



Smart Contract Audit Report

August, 2022


CFP

DEFIMOON PROJECT

Audit and
Development


CONTACTS

<https://defimoon.org>
audit@defimoon.org

 [defimoon_org](#)

 [defimoonorg](#)

 [defimoon](#)

 [defimoonorg](#)

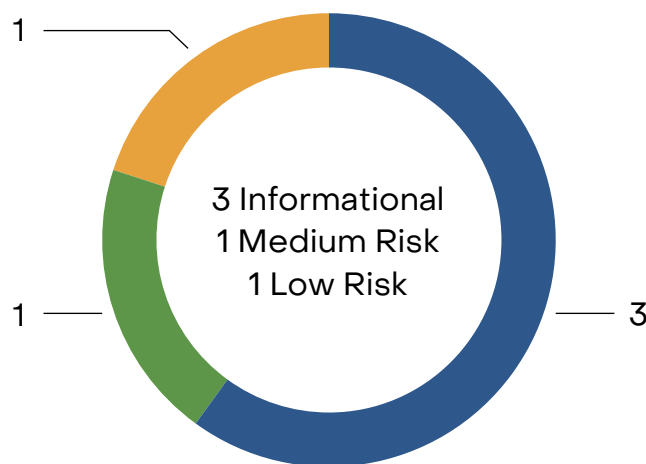


August 31th 2022

This audit report was prepared by Defimoon for CFP

Audit information

Description	The contract implements the ERC-20 type token and DEX interaction mechanics
Audited files	coin.sol, erc20.sol
Timeline	31 August
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://bscscan.com/address/0xd385b0df871472f92067b56b999fc4cc21f4c5a0
Chain	Binance smart chain
Status	Passed



	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.

Disclaimer

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.

Check list

Description	Status
No mint function found, owner cannot mint tokens after initial deploy	✓
Owner can't set max tx amount	✓
Owner can't set fees over 25%	✓
Owner can't pause trading	✓
Owner can't blacklist wallets	✓

Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are fairly secure but are not ready for production.

ID	Description	Severity	Status
<u>DFM-1</u>	Not using proven tools	Medium Risk	resolved
<u>DFM-2</u>	Unsafe Allowance	Medium Risk	acknowledged
<u>DFM-3</u>	Unmanaged fallback	Low Risk	acknowledged
<u>DFM-4</u>	Old solidity version	Informational	resolved
<u>DFM-5</u>	No "NatSpec" comment format	Informational	acknowledged
<u>DFM-6</u>	Redundant null address variable	Informational	acknowledged
<u>DFM-7</u>	Incorrect initialization of basic variables	Informational	acknowledged

Audit overview

No major security issues were found, however it is strongly advised to resolve remaining issues to improve codebase resilience and project's credibility. We would not recommend using this codebase in the production before remaining issues are solved.

Descriptions of functions do not comply with generally accepted standards called NatSpec, which impairs readability and understanding of the code(DFM-5).

In contract CFPToken you can get rid of the `DEAD` variable because in the solidity language you can use the `address(0)` construct(DFM-6).

It is customary to initialize the variables `name`, `symbol` and `decimals` in tokens of type ERC-20 when deploying in the `constructor` function(DFM-7). Safe methods like `safeApprove` are not used to allow the spending of tokens(DFM-2).

The user may mistakenly send their tokens to the contract, which lowers the credibility of the token, since it will be very difficult to return the lost funds.(DFM-3).

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

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

Gas Optimization

Assert () misuse	Passed
High consumption 'for/while' loop	Passed
High consumption 'storage' storage	Passed
"Out of Gas" Attack	Passed

Findings

DFM-1 «Not using proven tools»

Severity: Medium Risk

Severity: Resolved

Description: Non-use of libraries in the contract can lead to serious security problems, since in the openzeppelin library many of the methods that you use are already implemented, for example Ownable and ERC20.

Recommendations: Download the @openzeppelin/contracts library using the `npm install @openzeppelin/contracts` command and import it into your project.

DFM-2 « Unsafe Allowance»

Severity: Medium Risk

Status: Acknowledged

Description:

`safeApprove` function is not used to manage users allowances, this can potentially lead to funds being hijacked through the manipulations with the user allowance.

Recommendation:

Please use OpenZeppelin library and make sure to use `safeApprove` method to control use allowances. Rewrite the `approve` method in the erc20 token and internally call the `safeApprove` method from the `SafeERC20` library

DFM-3 «Unmanaged fallback»

Severity: Low Risk

Status: Acknowledged

Description:

Contracts are missing the `fallback` method, which means that any funds sent to the contract by mistake could potentially be lost permanently for the user (if the contract owner refuses to issue them back), which can lower the credibility of the contract.

Recommendation:

Please implement a `fallback` function.

DFM-4 «Old solidity version»

Severity: Informational

Severity: Resolved

Description: When using the old version of solidity, there are many problems such as number overflow, which is a reason to use additional libraries and, accordingly, adds gas when deploying and using the contract.

Recommendations: Update the solidity version to at least version `0.8.0` and also remove the unnecessary `SafeMath` library since it is not up-to-date and remove all `add`, `sub`, `div`, `mul` etc. functions

DFM-5 «No "NatSpec" comment format»

Severity: [Informational](#)

Status: [Acknowledged](#)

Description: Documentation and commenting in the current contract is not standardized. It is not informative enough, which makes the code difficult to read. Most importantly, it also don't follow the semantic rules required for the web3 applications (blockchain explorers) to process contracts properly.

Recommendation:

It is recommended at least to add attributes in comments such as "@notice", "@param".

You can read about it [here](#).

DFM-6 « Redundant null address variable»

Severity: [Informational](#)

Status: [Acknowledged](#)

Description: address DEAD = 0x00dEaD is redundant, extra variables add to the code complexity and use extra memory in EVM stack, which goes against the principle of code simplicity and optimization.

Recommendation: Use `address(0)` instead, no need for extra variables.

DFM-7 « Incorrect initialization of basic variables»

Severity: [Informational](#)

Status: [Acknowledged](#)

Description:

Variables `name`, `symbol` and `decimals` in the ERC-20-type token are not initialized when deploying in the `constructor` function.

Recommendation:

Please initialize those variables and even better, consider using OpenZeppelin library.

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