QuillAudits

Audit Report
August, 2021

Contents

Scope of Audit

The scope of this audit was to analyze and document the Rocket Global smart contract codebase for quality, security, and correctness.

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 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.

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.

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.

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

Every issue in this report has been assigned with 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 to be fixed before moving to a live environment.

Medium level 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 level severity issues

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 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	6	1
Acknowledged	0	0	0	0
Closed	0	0	0	0

Introduction

During the period of **August 16, 2021 to August 25, 2021** - QuillAudits Team performed a security audit for Rocket Global smart contracts.

The code for the audit was taken from the following official link:
https://bscscan.com/address/0xba07134eda453bbefa4c538c81441574fe65db63#code

https://bscscan.com/address/0x5e21d624631caf3a0a0429740bff3bd58746f1b8#code

Issues Found – Code Review / Manual Testing

High severity issues

No issues were found.

Medium severity issues

Low level severity issues

1. **Mismatch and outdated compiler version (both the contracts)**

```
4
5   // SPDX-License-Identifier: Unlicensed
6
7   pragma solidity ^0.8.0;
8
9   /**
10   * @dev Interface of the ERC20 standard as defined in the EIP.
11   */
12  interface IERC20 {
13      /**
```

Description

Using an outdated and mismatched compiler version can be problematic, especially if publicly disclosed bugs and issues affect the current compiler version.
There have been two compiler versions used in RocketVest1 contract.
And in the CoinToken contract outdated compiler has been used.

Remediation

It is recommended to use a recent and constant version of the Solidity compiler, Version 0.8.7.

Status: Open

2. **SWC 103: Floating Pragma (Both the contract)**

```
4
5   // SPDX-License-Identifier: Unlicensed
6
7   pragma solidity ^0.8.0;
8
9   /**
10   * @dev Interface of the ERC20 standard as defined in the EIP.
11   */
12  interface IERC20 {
13      /**
```

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might negatively introduce bugs that affect the contract system.

Remediation

Lock the pragma version and consider known bugs (https://github.com/ethereum/solidity/releases) for the chosen compiler version.
Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.
Reading Link.

Status: Open

3. Using the approve function of the token standard

In contract RocketVest1 → Line no 52
In contract CoinToken → Line no. 55, 269, 420

```
417      * -  owner  cannot be the zero address.
418      * - `spender` cannot be the zero address.
419      */
420     function _approve(address owner, address spender, uint256 amount) internal virtual {
421         require(owner != address(0), "ERC20: approve from the zero address");
422         require(spender != address(0), "ERC20: approve to the zero address");
423
424         _allowances[owner][spender] = amount;
425         emit Approval(owner, spender, amount);
```

```
51       * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
52       *
53       * Emits an {Approval} event.
54       */
55      function approve(address spender, uint256 amount) external returns (bool);
56
57      /**
58       * @dev Moves `amount` tokens from `sender` to `recipient` using the
59       * allowance mechanism. `amount` is then deducted from the caller's
60       * allowance.
61       *
```

Description

The approve function of ERC-20 is vulnerable. Using a front-running attack, one can spend approved tokens before the change of allowance value.
To prevent attack vectors described above, clients should make sure to

create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. However, the contract itself shouldn't enforce it to allow backward compatibility with contracts deployed before.
Detailed reading around it can be found at EIP 20.

Status: Open

4. Potential use of "block.timestamp" as a source of randomness (RocketVest1.sol)

In RocketVest1.sol → Line no. 121-122, 156, 167, 206, 224-227

```
153         ) {
154
155             uint start = _beneficiaries[beneficiary].start;
156             if (start > block.timestamp) return start - block.timestamp;
157             else return 0;
158         }
159
160         function walletEndDate(
161             address beneficiary
162             ) external view returns(
163                 uint seconds_left
164             ) {
165
166             uint release = _beneficiaries[beneficiary].releaseTime;
167             if (release > block.timestamp) return release - block.timestamp;
168             else return 0;
```

Description
Contracts often need access to time values to perform certain types of functionality. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta; however, they are not safe to use for most purposes.
In the case of block.timestamp, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise, the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

Remediation
Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

References
- Safety: Timestamp dependence
- Ethereum Smart Contract Best Practices - Timestamp Dependence
- How do Ethereum mining nodes maintain a time consistent with the network?
- Solidity: Timestamp dependency, is it possible to do safely?

Status: Open

5. Integer Overflow and Underflow (RocketVest1.sol), SWC 101

RocketVest1.sol → Line no. 121, 122, 146, 156, 167

```
163
164         uint release = _beneficiaries[beneficiary].releaseTime;
165         if (release > block.timestamp) return release - block.timestamp;
166         else return 0;
167     }
168
```

Description
An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance, if a number is stored in the uint8 type, it means that the number is stored in an 8 bits unsigned number ranging from 0 to 2^8-1. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.

Remediation
It is recommended to use vetted safe math libraries for arithmetic operations consistently throughout the smart contract system.

References
- Ethereum Smart Contract Best Practices - Integer Overflow and Underflow

Status: Open

6. Use Require statement for multiple checks in transfer event

Description

A function with a public visibility modifier that is not called internally. Changing the visibility level to external increases code readability. Moreover, in many cases, functions with external visibility modifiers spend less gas compared to functions with public visibility modifiers.

The function definition in the file which are marked as public are below:

CoinToken.sol

- increaseAllowance
- decreaseAllowance
- burnfrom
- supportsInterface
- transferAndCall
- transferFromAndCall
- renounceOwnership
- transferOwnership

However, it is never directly called by another function in the same contract or in any of its descendants. Consider marking it as "external" instead.

Remediation

Use the external visibility modifier for functions never called from the contract via internal call. Reading Link.

Status: Open

Informational

7. Open Zeppelin standards

Description

Follow Open Zeppelin standard for token contract making. There were few functions like Pausable and few others which are usually put in token contracts in general as best practice.

Status: Open

Automated Testing

Solhint

Solhint is an open-source project created by https://protofire.io. Its goal is to provide a linting utility for Solidity code. Below are the results.

```
CoinTOken.sol
  134:2  error  Line length must be no more than 120 but current length is 132  max-line-length
  539:2  error  Line length must be no more than 120 but current length is 160  max-line-length
  558:2  error  Line length must be no more than 120 but current length is 156  max-line-length
  577:2  error  Line length must be no more than 120 but current length is 122  max-line-length
  602:2  error  Line length must be no more than 120 but current length is 146  max-line-length
  627:2  error  Line length must be no more than 120 but current length is 133  max-line-length
  651:2  error  Line length must be no more than 120 but current length is 130  max-line-length
  659:2  error  Line length must be no more than 120 but current length is 134  max-line-length

✖ 8 problems (8 errors, 0 warnings)
```

Anchain

Anchain sandbox audits the security score of any Solidity-based smart contract, analyzing the source code of every mainnet EVM smart contract and the 1M + unique, user-uploaded smart contracts. Code has been analyzed there and got the report below for RocketVest1.

Recommendations

- No visibility specified. Defaulting to 'public'.
  See line(s) 106
- Consider using exact language version instead.
  See line(s) 7, 88
- Please keep in mind that private object is still publicly visible on the chain.
  See line(s) 92, 93, 102
- Integer division should be used with caution.
  See line(s) 227
- Underflow or overflow may happen here, consider check boundaries such as assert(n < INT_MAX).
  See line(s) 121, 122, 146, 156, 167

ANCHAIN.AI

Generated by Anchain.AI CAP Sandbox on 23/08/2021

Results

No major issues were found. Some false positive errors were reported by the tool. All the other issues have been categorized above according to their level of severity.

Disclaimer

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

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.

No instances of Integer Overflow and Underflow vulnerabilities or Back-Door Entry were found in the contract, but relying on other contracts might cause Reentrancy Vulnerability.

Numerous issues were discovered during the initial audit. Some issues are still open after the review; It is recommended to fix them.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com