DEFIMOON

be secure

Smart Contract Audit Report

July, 2022

CNX Token

DEFIMOON PROJECT

Audit and Development

CONTACTS

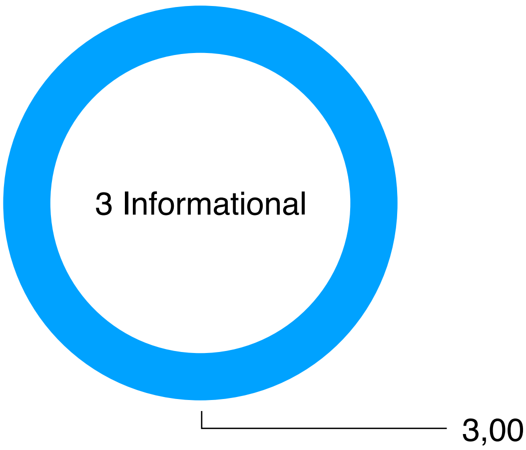
https://defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

be secure

Project information

Name	CNX Token
Audited by	Artur Makhnach
Approved by	Cyrill Minyaev
Description	BEP20 Token
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Whitepaper
Documentation quality	High
Contract address	0xAa55e7A4746644b3420142dc2823F192Bc509073
Chain	BNB Smart Chain



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer3

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

No security issues were found.

However some points should be taken into the consideration.
Those findings represent a "good to know while interacting with the project" information, but don’t directly damage the project in its current state.

The CoinX BEP20 is a slight modification to the BEP20 standard code written by OpenZeppelin. A number of small issues are present, one of them unavoidable concern of using a token on EVM-based chain (DFM-3). DFM-1 and DFM-2 are avoidable and can be fixed. Some best practices
could also be implemented.

Defimoon has only audited the CoinX.sol contract and did not audit the contracts outside of the BEP20 directory.

Summary of findings

According to the standard audit assessment, the audited solidity smart contract is well-secured. The contract has successfully passed the audit.

ID	**Description**	**Severity**
DFM-1	Greedy Contract	Informational
DFM-2	Allowance Double-Spend Exploit	Informational
DFM-3	Race Conditions / Front-Running	Informational

Application security 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
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed
Safe OpenZeppelin contracts and implementation usage	Passed
Front Running	Passed

Detailed Audit Information

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
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Detailed Audit Information

Gas Optimization

Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Findings

DFM-1 «Greedy Contract»

Severity: Informational

Description: A greedy contract is a contract that can receive ether which can never be redeemed.

Recommendation:
In accordance with best practices, to prevent tokens being accidentally stuck in the BEP20 contract itself, it is recommended to prevent the transferal of tokens to the contracts address. This can be achieved, by i.e. adding require statements to transfer functions, similar to `require(to != address(this));`

DFM-2 «Allowance Double-Spend Exploit»

Severity: Informational

Description: As it presently is constructed, the contract is vulnerable to the allowance double-spend exploit, as with other BEP20 tokens.

Exploit Scenario:

1. Alice allows Bob to transfer N amount of Alice's tokens (N>0) by calling the `approve()` method on Token smart contract (passing Bob's address and N as method arguments)

2. After some time, Alice decides to change from N to M (M>0) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve()` method again, this time passing Bob's address and M as method arguments.

3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls the `transferFrom()` method to transfer Alice's tokens somewhere.

4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens.

5. Before Alice notices any irregularities, Bob calls `transferFrom()` method again, this time to transfer M Alice's tokens.

Recommendation:
The exploit (as described above) is mitigated through use of function `safeApprove()`.
Pending community agreement on an ERC standard that would protect against this exploit, we recommend that developers of applications dependent on `approve()` / `transferFrom()` should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value.

DFM-3 «Race Conditions / Front-Running»

Severity: Informational

Description: A block is an ordered collection of transactions from all around the network. It's possible for the ordering of these transactions to manipulate the end result of a block. A miner
attacker can take advantage of this by generating and moving transactions in a way that benefits themselves.

Recommendation:
Make sure users are aware of this ubiquitous EVM-based chains issue.

Automated Analyses

Slither

Slither has reported 18 findings. These results were either related to code from dependencies, false positives or have been integrated in the findings or best practices of this report.

Adherence to Best Practices

1. Duplication of functionality. `getOwner()` function calls `owner()` while that function has `public` visibility.

2. A number of functions can be declared as `external` which will save some gas:
   - `renounceOwnership()`
   - `transferOwnership()`
   - `increaseAllowance()`
   - `decreaseAllowance()`
   - `mint()`

3. Never used functions which should be deleted:
   - `_burn()`
   - `burnFrom()`

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is 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.

Vulnerability Analysis

Our audit techniques include 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, review open issue tickets, and 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 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 to make a final decision.

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.

Appendix A — Finding Statuses

Closed	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed