



# Smart Contract Audit Report

August, 2022




---


DEFIMOON PROJECT

Audit and  
Development


## CONTACTS

<https://defimoon.org>  
[audit@defimoon.org](mailto:audit@defimoon.org)

 [defimoon\\_org](#)

 [defimoonorg](#)

 [defimoon](#)

 [defimoonorg](#)

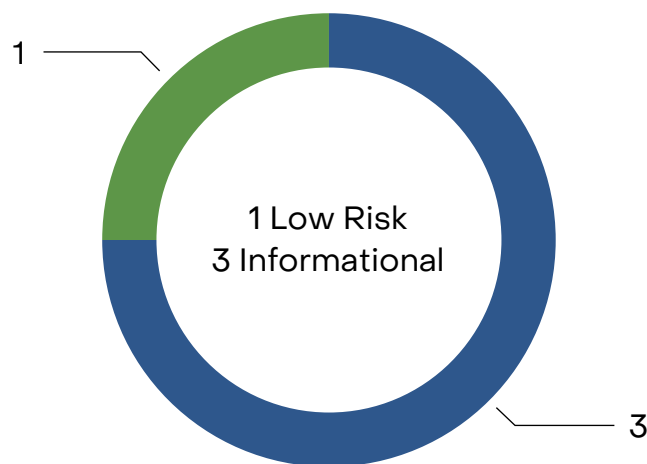


August 25st 2022

This audit report was prepared by Defimoon for SuperPad

### Audit information

Description	The contract implements the Initial DEX Offering
Audited files	IDO.sol, BEP20.sol
Timeline	27th July - 25th August 2022
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Specification	<u><a href="#">Documentation</a></u>
Docs quality	High
Source code	<u><a href="#">Prasoon180 git repo</a></u> / Commit: <u><a href="#">37f4336</a></u>
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.

# IDO Audit overview

## **No major security issues were found.**

However there are a number of small problems, one of which is the frequent use of `block.timestamp` (DFM-4), which, depending on the network, may lead to a loss of functionality, but this is however unlikely.

The absence of the "NatSpec" comment format adds to the unreadability of the contract code (DFM-7).

Some best practices could also be implemented.

1. Detailed description of errors in requires
2. NatSpec description of functions

# BEP20 Audit overview

## **No major 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 SPAD BEP20 is a slight modification to the BEP20 standard code written by OpenZeppelin. A number of small issues are present, some best practices could also be implemented.

It is recommended to add a description of the errors in the requirements(DMF-17).

For verification, it is better to create a modifier that will be used in each function rather than call `require(_to != 0x00, " Transfer to zero address»);` every time (DFM-12).

## Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are fairly secure and could be considered production ready, however, there are still issues remaining in the codebase. These issues are easily solvable but will improve the codebase noticeably, it is recommended to fix them before the production deployment.

ID	Description	Severity	Status
<u>DFM-1</u>	Incorrect second condition in the functions	High Risk	resolved
<u>DFM-2</u>	Incorrect check for the end of the period	High Risk	resolved
<u>DFM-3</u>	Incorrect conditions in the 'reserve' and 'team' functions	Medium Risk	resolved
<u>DFM-4</u>	Private allowances	Medium Risk	resolved
<u>DFM-5</u>	Incorrect initialization	Medium Risk	resolved
<u>DFM-6</u>	Change the condition in the burn function	Low Risk	resolved
<u>DFM-7</u>	Frequent use of «block.timestamp»	Low Risk	acknowledged
<u>DFM-8</u>	Not using proven tools	Low Risk	resolved
<u>DFM-9</u>	Unused "openzeppelin" libraries in the main contract	Informational	resolved
<u>DFM-10</u>	Allowance Double-Spend Exploit	Informational	acknowledged
<u>DFM-11</u>	Race Conditions / Front-Running	Informational	resolved
<u>DFM-12</u>	Use modifier to check the address	Informational	resolved
<u>DFM-13</u>	Incorrect function visibility	Informational	resolved
<u>DFM-14</u>	Extra check in _transfer function	Informational	resolved
<u>DFM-15</u>	No "NatSpec" comment format	Informational	acknowledged
<u>DFM-16</u>	Tautology in terms of functions	Informational	resolved
<u>DFM-17</u>	Error Descriptions	Informational	acknowledged
<u>DFM-18</u>	Greedy Contract	Informational	resolved

## 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 «Incorrect second condition in the functions»

**Severity:** High Risk

**Status:** resolved

**Description:**

Violates the logic in require, which is why the first time the require function is accessed, it will always be executed. Functions: '\_private', '\_public'

**Recommendation:**

Change the conditional `require(block.timestamp > _TGEforPrivate, "1 WEEK NOT OVER»)` to `require(block.timestamp + 604800 seconds > _TGEforPrivate, "1 WEEK NOT OVER»)`.

## DFM-2 «Incorrect check for the end of the period»

**Severity:** High Risk

**Status:** resolved

**Description:**

The condition will never be met since the left part will always be larger than the right.

**Exploit Scenario:**

1. You can withdraw money without waiting for the end of the period

**Recommendation:**

Since `block.timestamp` is always increasing and `_TGEforReward` will be added to it, it is natural that `block.timestamp + _TGEforReward` will always be more than just `_TGEforReward`.

To get rid of this, you need to change the condition require to `(block.timestamp - _TGEforReward > _TGEforReward)`.

### DFM-3 «Incorrect conditions in the 'reserve' and 'team' functions.»

**Severity:** Medium Risk

**Status:** resolved

**Description:**

Incorrect conditions in the 'reserve' and 'team' functions.

**Recommendation:**

The condition is fulfilled not in a month, but in a year. since `_TGEforReserve` will always be less than `block.timestamp`.

change the condition to `block.timestamp - _TGEforReward > 2592000 seconds`.

### DFM-4 «Private allowances»

**Severity:** Medium Risk

**Status:** resolved

**Description:** A user who is not a coin holder cannot see how many tokens other users are allowed to spend, which gives opacity and distrust to the token.

**Recommendation:** Remove this condition.

## DFM-5 «Incorrect initialization»

**Severity:** Medium Risk

**Status:** resolved

**Description:** In the `tge` function, variables are initialized, which is an incorrect implementation, since if it is not called, the variables will not be initialized, which will be fatal for the further operation of the contract.

**Recommendation:**

The first thing that is recommended to do is to transfer the initialization of variables to the contract constructor.

The second thing that is recommended is to transfer the call of the function to the constructor that simulates the initialization of variables, but this is an unnecessary solution.

The third and most correct thing is to add a certain boot initialized flag, which indicates that the contract has been initialized or not. And also the `IsInitialized` modifier with `require(initialized, "Variables not initialized")`.

## DFM-6 «Change the condition in the burn function»

**Severity:** Low Risk

**Status:** resolved

**Description:** Since there is a condition that the value must be strictly greater, it means that the sender will not be able to burn all his tokens to zero.

**Recommendation:** Change a strict comparison(>) to a non-strict(>=) one

### DFM-7 «Frequent use of "black.timestamp"»

**Severity:** Low Risk

**Status:** acknowledged

**Description:**

The use of "black.timestamp" on the network with the Proof of Work consensus algorithm is manipulated in a manner, but this is not a big problem since it is a rather time-consuming operation and it is extremely unlikely to be affected by this vulnerability

**Recommendation:**

since the logic of the contract is tied to "black.timestamp", I cannot give recommendations due to the lack of an alternative.

### DFM-8 «Not using proven tools»

**Severity:** Low Risk

**Status:** resolved

**Description:** In this contract, you do not use proven tools for your contract, such as ERC20 from the OpenZeppelin library

**Recommendation:** Import «@openzeppelin/contracts/token/ERC20/ERC20.sol» and inherit from ERC20

### DFM-9 «Unused "openzeppelin" libraries in the main contract»

**Severity:** Informational

**Status:** resolved

**Description:**

Some methods use "onlyOwner" which is implemented by the developer, which is not a serious problem, but is not a standard.

**Recommendation:**

It is recommended to import the "Ownable.sol" contract from the "openzeppelin" library and make the current contract inherited from it.

## DFM-10 «Allowance Double-Spend Exploit»

**Severity:** Informational

**Status:** acknowledged

### **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.

**Addition:** Renaming the function will not lead to any changes, meaning that you will import the contract from "`@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol`" and inherit from it and use the built-in `safeApprove` function.

## DFM-11 «Race Conditions / Front-Running»

**Severity:** Informational

**Status:** resolved

**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.

Adherence to Best Practices

1. A number of functions can be declared as **external** which will save some gas: -
  - renounceOwnership()
  - transferOwnership()
  - increaseAllowance()
  - decreaseAllowance()
  - mint()
2. Never used functions should be deleted:
  - \_burn()
  - burnFrom()

## DFM-12 «Use modifier to check the address»

**Severity:** Informational

**Status:** resolved

**Description:** In each function where it is required to check the address for null byte, similar code is repeated.

**Recommendation:** Create a modifier that will accept the address and check that it is not null.

## DFM-13 «Incorrect function visibility»

**Severity:** Informational

**Status:** resolved

**Description:** The use of public modifiers in functions that are not called inside the contract is necessary and not desirable

**Recommendation:** Replace public with external

### DFM-14 «Extra check in \_transfer function»

**Severity:** Informational

**Status:** resolved

**Description:** condition `require(balanceOf[_to] + _value >= balanceOf[_to]);` not relevant for the current version of solidity.

**Recommendation:** Remove this condition.

### DFM-15 «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-16 «Tautology in terms of functions»

**Severity:** Informational

**Status:** resolved

**Description:**

Extra number check.

**Recommendation:**

Delete everything it is checked that the number is greater than zero since `uint` in solidity is always greater than zero. And an extra check adds to the high cost of execution.

## DFM-17 «Error Descriptions»

**Severity:** Informational

**Status:** acknowledged

**Description:** `require` calls are missing error messages. This does not provide information about what happened if an error occurred and, accordingly, complicates the integration of the contract into the platforms.

**Recommendation:** Add understandable error descriptions to `requires`.

## DFM-18 «Greedy Contract»

**Severity:** Informational

**Status:** resolved

**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));`.



## Automated Analyses

### **IDO: Slither**

Slither has reported 59 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.

### **BEP20: Slither**

Slither has reported 23 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. Use "OpenZeppelin" library.
2. Use the correct visibility modifiers.
3. Use "NatSpec" comments format.
4. Code markup also suffers, tabs and spaces in the body of the contract and functions are not correctly placed.

# 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