QuillAudits

Audit Report
August, 2021

BIG league

Contents

Scope of Audit

The scope of this audit was to analyze and document the Big League Token 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	3	1	2	3
Acknowledged	0	0	0	0
Closed	0	0	0	0

Introduction

During the period of **Aug 22, 2021 to Aug 23, 2021** - QuillAudits Team performed a security audit for an ERC20 token smart contract for Big League.

The code for the audit was taken from the following official link: https://bscscan.com/address/0xea01d8d9eacca9996db6bb3377c1fe64308e7328#code

Issues Found – Code Review / Manual Testing

High severity issues

1. Backdoor in transferFrom function
   Line No: 131

```
function transferFrom(address _from, address _to, uint256 _value) public whenNotPaused returns (bool) {
    require(!tokenBlacklist[msg.sender]);
    require(_to != address(0));
    if (allowance[_from][msg.sender] < ~uint256(0)) allowance[_from][msg.sender] -= _value;
    _transfer(_from, _to, _value);
    return true;
}
```

Description
If a user gives an allowance of the maximum value of uint256, the spender can spend an infinite amount of tokens on the user's behalf. Allowance won't change on transfer as it should.

Remediation
Allowance should be deducted on each successful transferFrom; there should be no conditions regarding deduction. If a transfer is successful, the allowance should decrease respectively.

Status: Open

2. Possible denial of service for transfer functionality
   Line No: 167

```
function _sellAndDistributeAccumulatedTKNFee() internal swapping {
    uint256 _amount = ((totalSupply * _SWAPBACK_THRESHOLD[0]) / _SWAPBACK_THRESHOLD[1]);
    if (balanceOf[address(this)] < _amount) return;

    uint256 holdersFee = (_amount * fees.holders) / fees.total;
    _transfer(address(this), holdersWallet, holdersFee);

    uint256 halfLiquidityFee = fees.liquidity / 2;
    uint256 TKNtoLiquidity = (_amount * halfLiquidityFee) / fees.total;
    uint256 amountToSwap = _amount - holdersFee - TKNtoLiquidity;

    if (amountToSwap > 0) {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = WBNB;
        router.swapExactTokensForETHSupportingFeeOnTransferTokens(amountToSwap, 0, path, address(this), block.timestamp);

    } else return;

    uint256 _gotBNB = address(this).balance;
    uint256 BNBtoLiquidity = _gotBNB;

    if (BNBtoLiquidity > 0) {
        router.addLiquidityETH{value: BNBtoLiquidity}(address(this), TKNtoLiquidity, 0, 0, owner, block.timestamp);
    }
}
```

Description
Anyone can add more ether to the contract; if there is a disparity between ether and tokens when adding ether liquidity, it can result in reverting of tx. And all successive transfers will revert until enough tokens/ether are not added to the contract.

Remediation
Ensure token, and ether amount are synchronized before adding liquidity. A better approach would be to ask for a quote from DEX and manage tokens and ether accordingly.

Status: Open

3. Using the approve function of the token standard

Description
Proper events are not getting emitted for many of the critical operations; these are highly sensitive administrative operations as following -
- forbidMint
- setFees
- setHoldersWallet
- setLiquidityThreshold
- setRouter
- setPair
- _sellAndDistributeAccumulatedTKNFee

Events are necessary for tracking changes on blockchain as tracking each transaction is very tedious; events also help as notification in case of administrative compromise and planning mitigation of threat.

Remediation
Emit proper events for crucial operations.

Status: Open

Medium severity issues

4. Improper error handling

Description

Across the codebase, appropriate conditions are not checked, i.e., the balance of the user. If the user tries to transfer more tokens than he has, his transactions will simply fail due to solidity internal assertion of arithmetic overflow/underflow with unexpected revert message. This is consistent throughout the codebase among other functionalities as well, i.e., approve operations, etc. It's good to check for cases in which a transaction could fail and have them revert if it can be failed further down the execution.

Remediation

Properly analyze all pre and post conditions in which a transaction could fail and check them explicitly with appropriate error messages.

Status: Open

Low level severity issues

5. Tokens can be burned at the Owner behest

Line No: 213

```
function burn(uint256 _value) onlyOwner public {
    balanceOf[msg.sender] -= _value;
    totalSupply -= _value;
    emit Burn(msg.sender, _value);
    emit Transfer(msg.sender, address(0), _value);
}
```

Description

Generally, the functionality of minting and burning of tokens goes hand in hand. The MINT_FORBIDDEN is being used to decide whether the contract can mint more tokens or not. However, such restriction is not in the burn function; the owner can burn tokens anytime.

Remediation

Properly analyze requirements and consequences for burning tokens.

Status: Open

6. Lack of unit tests

Description
The code is taken from BscScan, so we are under the assumption that there are no unit tests for the codebase. Contracts meant to handle thousands of user funds should have appropriate testing and coverage.

Remediation
Implement unit tests with at least 98% coverage.

Status: Open

Informational

7. Gas costs optimization

Description
Storage variables can be packed together to reduce storage slot usage and can help in saving gas costs. uint8, bool variables can be clubbed with address variables to save gas.
Each storage slot costs around ~20000 in gas. In the codebase with optimum packing, the savings can be around ~60000 gas.

Remediation
Optimize storage variables to save gas. Few optimizations could be:
- Packing uint8, bool variables together with address variables.
- Across the codebase, uint256 variables have a very small value. Analyze the scope of the highest value possible for these addresses, and if a smaller data type can be used, then use it.

Status: Open

8. State variables that could be declared constants

address public WNB;

Description
The above constant state variable should be declared constant to save gas.

Remediation
Add the constant attributes to state variables that never change after contact creation.

Status: Open

9. Arithmetic calculations are not scaled off-chain

Description
This issue is not a vulnerability but a suggestion to keep in mind while doing administrative operations. Deduction of fees and burning of tokens are being made on each transfer, 1 unit of token is 1E18 in contract, so to calculate one percentage of a certain unit token amount, the parameters of FeeSettings needs to take 1E18 into the amount. This can easily be handled off-chain and certainly does not need to implement on-chain.

Status: Open

Automated Testing

Slither

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 Big League 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 Big League team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Closing Summary

Overall, in the initial audit, there were few high-level issues associated with token transfers. Some other vulnerabilities have also been recorded; It is recommended to fix these.

No instances of Integer Overflow and Underflow vulnerabilities are found in the contract, but relying on other contracts might cause Reentrancy Vulnerability as many addresses are mutable by the owner.

BIG league

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com