QuillAudits

Audit Report
August, 2022

For

kichee

Table of Content

Table of Content

Executive Summary

Project Name	Kichee
Overview	KICHEE Tokens are tokenized voluntary carbon credits, namely Carbon Offset Units (COUs), built on the Polygon blockchain
Timeline	28 July,2022 to 29 August,2022
Method	Manual Review, Functional Testing, Automated Testing, etc.
Scope of Audit	The scope of this audit was to analyse Kichee smart contracts for quality, security, and correctness.

Review 1: *https://github.com/FriendlyAmbar/kichee-contract/blob/master/contract.sol*

Review 2: *https://testnet.bscscan.com/address/0x1df3e9c1bf4f4a92d9d088f7352c69897319bee8#code*

Review 3: Contract Address 0x48b3cc2976260d8ef5dfaaefc7fa90ee87c8ddd9 | PolygonScan

Review 4: *https://polygonscan.com/address/0x9e61273e75c3cd3a3c630fde2afab79dbd69988a#code*

16
Issues Found

🟥 High 🟨 Medium 🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	**3**	**5**	**3**
Partially Resolved Issues	0	0	0	**1**
Resolved Issues	0	**4**	0	0

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
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ 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

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

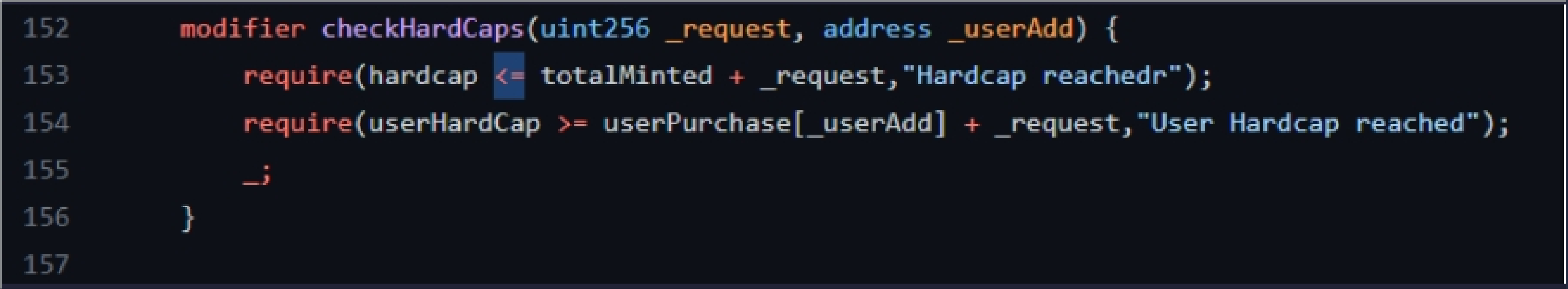
Manual Testing

High Severity Issues

No issues found

Medium Severity Issues

1. Broken Functionality

Line	modifier - checkHardCaps
152	

```
        modifier checkHardCaps(uint256 _request, address _userAdd) {
            require(hardcap <= totalMinted + _request,"Hardcap reachedr");
            require(userHardCap >= userPurchase[_userAdd] + _request,"User Hardcap reached");
            _;
        }
```

Description

checkHardCaps modifier checks for hardcap <= totalMinted + _request which means hardcap should be less than or equal to totalminted tokens plus requested tokens. Here there can following type of scenario:

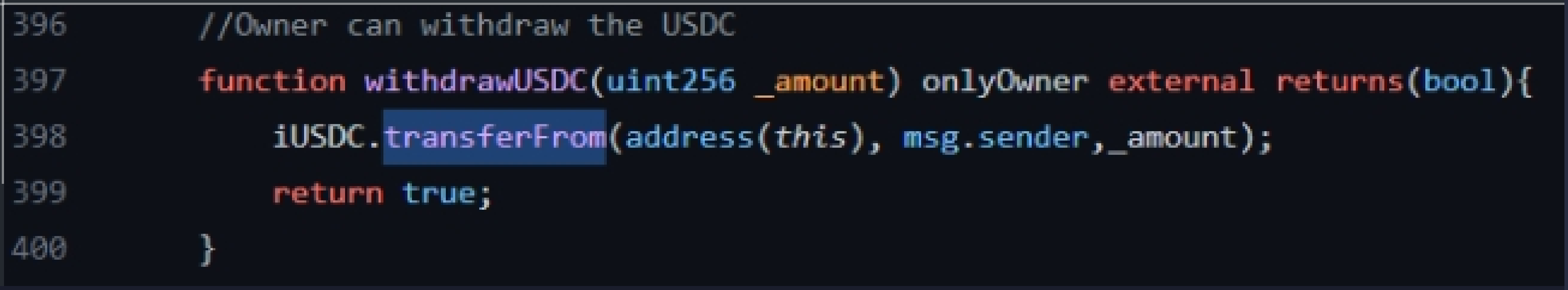
Remediation

change <= check to >= in first condition in checkHardCaps modifier. Which will check for - (the amount of already minted tokens + the amount of tokens user wants to buy) is less than hardcap.

Status

Resolved

2. Broken Functionality

Line	Function - withdrawUSDC
397	

```
396        //Owner can withdraw the USDC
397        function withdrawUSDC(uint256 _amount) onlyOwner external returns(bool){
398            iUSDC.transferFrom(address(this), msg.sender,_amount);
399            return true;
400        }
```

Description

withdrawUSDC() is using ERC20 transferFrom instead of transfer , Which will revert with 'ERC20: insufficient allowance' error message as this contract don’t has allowance to itself to move to

Remediation

Use transfer() instead of transferFrom in withdrawUSDC() function.

Status

Resolved

3. Old solidity version

Description

The contract is using solidity version 0.4.24. Using an old version prevents access to new Solidity security checks. This contract contains some functions or codeblocks that are not using safemath protection.

Remediation

Consider using the latest solidity version.

Status

Acknowledged

4. Missing test cases

Description

Test cases for the code and functions have not been provided.

Remediation

It is recommended to write test cases of all the functions. Any existing tests that fail must be resolved. Tests will help in determining if the code is working in the expected way. Unit tests, functional tests should have been performed to achieve good test coverage across the entire codebase. We recommend unit testing for the scenarios with expected high and low price of usdc and matic for Kichee token.

Status

Acknowledged

Kichee team's comment: We are testing functionality manually with remix and with UI.

5. totalSupply() can exceed the hardcap amount

Description

In the scenario where the owner sets hardcap as some value. some users buy tokens and there are only some amount of tokens left to reach the hardcap. There are some whitelisted addresses to which admin has allocated some amount of tokens which they can claim. now for e.g some users buy tokens using buyTokensByUSDC or buyTokensByMatic before whitelisted addresses can claim , after that whitelisted addresses claim their allocated tokens. in this scenario the totalsupply will exceed the hardcap as whitelistForClaim function is not increasing value of totalMinted variable which is getting checked in modifier which checks if hardcap is reached.

Remediation

Consider reviewing the code logic.

Status

Resolved

6. Amount is getting assigned instead of adding it to totalMinted

Description

whitelistForClaim() is allocating amount to addresses so that these users can claim tokens after. Value of allocated amount _amount[i] on line 429 is getting assigned to totalMinted variable and not getting added while iterating through for loop. This will set the totalMinted to last amount that is given from the array for allocation. In this case totalMinted should get updated to total allocated amount but its getting reassigned to the value of last allocated amount.
The value of totalMinted that is getting used in checkHardCaps, the value wont get updated and because of which users may buy tokens even while hardcap gets reched.

Remediation

Consider reviewing logic. Consider adding _amount[i] to totalMinted instead of assigning it.

Status

Resolved

7. Missing zero address check in whitelistForClaim

Description

whitelistForClaim() lacks zero address check for _userAddress. It is important to add zero address check for current logic. In the case where some amount gets allocated to zero address. The code will modify (in next fixed version of this contract) the totalMinted, but this amount cant be claimed, and admin cant deduct/cancel the allocation.

Remediation

Add zero address check in whitelistForClaim for entered _userAddress array.

Status

Acknowledged

Low Severity Issues

8. Care needs to be taken while allocating tokens

Description

1. Admin can by mistake exceed totalsupply

Admin can keep whitelisting allocations because there's no check/modifier to check whitelisted tokens are not exceeding hardcap

2. Admin can by mistake allocate more tokens

Admin/backend service can allocate more tokens to any address(s), looking at current code it is not possible to deduct the allocated tokens if more tokens get allocated as compared to the intended amount

Remediation

Remediation for description point 1 and 2 respectively:

1. Add require check which will check that the allocations to whitelisted addresses are not exceeding hardcap.
2. Care needs to be taken while allocating tokens to any address. Review code logic . Other functionalities can be added to deduct the amount of token allocated to any address in the case of any mistake as described.

Status

Acknowledged

9. Care needs to be taken for external calls

Description

buyTokensByUSDC makes external call on line 372 to transfer tokens from msg.sender to the contract. It is important to make calls to trusted addresses only where theres no possibility of reentering in transaction flow maliciously. It is important to take care of reentrancy by adding any checks or by using modifiers like reentrancyguard.

Remediation

It is advised that the team cover at least 80 percent of the test cases.

Status

Acknowledged

10. Check user hardcap logic

Description

1. Contract code implements user hardcap logic. Which limits users from buying more than certain amount of tokens. It needs to be noted that users can create new addresses to buy tokens more than the certain limit specified in contract.

2. Additionally , The logic that is getting implemented in whitelistForClaim and claimTokens functions is prone to some conditions where user can buy more than user hardcap. For e.g user can buy tokens off chain and admin will allocate tokens to that user on chain using this smart contract so that user can claim. In this scenario a whitelisted user can buy tokens on chain using smart contract functions and then he can claim the tokens even if user-hardcap is getting exceeded because userPurchase mapping (which is getting checked in checkHardCaps modifier) is getting updated in claimTokens() function i.e when user claims tokens and not when it is getting allocated.

Remediation

Review smart contract logic. Consider updating userPurchase mapping in whitelistForClaim() so that the user can only buy tokens according to the user hardcap set.

Status

Acknowledged

11. Lack of events

Description

Events for some important admin actions can be added. E.g : setUSDCPrice , setMaticPrice, setHardcap, setUserHardCap

Remediation

Consider adding events

Status

Acknowledged

12. _metaData string variable

Description

buyTokensByUSDC() and buyTokensByMatic() functions are taking _metaData parameter which is getting used to solve business logic according to the project team. The parameter is not getting used in code and not getting stored in smart contract storage

Remediation

Consider reviewing business logic and code logic. Use state variables to store any type of data that is required in future.

Status

Acknowledged

Informational Issues

13. Some functions can be declared external

Description

Some functions can be declared external that are not getting called by this contract, doing so can save gas each time function is executed. E.g: setUserHardCap, setHardcap, setMaticPrice, setUSDCPrice, buyTokensByMatic, buyTokensByUSDC, checkMinMaxToBuy, checkUserBurnTokens.

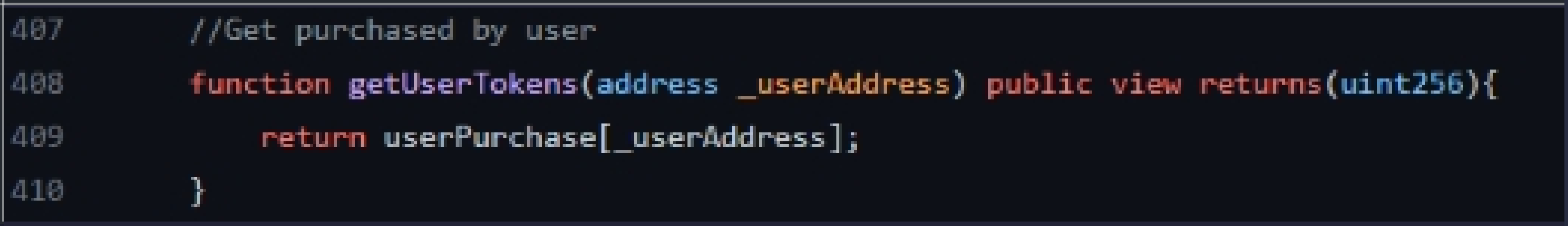
Remediation

Consider changing the visibility of these functions to external.

Status

Partially Resolved

14. Redundant method

Line	Function - getUserTokens
408	

```
407        //Get purchased by user
408        function getUserTokens(address _userAddress) public view returns(uint256){
409            return userPurchase[_userAddress];
410        }
```

Description

The compiler automatically creates getter functions for all public state variables. for userPurchase public mapping compiler will create public getter method from which the values can be accessed. Additionally there are some getter methods added for returning public variables where these variables can be accessed directly by using getter methods created by solidity compiler, No need to define new getter method.

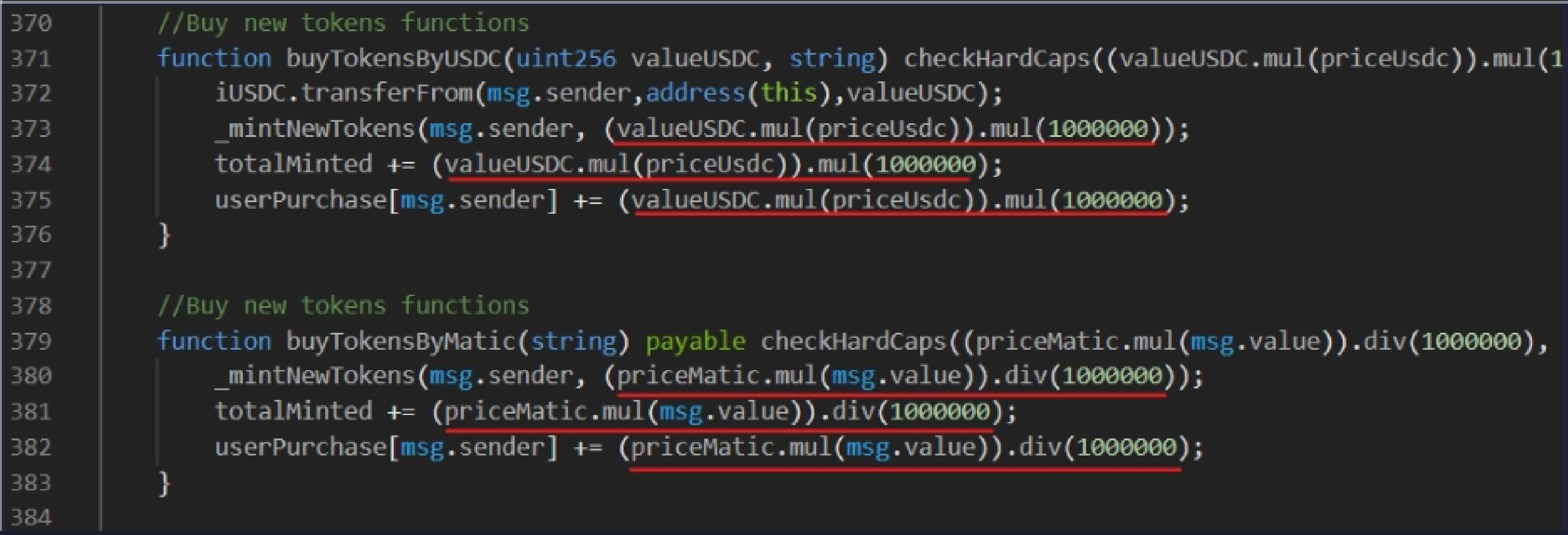
Remediation

Consider removing getUserTokens() and some other getter methods which are returning public variables

Status

Acknowledged

15. General Recommendation

Line	Function - getUserTokens
408	

```
    //Buy new tokens functions
    function buyTokensByUSDC(uint256 valueUSDC, string) checkHardCaps((valueUSDC.mul(priceUsdc)).mul(1
        iUSDC.transferFrom(msg.sender,address(this),valueUSDC);
        _mintNewTokens(msg.sender, (valueUSDC.mul(priceUsdc)).mul(1000000));
        totalMinted += (valueUSDC.mul(priceUsdc)).mul(1000000);
        userPurchase[msg.sender] += (valueUSDC.mul(priceUsdc)).mul(1000000);
    }

    //Buy new tokens functions
    function buyTokensByMatic(string) payable checkHardCaps((priceMatic.mul(msg.value)).div(1000000),
        _mintNewTokens(msg.sender, (priceMatic.mul(msg.value)).div(1000000));
        totalMinted += (priceMatic.mul(msg.value)).div(1000000);
        userPurchase[msg.sender] += (priceMatic.mul(msg.value)).div(1000000);
    }

```

Description

buyTokensByUSDC() and buyTokensByMatic() functions calculate amount of tokens to give in return when user purchases tokens with usdc or with matic. This calculation for kichee token according to price can be stored in one variable instead of calculating again on every line. For e.g in the case of buyTokensByUSDC() its using (valueUSDC.mul(priceUsdc)).mul(1000000) and in buyTokensByMatic() its using (priceMatic.mul(msg.value)).div(1000000) three times which can be stored in one variable and this variable can be used three times.

Remediation

Consider removing getUserTokens() and some other getter methods which are returning public variables

Status

Acknowledged

16. Check function name and error message

Description

There are some spelling mistakes in function name and in error message. incorrect naming may create confusion while calling contract functions through backend services and maybe while handling errors. Check functions name checkClaimabkleTokens and error message Hardcap reachedr on line 161.

Remediation

It is advised to follow this pattern to avoid the danger of any reentrancy attack. Also it is advised that the external call on line: 2270 of safeTransferFrom be done at the end of the function sellTokenTo() as it transfers the call of execution of the function to an external contract function that could be malicious.

Status

Acknowledged

Functional Testing

- ✓ Should be able to buy tokens with matic
- ✓ Should be able to buy tokens with USDC
- ✓ Should be able to set token rate for USDC
- ✓ Should be able to set tokens rate for matic
- ✓ Should be able to set hardcap
- ✓ Should be able to set user hardcap
- ✓ Should be able to burn available tokens
- ✓ Should be able to withdraw matic
- ✓ Should be able to withdraw USDC
- ✓ Reverts when user tries to burn more than available tokens
- ✓ Reverts when hardcap exceeds
- ✓ Reverts when hardcap for any user exceeds

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 Kichee. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the Kichee Team resolved few issues and Acknowledged other Issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Kichee 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 Kichee Team 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 August, 2022

For

kichee

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com