QuillAudits

# Audit Report
# May, 2022

For

# Table of Content

# Executive Summary

| | |
|---|---|
| **Project Name** | BscDragon |
| **Overview** | Dragon is a well-built P2E game inspired by the westerndragonmyth, featuring different powerful dragons and legendarykings.It provides an immersive world where players can live togetherwith those creatures of myth in the antediluvian period. The scope of this audit was to test BEP20 token (Dragon) by bscDragon. |
| **Timeline** | 20 May, 2022 - 22 May, 22 |
| **Method** | Manual Review, Functional Testing, Automated Testing etc. |
| **Scope of Audit** | The scope of this audit was to analyse Dragon.sol codebase for quality, security, and correctness. |
| **BscScan** | *https://bscscan.com/address/0x01daae426946a93681525bb97496a3a6279fac5d#code* |


High
Medium
Low
Informational

| | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open Issues | 0 | 0 | 0 | 0 |
| Acknowledged Issues | 0 | 0 | 1 | 4 |
| Partially Resolved Issues | 0 | 0 | 0 | 0 |
| Resolved Issues | 0 | 0 | 0 | 0 |

## Types of Severities

### 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.

## Types of Issues

### Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

### Resolved

These are the issues identified in the initial audit and have been successfully fixed.

### Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

### Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

# Checked Vulnerabilities

- 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
- BEP20 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

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

# Manual Testing

## A. Dragon.sol

### High Severity Issues

No issues were found

### Medium Severity Issues

No issues were found

### Low Severity Issues

#### A.1 Not complying with BEP20 standard completely.

5.1.1.6 getOwner

```
function getOwner() external view returns (address);
```

- Returns the bep20 token owner which is necessary for binding with bep2 token.
- **NOTE** - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

**Description**

As specified in the whitepaper, $ Dragon is supposed to be a BEP20 standard token. BEP20 standard makes it mandatory for the tokens to define a function as getOwner, which should return the owner of the token contract. However, the contract doesn't implement/define any such function, as a result, the token may not flow across the Binance Chain and Binance Smart Chain, as stated by BEP20 interface documentation.

**Remediation**

Consider adding getOwner function.

**Status**

**Acknowledged**

# Informational Issues

## A.2 Multiple unlocked pragmas

**Description**

Contracts should be deployed with the same compiler version and flags that they have been tested 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. Additionally, It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks.

**Remediation**

Lock the pragma version by removing the ^ sign to lock the file onto a specific Solidity version. Moreover, consider using the same solidity compiler version throughout the code.

**Status**

**Acknowledged**

## A.3 Unused code

**Description**

In the contract, there are two functions that are never used. _msgData() and _burn() these two functions never called internally or externally. It is recommended to remove the unused code from the contract.

**Remediation**

Consider removing unused code from the contract.

**Status**

**Acknowledged**

## A.4 Public functions that could be declared external in order to save gas

**Description**

Whenever 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 the function is only called externally, then you should explicitly mark it as external. External functions' parameters are not copied into memory but are read from call data 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 functions that could be declared external:

- name
- symbol
- totalSupply
- balanceOf
- transfer
- approve
- transferFrom
- increaseAllowance
- decreaseAllowance
- renounceOwnership
- transferOwnership

**Remediation**

Consider declaring the above functions as external in order to save some gas.

**Status**

**Acknowledged**

## A.5 Transfer of ownership must be a two-step process

### Description

The transferOwnership() function in the contract allows the current admin to transfer his privileges to another address. However, inside _transferOwnership(), the newOwner is directly stored in the storage, owner, after validating the newOwner is a non-zero address, which may not be enough.

**Line #586**

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);`
}
```

**Line #595**

```
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

As shown in the above code snippets, newOwner is only validated against the zero address. However, if the current admin enters the wrong address by mistake, he would never be able to take the management permissions back. Besides, if the newOwner is the same as the current admin address stored in _owner , it's a waste of gas.

### Remediation

It would be much safer if the transition is managed by implementing a two-step approach: _transferOwnership() and _updateOwnership(). Specifically, the _transferOwnership () function keeps the new address in the storage, _newOwner ,instead of modifying the _owner() directly. The updateOwnership() function checks whether _newOwner is msg.sender, which means _newOwner signs the transaction and verifies himself as the new owner. After that, _newOwner could be set into _owner.

### Status

**Acknowledged**

# Functional Testing

**Some of the tests performed are mentioned below**

- Should test all getters.
- Should test transfer.
- Transfers from/to zero address must revert.
- Should test approve.
- Should test transferFrom.
- Should revert when transfer amount is more than the allowance.
- Should test allowance.
- Should test increaseAllowance.
- Should test decreaseAllowance.
- Should test renounceOwnership and access control on renounceOwnership.
- Should test transferOwnership and access control on transferOwnership.

# Automated Tests

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

In this report, we have considered the security of the BscDragon. We performed our audit according to the procedure described above.

Some issues of Low and informational severity were found, which BscDragon Team Acknowledged

# Disclaimer

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

# About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

**500+**
Audits Completed

**$15B**
Secured

**500K**
Lines of Code Audited

## Follow Our Journey

# Audit Report
# May, 2022

For


QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com