QuillAudits

Audit Report
June, 2021

ORIJIN

Contents

Introduction

During the period of May 31st, 2021 to June 5th, 2021 – QuillAudits Team performed a security audit for Orijin Finance smart contracts. The code for audit was taken from the following link:

https://bscscan.com/address/0x00089e7D29C29e9B289A02F68f0fDB45D33066F0#code

Updated Contract:
https://testnet.bscscan.com/address/0x00089e7d29c29e9b289a02f68f0fdb45d33066f0#code

Overview of Orijin Finance

Orijin Finance enables users to trade, access liquidity pools, and yield farms across multiple blockchains - Matic, Binance Smart Chain, and Ethereum. Orijin makes it easy for anyone to access cross-chain network protocols from one simple user interface and dashboard. We also make it easy for you to visualise AMM LP's, yield farms, and other positions to give you a summary of your current performance.
Orijin will also have a cross-chain IDO token launchpad platform which will enable upcoming innovative projects to access investors.
We allow DeFi users to easily lend and borrow assets and capture untapped yield across multiple blockchains and invest in top projects.

- Hold, Earn & Stake with Orijin
- Access top projects with the Orijin platform
- Access yield & investment opportunities

ORIJIN Token Functionality

Token Metrics

The ORIJIN token ensures that there are transparent incentives between users, tokenholders and the team. We have ensured that the ORIJIN token has strong fundamental value in place, which will build value over time for investors.

Earn by Holding

Simply hold ORIJIN tokens in your wallet and earn staking rewards from each transaction that takes place on our smart contract.

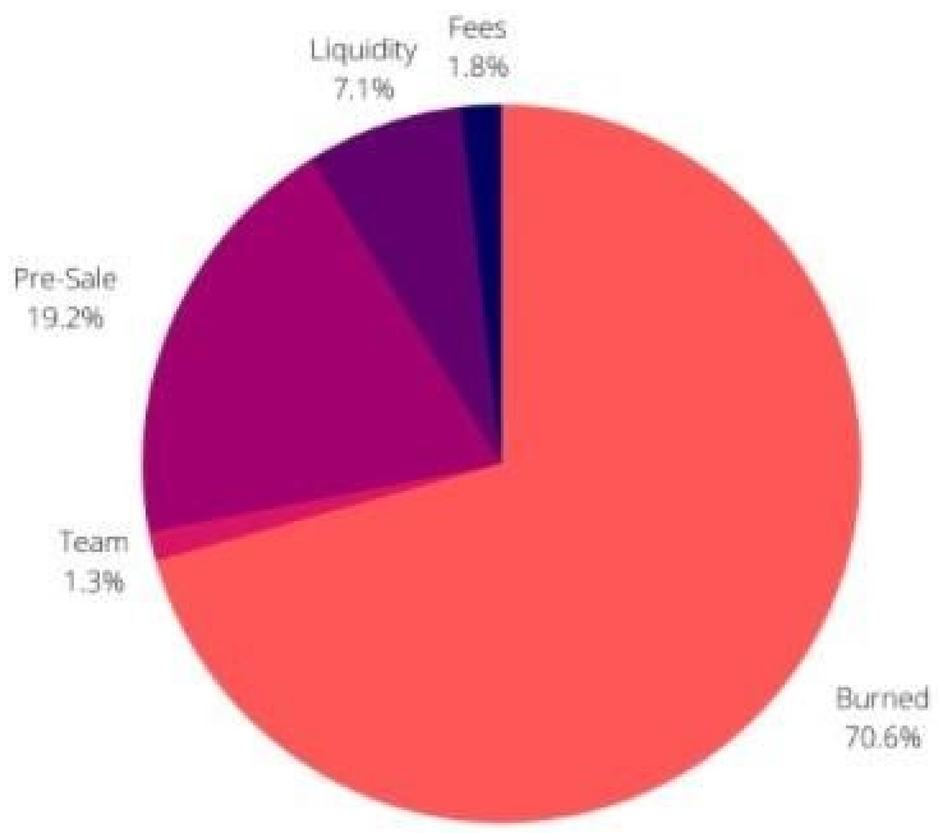
Buyback, Burn & Staking
Every month we will use 50% of all IDO token launchpad revenues to buy-back ORIJIN token from the public market and burn, thereby reducing the supply permanently.

ORIJIN will charge fees for services on Orijin Finance Dashboard, with 90% of fees going to stakers and 10% to the treasury.

Tokenomics

Total Supply: 1,000,000,000,000,000
Hard cap: 1,000 BNB
Soft Cap: 200 BNB
Liquidity Lock: 256 Years (permanent)
Burn: 70% of Total Supply was burned on inception
Liquidity Fee: 5% - Liquidity fee is activated when a sell is activated on Pancakeswap. This is then paired with BNB and added to our locked LP tokens to improve trading conditions.
Transaction Tax Fee: 4% - Each transaction
Marketing Fee: 1% - The treasury has a wallet to support ongoing marketing efforts as we expand Orijin Finance operations globally.
Dev/Team (Locked up): 1.3% will be allocated to the team and locked for 3 months with team finance.

Total Supply: 1,000,000,000,000,000



Details: https://www.orijinfinance.com/
Lite Paper: https://orijinfinance.gitbook.io/welcome-to-orijin-finance/

Scope of Audit

The scope of this audit was to analyse Orijin Finance smart contract codebase for quality, security, and correctness. Following is the list of smart contracts included in the scope of this audit:

- OrijinFinance.sol [OrijinFinance Smart Contract]

OUT-OF-SCOPE: External contracts, External Oracles, other smart contracts in the repository or imported smart contracts.

Checked Vulnerabilities

We have scanned Orijin smart contracts 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 (Unexpected) Throw
- DoS with (Unexpected) revert
- 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
- Address hardcoded
- Using delete for arrays
- Integer overflow/
- underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

- 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 Orijin smart contracts 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 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

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.

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 analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Manticore, Slither.

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

Gas Consumption
In this step, we have checked the behaviour of smart contract in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Steps Followed

- Research into architecture and purpose
- Smart Contract manual code read and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual Assessment of use and safety for the critical solidity variables and functions in scope to identify any arithmetic related vulnerability classes.

- Scanning of solidity files for vulnerabilities, security hotspots, or bugs. (MythX)

- Static Analysis of security for scoped contract and imported functions. (Slither)

- Smart Contract analysis and automatic exploitation (teEther)

- Symbolic Execution / EVM bytecode security assessment (Manticore)

Assessment Summary and Findings Overview

High severity issues

Issues that must be fixed before deployment else they can create major issues.

Medium level severity issues

These issues will not create major issues in working but affect the performance of the smart contract.

Low level severity issues

These issues are more suggestions that should be implemented to refine the code in terms of gas, fees, speed and code accuracy

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

	High	Medium	Low	Informational
Open	0	0	8	3
Closed	0	1	0	0

Findings and Tech Details

High severity issues

No Issue found under this category

Medium severity issues

1. **Flatten version of the contract fails to compile on REMIX**

   Flatten version of the contract failed to compile on REMIX as different compiler versions were used across the contracts. It is recommended to use a fixed compiler version across all contracts. Change the compiler version to 0.6.6.
   Code Lines: 12

```
Orijin
Finance/OriginToken.sol:12:1:
ParserError: Source file requires
different compiler version
(current compiler is
0.6.6+commit.6c089d02.Emscripten.
clang - note that nightly builds
are considered to be strictly
less than the released version
pragma solidity ^0.6.12; ^-------
--------------^
```

   Status: Fixed

Low level severity issues

2. **The compiler version should be fixed**

   Solidity source files indicate the versions of the compiler they can be compiled with. It's recommended to lock the compiler version in code, as future compiler versions may handle certain language constructions in a way the developer did not foresee. It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. **Any fixed compiler version 0.6.6 to 0.6.12 can be used.**
   Code Lines: 12

   Status: Acknowledged by the auditee

3. **Consider using underscores to aid readability**

   Underscores can be used to separate the digits of a numeric literal to aid readability.
   Code Lines: 708, 735, 736

   Status: Acknowledged by the auditee

4. **Consider declaring _tTotal as constant**

   The value of _tTotal is hardcoded and will never change. Consider it declaring as constant instead.
   Code Lines: 708

   Status: Acknowledged by the auditee

5. **Consider declaring _name, _symbol and _decimals as constant**

   The value of _name, _symbol and decimals are hardcoded and will never change. Consider it declaring as constant instead.
   Code Lines: 712-714

   Status: Acknowledged by the auditee

6. **Consider declaring numTokensSellToAddToLiquidity as constant**

   The value of numTokensSellToAddToLiquidity is hardcoded and will never change. Consider it declaring as constant instead.
   Code Lines: 735

   Status: Acknowledged by the auditee

7. **Implicit Visibility**

   It’s a good practice to explicitly define the visibility of state variables, functions, interface functions and fallback functions The default visibility of state variables – internal; function – public; interface function - external. Consider declaring “bool inSwapAndLiquify” as “bool public inSwapAndLiquify”.
   Code Lines: 732

   Status: Acknowledged by the auditee

8. **Use external function modifier instead of public**

   The public functions that are never called by contract should be declared external to save gas. List of functions that can be declared external:

   - name() [#770-772]
   - symbol() [#774-776]
   - decimals() [#778-780]
   - totalSupply() [#782-784]
   - transfer(address,uint256) [#791-794]
   - allowance(address,address) [#796-798]
   - approve(address,uint256) [#800-803]
   - transferFrom(address,address,uint256) [#805-809]
   - increaseAllowance(address,uint256) [#811-814]
   - decreaseAllowance(address,uint256) [#816-819]
   - isExcludedFromReward(address) [#821-823]
   - totalFees() [#825-827]
   - deliver(uint256) [#829-836]
   - reflectionFromToken(uint256,bool) [#838-847]
   - excludeFromReward(address) [#855-863]
   - isExcludedFromFee(address) [#971-973]
   - excludeFromFee(address) [#1139-1141]
   - includeInFee(address) [#1143-1145]
   - setSwapAndLiquifyEnabled(bool) [#1185-1188]

   These functions are never directly called by another function in the same contract or in any of its descendants. Consider marking them as "external" instead.

   Status: Acknowledged by the auditee

9. **Consider adding a zero-address validation**

   In function, setmarketingWallet consider adding a check that the address is not zero using require().
   Code Lines: 1174

   Status: Acknowledged by the auditee

Informational

10. Use local variable instead of state variable like length in a loop

For every iteration of for loop - state variables like length of non-memory array will consume extra gas. To reduce gas consumption, it's advisable to use local variable. Store the length of the array in a variable.
Code Lines: 867,869, 928
Links: Gas Limits and loops

Status: Acknowledged by the auditee

11. Approve function of ERC-20 is vulnerable

A security issue called "Multiple Withdrawal Attack" - originates from two methods in the ERC20 standard for approving and transferring tokens. The use of these functions in an adverse environment (e.g., front-running) could result in more tokens being spent than what was intended. This issue is still open on GitHub, and several solutions have been made to mitigate it.
For more details on how to resolve the multiple withdrawal attack, check the following document for details:

- https://drive.google.com/file/d/1skR4BpZ0VBSQICIC_eqRBgGnACf2li5X/view?usp=sharing
- https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit
- https://blog.smartdec.net/erc20-approve-issue-in-simple-words-a41aaf47bca6
- https://github.com/ethereum/EIPs/20#issuecomment-263524729

Status: Acknowledged by the auditee

12. Coding Style Issues

Coding style issues influence code readability and, in some cases, may lead to bugs in future. Smart Contracts have a naming convention, indentation and code layout issues. It's recommended to use Solidity Style Guide to fix all the issues. Consider following the Solidity guidelines on formatting the code and commenting for all the files. It can improve the overall code quality and readability. The order of functions, as well as, the rest of the code layout does not follow the solidity style guide.

Layout contract elements in the following order:

- Pragma statements
- Import statements
- Interfaces
- Libraries
- Contracts

Inside each contract, library or interface, use the following order:

- Type declarations
- State variables
- Events
- Functions

Status: Acknowledged by the auditee

Origin Finance Function List

The following is the list of functions tested and checked for vulnerabilities during audit:

Private:

- _transferBothExcluded(sender: address, recipient: address, tAmount: uint256)
- _reflectFee(rFee: uint256, tFee: uint256)
- _getValues(tAmount: uint256): (uint256, uint256, uint256, uint256, uint256, uint256)
- _getTValues(tAmount: uint256): (uint256, uint256, uint256)
- _getRValues(tAmount: uint256, tFee: uint256, tLiquidity: uint256, currentRate: uint256): (uint256, uint256, uint256)
- _getRate(): uint256
- _getCurrentSupply(): (uint256, uint256)
- _takeLiquidity(tLiquidity: uint256)
- calculateTaxFee(_amount: uint256): uint256
- calculateLiquidityFee(_amount: uint256): uint256
- removeAllFee()
- restoreAllFee()
- _approve(owner: address, spender: address, amount: uint256)
- _transfer(from: address, to: address, amount: uint256)
- swapAndLiquify(contractTokenBalance: uint256)
- swapTokensForEth(tokenAmount: uint256)
- addLiquidity(tokenAmount: uint256, ethAmount: uint256)

- _tokenTransfer(sender: address, recipient: address, amount: uint256)
- _transferStandard(sender: address, recipient: address, tAmount: uint256)
- _transferToExcluded(sender: address, recipient: address, tAmount: uint256)
- _transferFromExcluded(sender: address, recipient: address, tAmount: uint256)

External:

- <<payable>> null()
- includeInReward(account: address)
- enableAllFees()
- disableAllFees()
- setmarketingWallet(newWallet: address)
- setMaxTxPercent(maxTxPercent: uint256)

Public:

- <<event>> MinTokensBeforeSwapUpdated(minTokensBeforeSwap: uint256)
- <<event>> SwapAndLiquifyEnabledUpdated (enabled: bool)
- <<event>> SwapAndLiquify(tokensSwapped: uint256, ethReceived: uint256, tokensIntoLiqudity: uint256)
- <<modifier>> lockTheSwap()
- constructor()
- name(): string
- symbol(): string
- decimals(): uint8
- totalSupply(): uint256
- balanceOf(account: address): uint256
- transfer(recipient: address, amount: uint256): bool
- allowance(owner: address, spender: address): uint256
- approve(spender: address, amount: uint256): bool
- transferFrom(sender: address, recipient: address, amount: uint256): bool
- increaseAllowance(spender: address, addedValue: uint256): bool
- decreaseAllowance(spender: address, subtractedValue: uint256): bool
- isExcludedFromReward(account: address): bool
- totalFees(): uint256
- deliver(tAmount: uint256)

- reflectionFromToken(tAmount: uint256, deductTransferFee: bool): uint256
- tokenFromReflection(rAmount: uint256): uint256
- excludeFromReward(account: address)
- isExcludedFromFee(account: address): bool
- excludeFromFee(account: address)
- includeInFee(account: address)
- setSwapAndLiquifyEnabled(_enabled: bool)

Automated Testing

Slither

Slither is an open-source Solidity static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses.

```
INFO:Printers:
Compiled with solc
Number of lines: 1190 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 10 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 29
Number of informational issues: 68
Number of low issues: 12
Number of medium issues: 1
Number of high issues: 1
ERCs: ERC20

+----------------------+-------------+-------+----------------------+--------------+--------------+
|         Name         | # functions | ERCS  |     ERC20 info       | Complex code |   Features   |
+----------------------+-------------+-------+----------------------+--------------+--------------+
|      SafeMath        |      8      |       |                      |      No      |              |
|      Address         |      7      |       |                      |      No      |   Send ETH   |
|                      |             |       |                      |              |   Assembly   |
| IUniswapV2Factory    |      8      |       |                      |      No      |              |
|   IUniswapV2Pair     |      27     | ERC20 |     ∞ Minting        |      No      |              |
|                      |             |       | Approve Race Cond.   |              |              |
|                      |             |       |                      |              |              |
| IUniswapV2Router02   |      24     |       |                      |      No      | Receive ETH  |
|   OrijinFinance      |      66     | ERC20 |     No Minting       |      No      | Receive ETH  |
|                      |             |       | Approve Race Cond.   |              |   Send ETH   |
|                      |             |       |                      |              |              |
+----------------------+-------------+-------+----------------------+--------------+--------------+
INFO:Slither:OrijinFinance.sol analyzed (10 contracts)
```

Slither didn't raise any critical issue with smart contracts. The smart contracts were well tested and all the minor issues that were raised have been documented in the report. Also, all other vulnerabilities of importance have already been covered in the **Findings and Tech Details** section of the report.

Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.

Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the Findings and Tech Details section of this report.

Manticore

Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. During Symbolic Execution / EVM bytecode security assessment did not detect any high severity issue. All the considerable issues are already covered in the **Findings and Tech Details** of this report.

Closing Summary

Overall, the smart contract code is extremely well documented, follows a high-quality software development standard, contains many utilities and automation scripts to support continuous deployment/ testing/ integration, and does NOT contain any obvious exploitation vectors that QuillAudits was able to leverage within the timeframe of testing allotted. Overall, the smart contracts adhered to ERC20 guidelines. No critical or major vulnerabilities were found in the audit. **Several issues were found and reported during the audit.**

The outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties was performed to achieve objectives and deliverables set in the scope. QuillAudits recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts.

Disclaimer

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

ORIJIN

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com