Audit Report

Produced by CertiK

for bosagora

6th Jan, 2020

CertiK Audit Report For BosAgora

Request Date: 2019-12-27
Revision Date: 2020-01-06
Platform Name: Ethereum

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and BosAgora(the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/

Executive Summary

This report has been prepared for BosAgora to discover issues and vulnerabilities in the source code of their smart contracts. A comprehensive examination has been performed, utilizing CertiK's Formal Verification Platform, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Vulnerability Classification

CertiK categorizes issues into three buckets based on overall risk levels:

Critical

Code implementation does not match specification, which could result in the loss of funds for contract owner or users.

Medium

Code implementation does not match the specification under certain conditions, which could affect the security standard by loss of access control.

Low

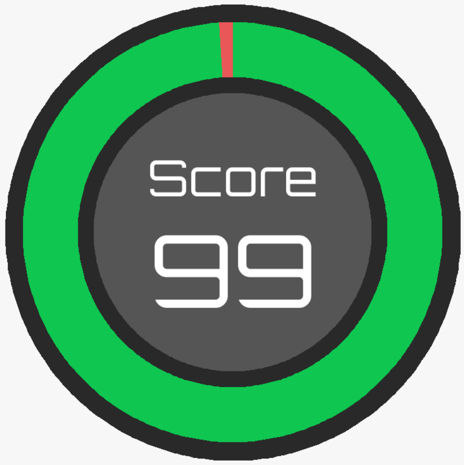
Code implementation does not follow best practices, or uses suboptimal design patterns, which could lead to security vulnerabilities further down the line.

Testing Summary

PASS

CERTIK believes this smart contract passes security qualifications to be listed on digital asset exchanges.

Jan 06, 2020



Type of Issues

CertiK's smart label engine applied 100% formal verification coverage on the source code. Our team of engineers has scanned the source code using proprietary static analysis tools and code-review methodologies. The following technical issues were found:

Title	Description	Issues	SWC ID
Integer Overflow/ Underflow	An overflow/underflow occurs when an arithmetic operation reaches the maximum or minimum size of a type.	0	SWC-101
Function Incorrectness	Function implementation does not meet specification, leading to intentional or unintentional vulnerabilities.	0	
Buffer Overflow	An attacker can write to arbitrary storage locations of a contract if array of out bound happens	0	SWC-124
Reentrancy	A malicious contract can call back into the calling contract before the first invocation of the function is finished.	0	SWC-107
Transaction Order Dependence	A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	0	SWC-114
Timestamp Dependence	Timestamp can be influenced by miners to some degree.	0	SWC-116
Insecure Compiler Version	Using a fixed outdated compiler version or floating pragma can be problematic if there are publicly disclosed bugs and issues that affect the current compiler version used.	1	SWC-102 SWC-103
Insecure Randomness	Using block attributes to generate random numbers is unreliable, as they can be influenced by miners to some degree.	0	SWC-120
"tx.origin" for Authorization	tx.origin should not be used for authorization. Use msg.sender instead.	0	SWC-115

Title	Description	Issues	SWC ID
Delegatecall to Untrusted Callee	Calling untrusted contracts is very dangerous, so the target and arguments provided must be sanitized.	0	SWC-112
State Variable Default Visibility	Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.	0	SWC-108
Function Default Visibility	Functions are public by default, meaning a malicious user can make unauthorized or unintended state changes if a developer forgot to set the visibility.	0	SWC-100
Uninitialized Variables	Uninitialized local storage variables can point to other unexpected storage variables in the contract.	0	SWC-109
Assertion Failure	The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.	0	SWC-110
Deprecated Solidity Features	Several functions and operators in Solidity are deprecated and should not be used.	0	SWC-111
Unused Variables	Unused variables reduce code quality	0	SWC-131

Vulnerability Details

Critical

No issue found.

Medium

No issue found.

Low

No issue found.

Manual Review Notes

Source Code SHA-256 Checksum

- **BOSAGORA.sol**
  `6b4cd1855ff0c31d9aaba0ff435683703a793ac7a388caafb321ed3b23756126`

Summary

CertiK worked closely with [BosAgora] to audit the design and implementation of its soon-to-be released smart contract. To ensure comprehensive protection, the source code was analyzed by the proprietary CertiK formal verification engine and manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with best practices.

Overall, we found [BosAgora]'s smart contracts to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the contract is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, continually improve the codebase, and perform additional tests before the mainnet release.

Static Analysis Results

INSECURE_COMPILER_VERSION

Line 1 in File BOSAGORA.sol

```
1 pragma solidity ^0.5.0;
```

Only these compiler versions are safe to compile your code: 0.5.10

Formal Verification Results

How to read

Detail for Request 1

transferFrom to same address

Verification date	20, Oct 2018
Verification timespan	395.38 ms
CERTIK *label location*	Line 30-34 in File howtoread.sol

CERTIK *label*

```
30  /*@CTK FAIL "transferFrom to same address"
31      @tag assume_completion
32      @pre from == to
33      @post __post.allowed[from][msg.sender] ==
34  */
```

Raw code location	Line 35-41 in File howtoread.sol

Raw code

```
35  function transferFrom(address from, address to
        ) {
36      balances[from] = balances[from].sub(tokens
37      allowed[from][msg.sender] = allowed[from][
38      balances[to] = balances[to].add(tokens);
39      emit Transfer(from, to, tokens);
40      return true;
41  }
```

Counterexample — This code violates the specification

Initial environment

```
1   Counter Example:
2   Before Execution:
3       Input = {
4           from = 0x0
5           to = 0x0
6           tokens = 0x6c
7       }
8       This = 0

53              balance: 0x0
54          }
55      }
56
```

Post environment

```
57  After Execution:
58      Input = {
59          from = 0x0
60          to = 0x0
61          tokens = 0x6c
```

Formal Verification Request 1

If method completes, integer overflow would not happen.

06, Jan 2020
146.23 ms

Line 16 in File BOSAGORA.sol

```
16   //@CTK NO_OVERFLOW
```

Line 30-32 in File BOSAGORA.sol

```
30   constructor () public ERC20Detailed("BOSAGORA", "BOA", DECIMALS) {
31       _mint(msg.sender, INITIAL_SUPPLY);
32   }
```

✓ The code meets the specification.

Formal Verification Request 2

Buffer overflow / array index out of bound would never happen.

06, Jan 2020
10.48 ms

Line 17 in File BOSAGORA.sol

```
17   //@CTK NO_BUF_OVERFLOW
```

Line 30-32 in File BOSAGORA.sol

```
30   constructor () public ERC20Detailed("BOSAGORA", "BOA", DECIMALS) {
31       _mint(msg.sender, INITIAL_SUPPLY);
32   }
```

✓ The code meets the specification.

Formal Verification Request 3

Method will not encounter an assertion failure.

06, Jan 2020
10.54 ms

Line 18 in File BOSAGORA.sol

```
18   //@CTK NO_ASF
```

Line 30-32 in File BOSAGORA.sol

```
30   constructor () public ERC20Detailed("BOSAGORA", "BOA", DECIMALS) {
31       _mint(msg.sender, INITIAL_SUPPLY);
32   }
```

✓ The code meets the specification.

Formal Verification Request 4

ERC20Detailed

06, Jan 2020
14.52 ms

Line 19-29 in File BOSAGORA.sol

```
19    /*@CTK "ERC20Detailed"
20      @tag assume_completion
21      @pre _totalSupply == 0
22      @pre _balances[msg.sender] == 0
23      @post msg.sender != address(0)
24      @post __post._name == "BOSAGORA"
25      @post __post._symbol == "BOA"
26      @post __post._decimals == 7
27      @post __post._totalSupply == 5421301301958463
28      @post __post._balances[msg.sender] == 5421301301958463
29     */
```

Line 30-32 in File BOSAGORA.sol

```
30    constructor () public ERC20Detailed("BOSAGORA", "BOA", DECIMALS) {
31        _mint(msg.sender, INITIAL_SUPPLY);
32    }
```

The code meets the specification.

Source Code with CertiK Labels

File BOSAGORA.sol

```
pragma solidity ^0.5.0;

//
// Copyright (c) 2019 BOS Platform Foundation
//
// https://github.com/bpfkorea/bosagora-erc20
//

import "../openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
import "../openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";

contract BOSAGORA is ERC20, ERC20Detailed {
    uint8 public constant DECIMALS = 7;
    uint256 public constant INITIAL_SUPPLY = 5421301301958463;

    //@CTK NO_OVERFLOW
    //@CTK NO_BUF_OVERFLOW
    //@CTK NO_ASF
    /*@CTK "ERC20Detailed"
      @tag assume_completion
      @pre _totalSupply == 0
      @pre _balances[msg.sender] == 0
      @post msg.sender != address(0)
      @post __post._name == "BOSAGORA"
      @post __post._symbol == "BOA"
      @post __post._decimals == 7
      @post __post._totalSupply == 5421301301958463
      @post __post._balances[msg.sender] == 5421301301958463
     */
    constructor () public ERC20Detailed("BOSAGORA", "BOA", DECIMALS) {
        _mint(msg.sender, INITIAL_SUPPLY);
    }
}
```