BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for BNB Kingdom Smart Contract

Request Date: 2022-05-22
Completion Date: 2022-05-27
Language: Solidity

Contents

Commission 3
BNB KINGDOM Properties 4
Contract Functions 5
Executables 5
Checklist 6
Owner privileges 8
BNB KINGDOM Smart Contract 8
Testing Summary 13
Quick Stats 14
Executive Summary 15
Code Quality 15
Documentation 15
Use of Dependencies 15
Audit Findings 16
Critical 16
High 16
Medium 16
Low 16
Conclusion 17
Our Methodology 17

Commission

Audited Project	BNB Kingdom Smart Contract
Contract Address	0x2Fe004Ae1b6718b09380F8392aA91CD2d5039B98
Contract Creator	0x5251aab2c0Bd1f49571e5E9c688B1EcF29E85E07
Blockchain Platform	Binance Smart Chain Mainnet

Block Solutions was commissioned by BNB KINGDOM Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

BNB KINGDOM
Properties

Compound Bonus Max Time	10
Compound For No Tax Withdrawal	6
Compound Step	86400 s
Cutoff Step	172800 s
Egg to Hire One Miner	864000
Market Egg Divisor	5
Market Egg Divisor Sell	2
Minimum Investment Limit	0.01 BNB
Wallet Deposit Limit	50 BNB
Contract Balance	0 BNB
Total Compound	0 BNB
Total Referral Bonus	0 BNB
Total Staked	0 BNB
Total Withdrawn	0 BNB

Contract Functions

Executables

i. function blackListWallet(address Wallet, bool isBlacklisted) public
ii. function blackMultipleWallets(address[] calldata Wallet, bool isBlacklisted) public
iii. function BONUS_COMPOUND_STEP(uint256 value) external
iv. function BONUS_DAILY_COMPOUND(uint256 value) external
v. function BONUS_DAILY_COMPOUND_BONUS_MAX_TIMES(uint256 value) external
vi. function BuyLands(address ref) public payable
vii. function CHANGE_OWNERSHIP(address value) external
viii. function CHANGE_DEV1(address value) external
ix. function CHANGE_DEV2(address value) external
x. function CompoundRewards(bool isCompound) public
xi. function fundContract() external payable
xii. function PRC_EGGS_TO_HIRE_1MINERS(uint256 value) external
xiii. function PRC_REFERRAL(uint256 value) external
xiv. function PRC_MARKET_EGGS_DIVISOR(uint256 value) external
xv. function PRC_MARKET_EGGS_DIVISOR_SELL(uint256 value) external
xvi. function SellLands() public
xvii. function setblacklistActive(bool isActive) public
xviii. function SET_WITHDRAWAL_TAX(uint256 value) external
xix. function SET_INVEST_MIN(uint256 value) external
xx. function SET_CUTOFF_STEP(uint256 value) external
xxi. function SET_WITHDRAW_COOLDOWN(uint256 value) external
xxii. function SET_WALLET_DEPOSIT_LIMIT(uint256 value) external
xxiii. function SET_COMPOUND_FOR_NO_TAX_WITHDRAWAL(uint256 value) external
xxiv. function startKingdom(address addr) public payable

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed

Whitepaper-Website-Contract correlation.	Not Checked
Front Running.	Not Checked

Owner privileges

BNB KINGDOM Smart Contract

Owner of this contract set the blacklist active.

```
function setblacklistActive(bool isActive) public{
    require(msg.sender == owner, "Admin use only.");
    blacklistActive = isActive;
}
```

Owner of this contract add/remove the wallet address into blacklist addresses.

```
function blackListWallet(address Wallet, bool isBlacklisted) public{
    require(msg.sender == owner, "Admin use only.");
    Blacklisted[Wallet] = isBlacklisted;
}
```

Owner of this contract add/remove the addresses into blacklist addresses.

```
function blackMultipleWallets(address[] calldata Wallet, bool isBlacklisted) public{
    require(msg.sender == owner, "Admin use only.");
    for(uint256 i = 0; i < Wallet.length; i++) {
        Blacklisted[Wallet[i]] = isBlacklisted;
    }
}
```

Owner of this contract starts the contract by executing this function.

```
function startKingdom(address addr) public payable{
    if (!contractStarted) {
        if (msg.sender == owner) {
            contractStarted = true;
            BuyLands(addr);
        } else revert("Contract not yet started.");
    }
}
```

Current owner of the contract transfers the ownership to another wallet.

```
function CHANGE_OWNERSHIP(address value) external {
    require(msg.sender == owner, "Admin use only.");
    owner = value;
}
```

Owner of this contract update the dev1 address.

```
function CHANGE_DEV1(address value) external {
    require(msg.sender == dev1, "Admin use only.");
    dev1 = payable(value);
}
```

Owner of this contract updates the dev2 address.

```
function CHANGE_DEV2(address value) external {
    require(msg.sender == dev2, "Admin use only.");
    dev2 = payable(value);
}
```

Owner of this contract updates the PRC eggs to hire 1 miner value.

```
function PRC_EGGS_TO_HIRE_1MINERS(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value >= 479520 && value <= 720000); /** min 3% max 12%**/
    EGGS_TO_HIRE_1MINERS = value;
}
```

Owner of this contract updates the PRC referral.

```
function PRC_REFERRAL(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value >= 10 && value <= 100);
    REFERRAL = value;
}
```

Owner of this contract updates the PRC market egg divisor value.

```
function PRC_MARKET_EGGS_DIVISOR(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 50);
    MARKET_EGGS_DIVISOR = value;
}
```

Owner of this contract updates the PRC market eggs divisor sell value.

```
function PRC_MARKET_EGGS_DIVISOR_SELL(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 50);
    MARKET_EGGS_DIVISOR_SELL = value;
}
```

Owner of this contract updates the withdrawal tax value.

```
function SET_WITHDRAWAL_TAX(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 900);
    WITHDRAWAL_TAX = value;
}
```

Owner of this contract updates the bonus daily compound value.

```
function PRC_REFERRAL(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value >= 10 && value <= 100);
    REFERRAL = value;
}
```

Owner of this contract updates the compound bonus maximum times value.

```
function BONUS_DAILY_COMPOUND_BONUS_MAX_TIMES(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 30);
    COMPOUND_BONUS_MAX_TIMES = value;
}
```

Owner of this contract updates the bonus compound steps.

```
function BONUS_COMPOUND_STEP(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 24);
    COMPOUND_STEP = value * 60 * 60;
}
```

Owner of this contract updates the minimum investment limit.

```
function SET_INVEST_MIN(uint256 value) external {
    require(msg.sender == owner, "Admin use only");
    MIN_INVEST_LIMIT = value * 1e18;
}
```

Owner of this contract updates the cutoff step.

```
function SET_CUTOFF_STEP(uint256 value) external {
    require(msg.sender == owner, "Admin use only");
    CUTOFF_STEP = value * 60 * 60;
}
```

Owner of this contract updates the withdraw cooldown values.

```
function SET_WITHDRAW_COOLDOWN(uint256 value) external {
    require(msg.sender == owner, "Admin use only");
    require(value <= 24);
    WITHDRAW_COOLDOWN = value * 60 * 60;
}
```

Owner of this contract updates the wallet deposit limit.

```
function SET_WALLET_DEPOSIT_LIMIT(uint256 value) external {
    require(msg.sender == owner, "Admin use only");
    require(value >= 10);
    WALLET_DEPOSIT_LIMIT = value * 1 ether;
}
```

Owner of this contract updates the compound for no tax withdrawal value.

```
function SET_COMPOUND_FOR_NO_TAX_WITHDRAWAL(uint256 value) external {
    require(msg.sender == owner, "Admin use only.");
    require(value <= 12);
    COMPOUND_FOR_NO_TAX_WITHDRAWAL = value;
}
```

Testing Summary

PASS

Block Solutions Believes
this smart contract pass the security tests.

27 May, 2022

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert () misuse	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is **Well-Secured.** Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.
We found critical, 0 high, 0 medium and 0 low level issues.

Code Quality

The BNB KINGDOM Smart Contract protocol consists of one smart contract. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.
Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a BNB KINGDOM Smart Contract smart contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No Critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Conclusion

The Smart Contract code passed the audit successfully with some considerations to take. There Were no warnings raised. We were given a contract code. And we have used all possible tests based on given objects as files. So, it is good to go for production.
Since possible test cases can be unlimited for such extensive smart contract protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope was manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report.
Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is "Well Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally, follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating

our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.