QuillAudits

Audit Report
February, 2022

For

FUTIRA

Contents

Overview

FutiraCoin

FutiraCoin is an ERC20 token on the Futira Chain, a private and permissioned blockchain that will not be anonymous. 10 billion coins will be issued via sale with vesting and the coin anticipates a cap of 20 billion coins.

Scope of the Audit

The scope of this audit was to analyze FutiraCoin smart contract's codebase for quality, security, and correctness.

FutiraCoin Contracts code is Taken from

Source code file Provided by Futira coin team - https://docs.google.com/document/d/1aSOYreFQVxGugPx2x2WEckSBVgiWESdovIq0omPZurM/edit?usp=sharing

Fixed In - https://docs.google.com/document/d/1IQp1ExvJBdgqOw12KA6pbO5jk6z4i7GTdt7kv2bwmiA/edit?usp=sharing

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.

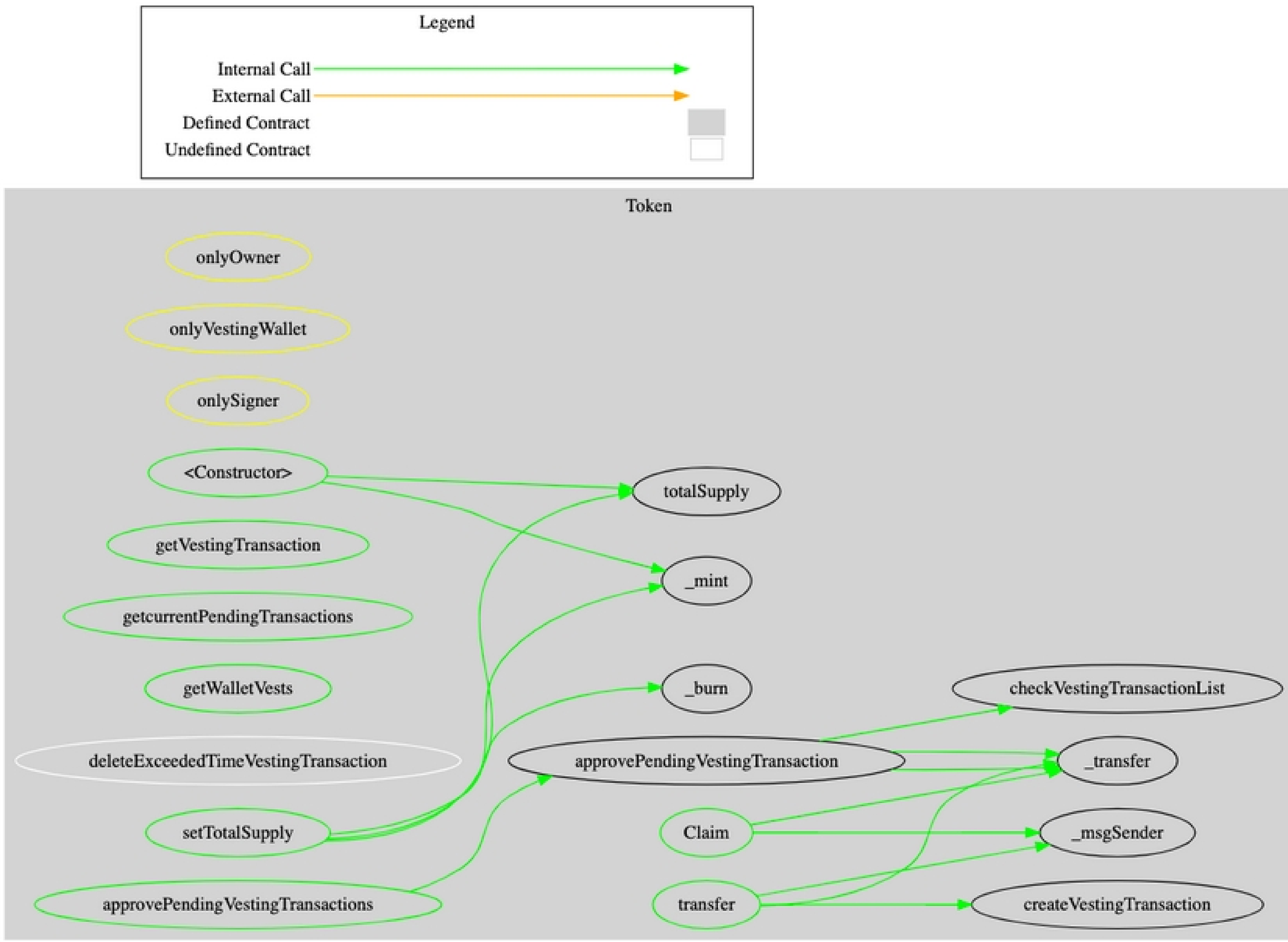
Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

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
Slither, MythX, Truffle, Remix, Ganache, Solidity Metrics

Call Graphs


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Token
onlyOwner
onlyVestingWallet
onlySigner
<Constructor>
getVestingTransaction
getcurrentPendingTransactions
getWalletVests
deleteExceededTimeVestingTransaction
setTotalSupply
approvePendingVestingTransactions
totalSupply
_mint
_burn
approvePendingVestingTransaction
checkVestingTransactionList
_transfer
Claim
_msgSender
transfer
createVestingTransaction

Slither Analysis Results

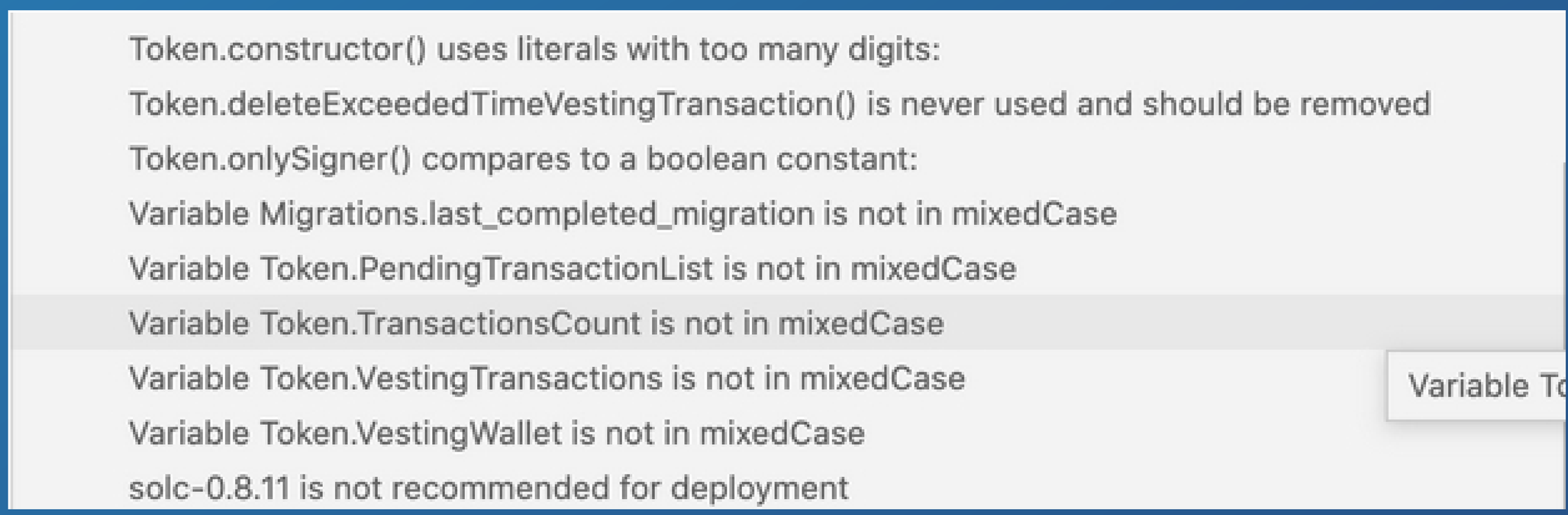
SLITHER

ANALYSIS

- High (0)
- Medium (7)
  - Token.Claim() uses a dangerous strict equality:
  - Token.approvePendingVestingTransaction(uint256) uses a dangerous strict equality:
  - Token.approvePendingVestingTransaction(uint256) uses a dangerous strict equality:
  - Token.checkVestingTransactionList(uint256) uses a dangerous strict equality:
  - Token.deleteExceededTimeVestingTransaction() uses a dangerous strict equality:
  - Token.getWalletVests(address) uses a dangerous strict equality:
  - Token.getcurrentPendingTransactions() uses a dangerous strict equality:

Token.getWalletVests(address) uses a dangerous strict equality:
Token.getcurrentPendingTransactions() uses a dangerous strict equality:

  - Token.Claim() uses timestamp for comparisons
  - Token.approvePendingVestingTransaction(uint256) uses timestamp for comparisons
  - Token.checkVestingTransactionList(uint256) uses timestamp for comparisons
  - Token.deleteExceededTimeVestingTransaction() uses timestamp for comparisons
  - Token.getWalletVests(address) uses timestamp for comparisons
  - Token.getcurrentPendingTransactions() uses timestamp for comparisons
- Informational (24)
  - Function Token.Claim() is not in mixedCase
  - Parameter Migrations.upgrade(address).new_address is not in mixedCase
  - Parameter Token.approvePendingVestingTransaction(uint256)._id is not in mixedCase
  - Parameter Token.checkVestingTransactionList(uint256)._id is not in mixedCase
  - Parameter Token.createVestingTransaction(address,uint256,uint256).Due_Date is not in mix...
  - Parameter Token.createVestingTransaction(address,uint256,uint256)._amount is not in mixe...
  - Parameter Token.createVestingTransaction(address,uint256,uint256)._to is not in mixedCase
  - Parameter Token.getVestingTransaction(uint256)._id is not in mixedCase
  - Parameter Token.setTotalSupply(string,uint256)._amount is not in mixedCase
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...
  - Pragma version^0.8.0 necessitates a version too recent to be trusted. Consider deploying wit...



All output checked for false positives, validity and extract out relevant results only.

MythX Analysis Results

Issues

These were the issues detected in the scan. You can use the toggles to filte

Severity	Low (2)	Medium (0)	High (0)
Ignored Issues	Hidden (0)		

ID	Severity	Name
SWC-103	Low	A floating pragma is set.
SWC-108	Low	State variable visibility is not set.

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	1	1
Closed	0	0	3	8

Functional Testing Results

Complete functional testing report has been attached below:

- ☑ Should be able to create vesting transaction
- ☑ Should be able to approve vesting transaction
- ☑ Should be able to approve multiple transactions
- ☑ Should be able to transfer tokens
- ☑ Should be able to set totalsupply
- ☑ Should be able to claim tokens
- ☑ Should be able to get current vesting transactions
- ☑ Should be able to get wallet vests
- ☑ Should create vesting transaction if transfer function caller is owner
- ☑ Should be able to get current pending transactions
- ☑ Should revert if caller is not owner
- ☑ Should revert if caller is not signer
- ☑ Should revert if caller currently don't has duted transaction while claiming

Issues Found – Code Review / Manual Testing

High severity issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

1. **Missing address verification**

   Certain functions lack a safety check in the address, the address-type argument should include a zero-address test, otherwise, the contract's functionality may become inaccessible, tokens sent to invalid addresses, tokens may be burned in perpetuity or code may behave unexpectedly.

```
function createVestingTransaction(address _to, uint _amount, uint Due_Date) public
onlyOwner returns(uint)

function transfer(address recipient, uint amount) override public returns (bool)
```

   For example in the above function _to and recipient are not checked.

   Remedy
   It is recommended to check addresses are not zero addresses using for example require(_to != address(0)). Consider not only fixing the specific instances mentioned above, but also reviewing the entire codebase to make sure every address input that needs some verification is checked.

   Status: Fixed

2. **Ownership**

   Token Contract is Ownable with the owner having certain privileges like minting new tokens, burning tokens, creating Vesting Transactions. If ownership is controlled by a single party, this poses centralization and other risks. Additionally contracts do not allow for change of ownership. In the event access is lost to current owner contract functionality is compromised.

Remedy
It’s recommended to ensure the owner is not a single entity, consider using a Multisig for the owner. Consider inheriting from the Ownable contracts from OpenZeppelin and consider deactivating renounceOwnership function.

Status: Acknowledged

3. Redundant Code

Code that is not used or applied may indicate errors or missing logic. Consider code below

```
event TransferedFromCurrentSupply(uint from, address indexed to, uint256 value);

function deleteExceededTimeVestingTransaction() internal returns(bool)
```

Remedy
Review above code snippets if not needed consider to remove the code.

Auditor's Response: Reductant code removed

Status: Fixed

4. State variable visibility

Variable without visibility specified. The default visibility for state variables is internal. Specifying visibility adds to code clarity. If the state variable was meant to have a public getter this will not work due to default being internal. Consider the below state variable without visibility specified.

```
uint pendingTransactionID = 1;
```

Remedy
It’s recommended to be explicit about variable visibility by making it public, private or other as required.

Auditor's Response: State variable visibility made private

Status: Fixed

Informational issues

5. Floating pragma

Contract makes use of pragma ^0.8.0 which allows for solidity compiler versions from 0.8.0 to the version just before 0.9.0. This can result in different versions being used for testing and production.

Remedy
It is recommended to deploy contracts using the same compiler version/ flags with which they have been tested. The solidity version must be fixed by locking the pragma by avoiding using ^. Consider using an earlier stable version like pragma 0.8.4

Auditor's Response: Version 0.8.4 used

Status: Fixed

6. SPDX Licence Identifier

A SPDX License identifier is not used and applied at the top of the contract file.

Remedy
It is recommended to make use of a License identifier, consider the example
// SPDX-License-Identifier: MIT

Auditor's Response: // SPDX-License-Identifier: Unlicensed

Status: Fixed

7. **Natspec**

The code is lacking commenting. Comments inline, particularly in Natspec format help to clarify what the code does.

Remedy
It is recommended to use Natspec, a form of comments in Solidity that provides rich documentation for functions, return variables and more.
E.g /// @notice Returns the amount of leaves the tree has.
/// @dev Returns only a fixed number.

Auditor's Response: Natspec added

Status: Fixed

8. **Long number literals**

Long number literals are prone to error and require more checking and when reading can be misread. Consider below in the constructor for 10 billion

_mint(msg.sender, 10000000000 * 10 ** 6);

Remedy
It is recommended to use scientific notation e.g 1e10 * 10 **6 or as required. It may be recommended to make this a state variable for greater transparency.
/// @dev Returns only a fixed number.

Auditor's Response: Scientific notation used

Status: Fixed

9. **Error messages in require**

The error messages in the require statements are too long.

require(msg.sender == owner, "You are not the owner of this smart contract. Contact info@futiracoin.com for help.");

Remedy
It is recommended to shorten the error messages.
/// @dev Returns only a fixed number.

Auditor's Response: Require error messages shortened

Status: Fixed

10. Variables declared as int

Some variables are declared as uint without specifying if uint8, uint16... uint256 is the required value. Although uint defaults to uint256 for better code clarity it is important to specify explicitly. Consider function below uint _id

function checkVestingTransactionList(uint _id) internal view returns(bool)

Remedy
It is recommended to favor explicitness, consider changing all instances of uint into uint256 in the entire codebase.

Status: Fixed

11. Missing test cases

Test cases for the code and functions have not been provided

Remedy
It is recommended to write test cases for all the functions. Any existing tests done if failures must be resolved. Tests will help determine if the code is working in the expected way. Unit tests, functional tests and integration tests should have been performed to achieve good test coverage across the entire codebase.

Status: Acknowledged

12. Public functions that can be made external

Public functions that are not called within the contract can be made external. External functions cost less than public functions.

```
function Claim() public returns (bool)

function getVestingTransaction(uint _id) public view returns(address, uint, string memory, uint)

function getcurrentPendingTransactions() public view returns( uint, uint, uint)

function getWalletVests(address account) public view virtual  returns (uint, uint)

function approvePendingVestingTransactions() onlySigner public returns(bool)

function setTotalSupply(string memory operation, uint _amount) onlyOwner public returns(uint)
```

Remedy
It is recommended to make the above mentioned functions external to save on gas costs. Consider not only fixing the specific instances mentioned above, but also reviewing the entire codebase.

Status: Fixed

13. Missing events

The missing event makes it difficult to track off-chain and impact off-chain monitoring and incident response functionality. An event should be emitted for critical operations, critical functions and significant transactions: Consider the functions below

```
function approvePendingVestingTransactions() onlySigner public returns(bool)

function Claim() public returns (bool)

function createVestingTransaction(address _to, uint _amount, uint Due_Date) public onlyOwner returns(uint)
```

Remedy
Critical updates may need to emit events. We recommend emitting an event to log the update of the above functions or any other functions or operations deemed requiring event emission.

Auditor's Response: Events for updating TotalSupply and Claiming added

Status: Fixed

Closing Summary

Some issues of Low and Informational severity were found in the Initial Audit,which has been fixed by the FutiraCoin Team.

Disclaimer

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

Audit Report
February, 2022

For

FUTIRA

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com