract.sol
   Line: 90
   Issues: Add check approval to revert contract if fails.
   Reasons: Checking approval from the token contract before execution fail with gas loss.
   Suggested Fixes: Add modifier to check approval.

```
86
87    function deposit(uint256 amount) external nonReentrant whenDepositNotPaused {
88      require(amount <= stakedToken.balanceOf(msg.sender), 'NOT ENOUGH TOKEN BALANCE');
89      require(amount >= minimumStakeAmount, 'AMOUNT CANNOT BE SMALLER THAN MINIMUM AMOUNT');
90
91      uint256 balanceOfAccount = stakedBalances[msg.sender];
```

   Status: Closed

   Contract: RewardToken.sol
2. **Line:** NA
   Issues: Same owner addresses failed in multiple scenarios.
   Reasons: If any two owners are the same, then this may be an issue with a further logic.
   Suggested Fixes: Add a check for different owners, function to get list of owners.
   Status: Closed

Informational

1. **Contract:** StakingContract.sol
   Line: 87
   Issues: Add description/Approve function details/Checks in comment.
   Reasons: For transparency, user guide and code optimisation.
   Suggested Fixes: Add informative comment

```
86
87  function deposit(uint256 amount) external nonReentrant whenDepositNotPaused {
88      require(amount <= stakedToken.balanceOf(msg.sender), 'NOT ENOUGH TOKEN BALANCE');
89      require(amount >= minimumStakeAmount, 'AMOUNT CANNOT BE SMALLER THAN MINIMUM AMOUNT');
90
91      uint256 balanceOfAccount = stakedBalances[msg.sender];
```

   Status: Closed

2. **Contract:** RewardToken.sol
   Suggested Fixes: Add more descriptive comments so that users can understand the working for better transparency.
   Status: Closed

Functional Test Table

Function Names	Technical results	Logical results	Overall
approveMinterRemoval()	Pass	Pass	Pass
proposeMinterRemoval()	Pass	Pass	Pass
approveProposedMinter()	Pass	Pass	Pass
proposeMinter()	Pass	Pass	Pass
updateConsensusState()	Pass	Pass	Pass
removeMintingConsent()	Pass	Pass	Pass
giveMintingConsent()	Pass	Pass	Pass
pauseWithdrawals()	Pass	Pass	Pass
pauseDeposits()	Pass	Pass	Pass
setFeeVaultAddress()	Pass	Pass	Pass
setMinimumStakeTime()	Pass	Pass	Pass
setMinimumStakeAmount()	Pass	Pass	Pass
calculateTotalPendingRewards()	Pass	Pass	Pass
calculateLatestRewards()	Pass	Pass	Pass
emergencyWithdraw()	Pass	Pass	Pass
withdraw()	Pass	Pass	Pass
claimRewards()	Pass	Pass	Pass
deposit()	Pass	Pass	Pass

Closing Summary

All the issues and suggestions are fixed properly and tested. Code is clean, and code comments are descriptive. This contract can be used for the staking and reward logic with safe transfer and checks. Both the contracts are tested and bugs reported are fixed now.

Disclaimer

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

Kangal

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com