QuillAudits

Audit Report
January, 2023

For

WeSleep

Table of Content

Executive Summary

Project Name	WeSleep
Overview	WeSleep contract is a token creation contract that mints a total of 1,000,000,000 WEZ tokens to the deployer of the contract. The contract is made up of four critical functions that allows for approval and transfer of tokens by both immediate users and spenders. It also has some public functions generated automatically by the inherent features of public state variables.
Timeline	4 January, 2023 - 10 January, 2023
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse Beimagine codebase for quality, security, and correctness.
Commit Hash	*https://testnet.bscscan.com/address/0xf9353b51210a7477928959bac92698ae9271c62b#code*
Fixed in	*https://testnet.bscscan.com/address/0xbf6eff879be1cbe27d3f6305ff714a01391618b7#code*

High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	1	1	2

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
- 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 leve

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 Analysis

A. Contract - Token.sol

High Severity Issues

No issues were found

Medium Severity Issues

A.1 Frontrunning / Race Condition Possibilities

Description

The token contract is written from scratch up and comes with few basic features that allows for transfer of tokens by token owners and also allows token owners to approve spenders to expend their tokens, supposing that spenders are granted enough amounts for spending. However, there are possibilities of front running and race conditions. If a token owner first approves a spender a certain amount and intends to change it later in the future, say to a lesser value. A spender who is aware of this can frontrun the token owner and spend up to the first approval before the changes of approval price is effected.

Remediation

Use standard libraries that have resolved such issues in their audited libraries. *Reference*.

Status

Resolved

Low Severity Issues

A.2 Missing Address Zero Check

```
ftrace | funcSig
26    function transfer(address to↑, uint value↑) public returns(bool) {
27        require(balanceOf(msg.sender) >= value↑, 'balance too low');
28        balances[to↑] += value↑;
29        balances[msg.sender] -= value↑;
30        emit Transfer(msg.sender, to↑, value↑);
31        return true;
32    }
33
ftrace | funcSig
34    function transferFrom(address from↑, address to↑, uint value↑) public returns(bool) {
35        require(balanceOf(from↑) >= value↑, 'balance too low');
36        require(allowance[from↑][msg.sender] >= value↑, 'allowance too low');
37        balances[to↑] += value↑;
38        balances[from↑] -= value↑;
39        emit Transfer(from↑, to↑, value↑);
40        return true;
41    }
42
ftrace | funcSig
43    function approve(address spender↑, uint value↑) public returns (bool) {
44        allowance[msg.sender][spender↑] = value↑;
45        emit Approval(msg.sender, spender↑, value↑);
46        return true;
47    }
48
```

Most of the critical functions that make up the contracts fail to check for the possibilities of receiving address zero parameters. This implies that tokens can be sent to the null address which is afterwards irredeemable.

Remediation

Add relevant checks to prevent the loss of tokens to the null address.

Status

Resolved

Informational Issues

A.3 Unlocked pragma (pragma solidity ^0.8.2

Description

Contract has a floating solidity pragma version. Locking the pragma version helps to ensure that the contract does not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. The recent solidity pragma version also possesses its own unique bugs.

Recommendation

Making the contract use a stable solidity pragma version prevents bugs occurrence that could be ushered in by prospective versions. It is recommended, therefore, to use a fixed solidity pragma version while deploying to avoid deployment with versions that could expose the contract to attack.

Status

Resolved

A.4 General Recommendation

WeSleep contract is a token contract with basic features that allows the transfer of WEZ token, however there are possibilities of front running and issues of race condition. It is recommended to use standard libraries that have been audited and aids in token creation. These libraries also have extra features in contract to prevent issues likely to happen in token contracts created from scratch up.

Functional Testing

Some of the tests performed are mentioned below

- Should get the name of the token
- should get the symbol of the token
- should get the symbol of the token
- should get the total supply of the token when deployed
- should get balance of the owner when contract is deployed
- should transfer tokens to other address
- should approve another account to spend tokens

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 WeSleep. We performed our audit according to the procedure described above.

Some issues of Medium, Low and Informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the End, We Sleep team Resolved All Issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the WeSleep 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 WeSleep 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.

700+
Audits Completed

$16B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
January, 2023

For

WeSleep

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com