CERTIK

CertiK Audit Report for Cojam

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 Cojam (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, CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that projects are checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and assessments to each project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor. CertiK customizes its engineering tool kits, while applying cutting-edge research on smart contracts, for each client on its project to offer a high quality deliverable. For more information: https://certik.io.

Executive Summary

This report has been prepared for **Cojam** to discover issues and vulnerabilities in the source code of their **ERC-20 Smart Contract** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, 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.

Testing Summary

SECURITY LEVEL

Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Cojam.

This audit was conducted to discover issues and vulnerabilities in the source code of Cojam's ERC-20 Smart Contract.

TYPE	Smart Contract
SOURCE CODE	https://etherscan.io/address/0x5c1209acd4fae170eea93c830a0cb74bc5f9ef2b#code
PLATFORM	EVM
LANGUAGE	Solidity
REQUEST DATE	Aug 14, 2020
DELIVERY DATE	Aug 23,, 2020
METHODS	A comprehensive examination has been performed using Dynamic Analysis, Static Analysis, and Manual Review.

Review Notes

Introduction

CertiK team was contracted by the Cojam team to audit the design and implementation of their Cojam token smart contract and its compliance with the EIPs it is meant to implement.

The audited source code link is:

- Token Source Code: https://etherscan.io/address/0x5c1209acd4fae170eea93c830a0cb74bc5f9ef2b#code

The goal of this audit was to review the Solidity implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

The findings of the initial audit have been conveyed to the team behind the contract implementations and the source code is expected to be re-evaluated before another round of auditing has been carried out.

Documentation

The sources of truth regarding the operation of the contracts in scope were minimal although the token fulfilled a simple use case we were able to fully assimilate. To help aid our understanding of each contract's functionality we referred to in-line comments and naming conventions.

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Cojam team or reported an issue.

Summary

The codebase of the project is a typical EIP20 implementation with additional support for a locking mechanism.

Certain optimization steps that we pinpointed in the source code mostly referred to coding standards and inefficiencies.

The codebase of the project strictly adheres to the standards and interfaces imposed by the OpenZeppelin open-source libraries and as such its typical ERC-20 functions **can be deemed to be of high security and quality**.

Recommendations

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report **to achieve a high standard of code quality and security**.

Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Unlocked Compiler Version	Language Specific Issue	Informational	All “pragma” statements

[INFORMATIONAL] Description:

The smart contract “pragma” statements regarding the compiler version indicate that version 0.6.2 or higher should be utilized.

Recommendations:

We advise that the compiler version is locked at version 0.6.2 or whichever Solidity version higher than that satisfies the requirements of the codebase as an unlocked compiler version can lead to discrepancies between compilations of the same source code due to compiler bugs and differences.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
Inefficient Greater-Than Comparison w/ Zero	Optimization	Informational	Cojam: L274

[INFORMATIONAL] Description:

The lines above conduct a greater-than “>” comparison between unsigned integers and the value literal “0”.

Recommendations:

As unsigned integers are restricted to the positive range, it is possible to convert this check to an inequality “!=” reducing the gas cost of the functions.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Unnecessary Struct	Optimization	Informational	Cojam: L437 - L439

[INFORMATIONAL] Description:

The “AddressSet” struct is meant to retain a mapping to indicate whether an address is locked and unable to transfer funds.

Recommendations:

We advise changing the “struct” to a “mapping”, rather than having a “struct” containing a “mapping”.

Example:

mapping(address => bool) internal lockedUsers;

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
Initial Supply Calculation	Coding Style	Informational	Cojam: L491

[INFORMATIONAL] Description:

The initial supply of the token is calculated by offsetting 5 billion by 18 decimal places, the number of decimals supported by the token, but in a hard-coded way.

Recommendations:

We advise that the initial supply is directly calculated by a “constant” variable, as the token retains the number of decimals.

Example:

```
uint256 initialSupply = 5000000000 * (10 ** uint(decimals()));
```

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
Error Messages	Coding Style	Informational	Cojam: L185, L200, L239, L256, L293, L355, L360, L361, L362, L368, L375, L376

[INFORMATIONAL] Description:

It is a generally accepted coding practice to add descriptive error messages to all types of “require” invocations to aid in the debugging of the application.

Recommendations:

We advise that proper error messages are provided for these statements.