QuillAudits

Audit Report
June, 2024

For

ecotrader.io

Table of Content

Executive Summary 03

Number of Security Issues per Severity 04

Checked Vulnerabilities 05

Techniques and Methods 06

Types of Severity 07

Types of Issues 07

High Severity Issues 08

Medium Severity Issues 08

1. Syntactial error in getInvestorDetails 08

2. Incorrect token amount minting for each token 09

3. _getUsdcEquivalent returns incorrect USDC equivalent amount for matic 09

4. set the vestingMonthsRemaining to zero if elapsedTime is greater than 20 months 11

5. Centralization concern 13

6. Add a slippage check in the invest() 14

7. Incorrect investedValue calculation. 15

Low Severity Issues 16

8. Updated libraries should be used 16

9. View functions with onlyAdmin modifier 16

10. Incorrect event argument 17

Table of Content

11. Incorrect naming 18

12. Users can claim for elapsed months without waiting if already invested 19

13. Floating pragma 20

14. Remove redundant logic 21

Informational Issues 22

15. Incorrect event parameter name 22

16. The maximum investment limit can be tricked 22

17. Discrepancy in decimals 23

18. The contract has two pausing mechanism 24

19. The contract has no start and end time to invest. 24

20. Note regarding some function behaviors 25

21. Remove redundant require check in _withdraw() 26

Functional Tests Cases 27

Automated Tests 27

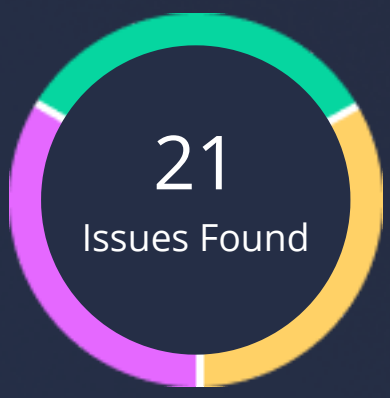
Closing Summary 28

Disclaimer 28

Executive Summary

Project Name	Ecotrader
Overview	InvestmentToken is a vesting contact that allows users to invest tokens and then claim the investment token after 20 months or for the elapsed time. Users can invest USDC, MATIC and WETH to get the Investment Token.
Timeline	1st April 2024 - 14th April 2024
Updated Code Received	3rd June 2024
Second Review	22nd April 2024 - 23rd April 2024
Third Review	23rd May 2024
Fourth Review	13th June 2024 - 17th June 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyse the InvestmentToken Contract for quality, security, and correctness.
Source Code	https://gitlab.com/-/snippets/3692412
Contracts In-Scope	InvestmentToken

Number of Security Issues per Severity



High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	2	4
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	7	5	3

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
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

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

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC's 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 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.

Static Analysis

A 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 analyzed, 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 behavior 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

Hardhat, Foundry.

Types of Severity

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.

High Severity Issues

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 Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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 four 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.

High Severity Issues

No issues were found.

Medium Severity Issues

1. Syntactial error in getInvestorDetails

Path

InvestmentToken#L321

Function

getInvestorDetails()

Description

The contract has a syntactical error on L324-L347. Because of this, the contract won't be able to compile. The function declaration + definition should be checked again for syntactical errors and should be fixed.

Recommendation

Consider fixing the syntax error.

Status

Resolved

2. Incorrect token amount minting for each token

Path

InvestmentToken

Description

According to the comment on L18, 6.67 tokens per dollar should be minted. Currently, TOKEN_PER_DOLLAR is assigned as 667. which is getting used while minting tokens on L147 and for division on L349.

While calculating tokenAmount on L147, let's say the total investment amount (totalInvestmentAmount) is 1e6 and its getting multiplied with TOKEN_PER_DOLLAR

So the answer would be 1e6 * 667 = 667000000 which is equal to 667e6. This is incorrect because according to the comment, it should mint only mint 6.67 tokens per USDC.

This can be corrected by making TOKEN_PER_DOLLAR = 6670000 (i.e 6.67e6) on **L18**.

So the calculation for tokenAmount on L147 would be:
tokenAmount = (totalInvestmentAmount * TOKEN_PER_DOLLAR) / 1e6

So when the totalInvestmentAmount would be 1e6:
tokenAmount = (1e6 * 6670000) / 1e6
tokenAmount = 6670000 (i.e 6.67e6)

Recommendation

Change the value of TOKEN_PER_DOLLAR to 6670000 as suggested above.

Status

Resolved

3. _getUsdcEquivalent returns incorrect USDC equivalent amount for matic

Path

InvestmentToken#L160

Function

_getUsdcEquivalent()

Description

_getUsdcEquivalent() function currently supports wMatic to USDC path while getting the amountOut. Because on L170 the path[0] is assigned as uniswapRouter.WETH() which would be wMATIC and not WETH on the Quickswap router this can be verified by checking **this address.**

Invest() calls _getUsdcEquivalent() on L110 also for getting USDC equivalent for WETH. But in this case, because the _getUsdcEquivalent() only supports the wMATIC to USDC path. The USDC equivalent that it returns is incorrect when checking for WETH to USDC.

Recommendation

Add support for WETH to USDC path in _getUsdcEquivalent(). This can be done by using if else statement and one boolean parameter. The updated code would look like this:

```
function _getUsdcEquivalent(uint256 ethAmount, bool isMatic)
        public
        view
        returns (uint256)
    {
        if (ethAmount == 0) {
            return 0;
        }


        address[] memory path = new address[](2);
        if (isMatic) {
            path[0] = uniswapRouter.WETH(); // uniswapRouter.WETH() is wMatic.
        }else {
            path[0] = 0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619; // WETH address on
polygon.
        }
path[1] = usdcAddress;


        uint256[] memory amounts = uniswapRouter.getAmountsOut(ethAmount, path);
        return amounts[1];
    }
```

In the above code _getUsdcEquivalent() takes two parameters, there's an additional isMatic parameter that helps to choose the correct path for the fetching price.

Additionally the calls to _getUsdcEquivalent() on L109,L110 can be changed to the code below to support the changed _getUsdcEquivalent() function:

uint256 usdcEquivalent = _getUsdcEquivalent(ethAmount, true);
uint256 maticUsdcEquivalent = _getUsdcEquivalent(maticAmount, false);

Status

Resolved

4. set the vestingMonthsRemaining to zero if elapsedTime is greater than 20 months

Path

InvestmentToken

Description

In getInvestorDetails() while finding vestingMonthsRemaining, subtraction happens for (vestingDuration - elapsedTime). here if the elapsedTime is greater than vestingDuration which is 20 months timestamp, then the the call will revert with the panic code 0x11. because of arithmetic overflow.

This can fail some off-chain systems that check the investor's details for important logic.

Recommendation

To avoid the case mentioned above, If elapsedTime is greater than 20 months then vestingMonthsRemaining should be set as 0.

The modified code would look like this:

```
function getInvestorDetails(address investor)
        external
        view
        returns (
            uint256 investedAmount,
            uint256 investedValue,
            uint256 claimedTokens,
            uint256 claimableTokens,
            uint256 vestingStartTime,
            uint256 vestingMonthsPassed,
            uint256 vestingMonthsRemaining
        )
    {
        investedAmount = investorTokenBalance[investor];
        investedValue = investedAmount / TOKEN_PER_DOLLAR;
        claimedTokens = investorClaimedTokens[investor];
        vestingStartTime = investorVestingStartTime[investor];


        if (vestingStartTime > 0) {
            claimableTokens = _calculateClaimableTokens(investor);


            uint256 vestingDuration = 20 * 30 days; // 20 months in seconds
            uint256 elapsedTime = block.timestamp - vestingStartTime;
            if(elapsedTime > vestingDuration){
                vestingMonthsRemaining = 0;
            }
            else {
                vestingMonthsRemaining = (vestingDuration - elapsedTime) / 30 days;
            }
            vestingMonthsPassed = elapsedTime / 30 days;
        }
    }
```

Status

Resolved

5. Centralization concern

Path

InvestmentToken

Function

withdrawMintedTokens()

Description

The contract has a function named withdrawMintedTokens(), This can be used by the admin to withdraw the minted tokens to the contract which gets minted when the user invests so that can be claimed by the user.

Recommendation

Consider removing this function which reduces the centralization concern and increases the trust among users.

Status

Resolved

6. Add a slippage check in the invest()

Path

InvestmentToken#L96

Function

invest()

Description

Because of the public nature of blockchain, it's possible to monitor broadcasted transactions.

Anyone can monitor the investment transactions and may try to frontrun the transaction where before executing the victim transaction the malicious user will manipulate the token pool(s) so that when the investment contract fetches the equivalent price in USDC for WETH/wMATIC, the price fetched would be less than what it was expected by the user before sending the transaction.

In this way, the totalInvestmentAmount would be less than what it would be if someone doesn't perform the transaction frontrunning to manipulate the the pools.

The frontrunning can happen intentionally or unintentionally where the pools will get manipulated which will result in less totalInvestmentAmount than what it should be.

Recommendation

This can be mitigated on some level by using the slippage parameter. The invest() will take an additional slippage parameter where the user will specify the minimum total usdc price (for totalInvestmentAmount) which would be calculated by user off-chain/on frontend. if the calculated totalInvestmentAmount is less than the entered slippage then it will revert.

Status

Resolved

7. Incorrect investedValue calculation.

Path

InvestmentToken#L334

Function

getInvestorDetails()

Description

Currently investedValue is calculated using investedAmount / TOKEN_PER_DOLLAR / 1e6 this is incorrect. On L334 the calculation for investedValue should be changed to investedValue = investedAmount * 1e6 / TOKEN_PER_DOLLAR;

Example:
lets say investorTokenBalance[address] is 6670000 (6.67e6) which gets assigned to investedAmount on L333

so, investedValue = investedAmount * 1e6 / TOKEN_PER_DOLLAR
investedValue = 6670000 * 1e6 / 6670000
investedValue = 1e6 (the invested value.)

Recommendation

Change to code on L334 to investedValue = investedAmount * 1e6 / TOKEN_PER_DOLLAR; as suggested.

Status

Resolved

Low Severity Issues

8. Updated libraries should be used

Path

InvestmentToken

Description

InvestmentToken uses some OZ libraries like Ownable, Pausable, ReentrancyGuard. These libraries should be imported for the updated version.

Changing/updating the OZ dependency version will change the path for some of the imported libs. For E.g

For import “@openzeppelin/contracts/security/Pausable.sol”; and
import “@openzeppelin/contracts/security/ReentrancyGuard.sol”;

This updated path can be used:
import “@openzeppelin/contracts/utils/Pausable.sol”;
import “@openzeppelin/contracts/utils/ReentrancyGuard.sol”;

Recommendation

Use the updated version of Openzeppelin contracts and use the mentioned paths for imports.

Status

Resolved

9. View functions with onlyAdmin modifier

Path

InvestmentToken#L309, InvestmentToken#L313 , InvestmentToken#L317 ,InvestmentToken#L321

Function

getEthBalance(), getUsdcBalance(), getMaticBalance(), getInvestorDetails()

Description

View functions like getEthBalance(), getUsdcBalance(), getMaticBalance(), getInvestorDetails() are using onlyAdmin.

The view functions should not use the modifier(s) because these functions are needed by the users (or other contracts in some cases when integrating) to know the state of the contract e.g. getInvestorDetails() can be used by users to know details about the their investment.

Additionally, it makes the contract state more transparent.

Recommendation

Remove the modifier from the function definitions for the view functions.

Status

Resolved

10. Incorrect event argument

Path

InvestmentToken#L235

Function

setAdmin2()

Description

setAdmin2() function emits the AdminUpdated() event but for the second isAdmin1 parameter it passes argument as false while setting the _admin2 address as a new admin.

Recommendation

Consider checking the intended behavior and replacing false with true.

Status

Resolved

11. Incorrect naming

Path

InvestmentToken#L96

Function

invest()

Description

According to the specification document, the user can invest USDC, MATIC and ETH.

The invest() function accepts the MATIC as msg.value but invest() still takes maticAmount as a parameter. Additionally, the constructor sets the maticAddress.

According to a discussion with the project team, it should be WETH address because the function allows the user to invest three tokens MATIC (paid via msg.value), USDC and ETH(WETH). So in the code maticAmount should be renamed to wethAmount and maticAddress should be renamed to wethAddress.

Also, change the name of withdrawMatic() to withdrawWeth().

Recommendation

Change the naming of MATIC to WETH as suggested above

Status

Resolved

12. Users can claim for elapsed months without waiting if already invested

Path

InvestmentToken

Function

claim(), _calculateClaimableTokens()

Description

Users would be able to claim some percentage of the amount for the elapsed month directly after investing.

This is because in the _calculateClaimableTokens() function, the claimablePercentage gets calculated with monthsPassed (already elapsed time) let's say that is 19 months already.

So now when the user invests again after 19 months the claimableTokens will be calculated according to the elapsed time. So the user can claim the amount for 19 months without waiting for 19 months.

Additionally, the user can perform this action in a way where the already invested user will invest after 19 month and after 1 month (when 20 months will complete) will take 100 percent claim amount.

In another case, an already invested user can invest and claim both after 20 months have elapsed, so in this case user would be able to directly claim the amount without waiting.

Our recommendation is to consider checking that it matches the business logic.

Recommendation

consider checking that the above-mentioned scenario matches the business logic.

Status

Acknowledged

13. Floating pragma

Path

InvestmentToken

Description

InvestmentToken does not use fixed solidity versions. The contract is using unfixed pragma (^0.8.0), Contract should be deployed with the same compiler version and flags that they have been tested thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. The most recent compiler version may get selected while deploying a contract which has a higher chance of having bugs in it.

Recommendation

Remove floating pragma and use a specific compiler version with which contracts have been tested e.g. in this case 0.8.20 can be used which also matches the compiler version used by libraries.

Status

Acknowledged

14. Remove redundant logic

Path

InvestmentToken#L254-260

Function

_withdraw()

Description

Currently _withdraw() contains logic to handle (token == address(this)) in if{}. Because the withdrawMintedTokens() is removed, The redundant code needs to be removed from _withdraw().

Recommendation

Remove the if{} and move the code from the else{} block outside of it.

The code would look like this:

```
function _withdraw(
        address token,
        uint256 amount,
        address recipient
    ) internal {
        require(amount > 0, “Withdrawal amount must be greater than zero”);
        require(recipient != address(0), “Recipient address cannot be zero”);


        require(
            IERC20(token).balanceOf(address(this)) >= amount,
            “Insufficient token balance”
        );
        IERC20(token).transfer(recipient, amount);
        emit TokensWithdrawn(recipient, amount);
    }
```

Status

Resolved

Informational Issues

15. Incorrect event parameter name

Path

InvestmentToken#34

Description

setAdmin1() and setAdmin2() are emitting AdminUpdated() event, the AdminUpdated() has second parameter name as isAdmin1 whereas this should be isAdmin2 when getting emitted by setAdmin2() function.

This is happening because of using the same event for both functions. This may create confusion while reading these events.

Recommendation

Consider using different events for setAdmin1() and setAdmin2() functions e.g. AdminUpdated1 and AdminUpdated2 with a second parameter as isAdmin2.

Status

Resolved

16. The maximum investment limit can be tricked

Path

InvestmentToken#L119

Function

invest()

Description

The maximum investment limit check on the L119 can be tricked by users by using multiple addresses by one user to invest more amount than the maximumInvestment limit.

The issue doesn't create any direct security risk and is getting highlighted only for providing more information.

Recommendation

The issue can be acknowledged if the project team is aware of it.

Status

Acknowledged

17. Discrepancy in decimals

Path

InvestmentToken#L148

Function

invest()

Description

While minting an Investment token amount on L148 it's minting tokenAmount which would be in 6 decimals. according to the contract's ERC.decimal() function the contract uses 18 decimals to represent one whole token.

It needs to be checked it's intended to use 6 decimal values while minting.

Recommendation

Verify that the investment token will be using 6 decimals while minting tokens. If it would be using 6 decimals then override the decimals() function and assign 6 (instead of 18).

While decimals() is only for representational purposes, it helps understand how the exact 1 whole token would look in numbers.

Status

Resolved

18. The contract has two pausing mechanism

Path

InvestmentToken

Description

This should be noted that the contract has two pausing mechanisms.

The pauseInvestment() and unpauseInvestment() functions set the value for investmentPaused. And pause() and unpause() which calls inherited _pause() and _unpause() from the Pausable contract respectively.

whenNotPaused() modifier from the inherited functionality is getting used for the invest() function and investmentPaused value is getting used for the invest() function on the L102.

Recommendation

The redundant functionality for pausing the investment can be removed.

Status

Acknowledged

19. The contract has no start and end time to invest.

Path

InvestmentToken

Description

Currently, users can invest directly after the contract gets deployed. There’s no start and end period for calling invest().

Recommendation

Verify the business logic and check that there’s no requirement for start and end periods for calling invest().

Status

Acknowledged

Ecotrader Team’s Comment

We will be using the pausing mechanisms to manage seed rounds.

20. Note regarding some function behaviors

Path

InvestmentToken

Description

1. **getInvestorDetails()** returns vestingMonthsRemaining as 0 for not invested address. This can create a problem if any incorrect assumption is made for not invested address (maybe in off-chain service) in this case apart from 0.
2. **investorTokenBalance[user address]** is not getting subtracted after claiming all the amount. Any incorrect assumption regarding this can create problems in another system (smart contract or off-chain service) where that system will expect that the investorTokenBalance for the address that claimed all the amount is 0.

Recommendation

The mentioned code behaviors should be taken into account before making assumptions regarding the code.

Status

Resolved

QuillAudits Team's Comment

1st point is fixed by returning type(uint256).max for not invested address. 2nd point is fixed by using getCurrentInvestorTokenBalance() function for getting the currently available invested token balance.

21. Remove redundant require check in _withdraw()

Path

InvestmentToken

Function

_withdraw()

Description

Currently _withdraw() is getting used in withdrawUSDC() and withdrawWeth() , in these two functions its checking that the amount that admin is trying to withdraw is less than or equal to what this contract address holds, the similar thing is getting checked in _withdraw() on L256. This redundant check can be removed.

Recommendation

Remove redundant balance check on L256 from _withdraw().

Status

Acknowledged

Functional Tests Cases

Some of the tests performed are mentioned below:

- Should be able to invest in USDC, MATIC, and WETH.
- Should be able to claim the amount partially for the elapsed month.
- Should be able to claim the whole amount after 20 months.
- Should be able to invest and claim multiple times.
- Admin should be able to withdraw invested tokens.
- Admin should be able to pause the investment functionality.
- Reverts if the investments are paused
- Reverts if the claimable amount is 0
- Reverts if the investment amount is less than minimumInvestment
- Reverts if the investment amount is greater than maximumInvestment

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 Ecotrader codebase. 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.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Ecotrader smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Ecotrader smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Ecotrader to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
June, 2024

For

ecotrader.io

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com