QuillAudits

Audit Report
April, 2022

For

Contents

Contents

Overview

Ten Best Coins

TBC is a coin that tracks and weighs the ten largest crypto coins for one price. The coins were created on the Ethereum ERC-20 blockchain network.

Scope of the Audit

The scope of this audit was to analyse the Ten best coins' smart contract's codebase for quality, security, and correctness.

Time Duration: 25 March, 2022 - 27 March, 2022

Ten Best Coin Contract deployed at:
https://etherscan.io/address/0x03042ae6fcfd53e3a0baa1fab5ce70e0cb74e6fb#code

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
- 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 level

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
Mythril, Slither, SmartCheck, Surya, Solhint, Solidity statistic analysis plugin.

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	7
Closed	0	0	0	0

Functional Testing

Some of the tests performed are mentioned below:

- Should get the name of the contract.
- Should get the symbol of the contract.
- Should get decimals of the contract.
- Should get totalSupply of the contract.
- Should Mint the provided initial supply to the owner account.
- Should check if tokens are transferred.
- Should update balances correctly.
- Should call approve to another account
- Allowance between two accounts should be returned correctly.
- Should test transferFrom function
- Should revert if exceeds the allowance
- Should update the balances of respective accounts correctly
- Should correctly update allowance value.
- Should revert on transfer to zero address.

Issues Found – Code Review/Manual Testing

High severity issues

No issues found

Medium severity issues

No issues found

Low severity issues

1. **Race conditions due to ERC20 Approve function.**

   The standard ERC20 implementation contains a widely-known race condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval, and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the owner’s balance to the spender. If the spender’s transaction is validated before the owner’s, the spender is able to spend their entire approval amount twice.

```
function approve(address spender, uint tokens) external returns(bool success) {
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
}
```

Here is a possible attack scenario: Alice allows Bob to transfer N of Alice's tokens (N>0) by calling approve method on the Token smart contract passing Bob's address and N as method arguments After some time, Alice decides to change from N to M (M>0) the number of Alice's tokens Bob is allowed to transfer, so she calls approve method again, this time passing Bob's address and M as method arguments Bobnotices Alice's second transaction before it was mined and quickly sends another transaction that calls·transferFrom· method to transfer N Alice's tokens somewhere If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain the ability to transfer another M tokens Before Alice noticed that something went wrong, Bob calls ·transferFrom· method again, this time to transfer M Alice's tokens.

So, Alice's attempt to change Bob's allowance from N to M (N>0 and M>0) made it possible for Bob to transfer N+M of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

Recommendation
One possible solution to mitigate this race condition is to first reduce the spender’s allowance to 0 and set the desired value afterwards.

Reference:
1. https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM
2. https://eips.ethereum.org/EIPS/eip-20

Status: Acknowledged

Informational issues

2. **Unlocked pragma (pragma solidity ^0.5.16)**

Contracts should be deployed with the same compiler version and flags that the 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. It is also recommended to use the new stable version of the solidity. 0.5.16 is a relatively older version hence we would recommend you upgrade to the latest version.

Recommendation
Lock the pragma version by removing the caret to lock the file onto a recent Solidity version. Also consider known bugs (*https://github.com/ethereum/solidity/releases*) for the compiler version that is chosen.

Status: Acknowledged

3. **Centralization Risk**

   Ten Best Coins is a fixed supply token and mints 100% of the total tokens liquidity to the deployer of the contract. The implementation is prone to centralization risk that may arise if the deployer loses its private key.

   Recommendation
   Ensure the private keys of the owner account are diligently handled. Alternatively, you can also use a multi-signature wallet to further reduce the risk.

   Status: Acknowledged

4. **Missing zero address validation at multiple instances.**

   There are multiple functions where the address is passed as a function parameter and is not validated against zero address. Adding a zero address check is necessary because, in Ethereum, a zero address is something to which if any funds or tokens are transferred, it can not be retrieved back. If a user mistakenly makes a transfer to the zero address, tokens can be lost forever.

   The checks are missing in the following functions:
   - approve()
   - transfer()
   - transferFrom()

   Recommendation
   Consider adding zero address checks in order to prevent unintentional transfers and approvals.

   Status: Acknowledged

5. **TotalSupply function can be manipulated to display false information**

   As per current implementation, the totalSupply() is a public function that returns (25 million - Tokens held by zero address). In the future, if the team decides to develop a contract that heavily relies on the value returned by this function, it can be manipulated by the attacker by purchasing the tokens and transferring the token to a zero address. If it is the intended behavior, it can be left as it is since it is not a major threat as of now.

   Recommendation
   Return the total supply of the token that was set at the time of deployment regardless of the address holding it. Just remove the balances[address(0)] part.

   Status: Acknowledged

6. **Improper implementation of checks.**

   The contract has various functions where there are no require statements to check if certain conditions are met before changing the state. For instance, consider the transfer function. Here there is no check if the user has a sufficient amount of tokens to transfer or not. The transaction is initiated simply without any check. Although the transaction won't succeed due to math errors, it is a poor coding style. It is recommended to add require statements wherever necessary and throw an appropriate error with a specific error message whenever an error is encountered. The same issue exists in other functions as well.

   Recommendation
   It is recommended to add appropriate require statements wherever necessary in order to add an extra check and also revert with an appropriate error message when certain conditions are not met.

   Status: Acknowledged

7. **Inappropriate name of the contract**

   The name of the deployed contract on etherscan is SlashToken which does not seem to align with Ten Best Coins. If it is the intended implementation, it must be left as it is. Else it must be changed to something related to Ten Best Coins.

   Status: Acknowledged

8. **Public functions that could be declared external inorder to save gas.**

   henever a function is not called internally, it is recommended to define them as external instead of public in order to save gas. For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If your function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from calldata directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge.

   Here is a list of function that could be declared external:

   - balanceOf()
   - totalSupply()
   - allowance()
   - approve()
   - Transfer()
   - TransferFrom()

   Status: Acknowledged

Issues Identified Via Automation Testing

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

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

Disclaimer

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

Audit Report
April, 2022

For

10

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com