DEFIMOON

be secure

Smart Contract Audit Report

March, 2023

DEFIMOON PROJECT

Audit and Development

CONTACTS

defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

be secure

March 23st 2023

This audit report was prepared by Defimoon for Moovyio

Audit information

Description	Moovy token and ICO contracts.
Audited files	Project repo
Timeline	21st March 2023 - 23d March 2023
Audited by	Daniil Rashin
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Specification	Whitepaper
Docs quality	N/A
Source code	Github commit 8314344
Network	N/A
Status	Passed

Disclaimer

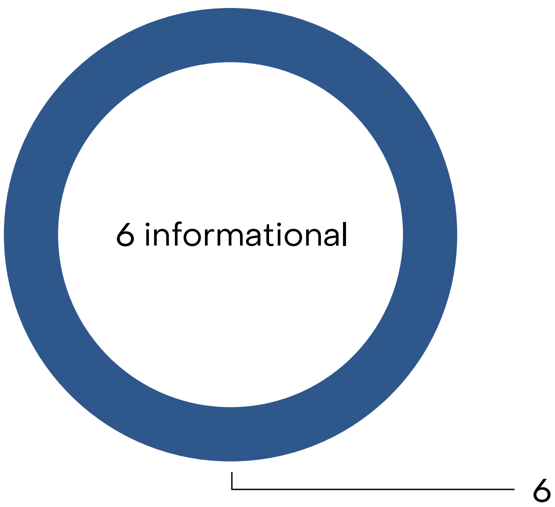
This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, 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.



◆	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.

Moovy contracts Audit overview

Moovy.sol

No major issues were found.
Although no major issues were found, contract may be slightly improved in several ways:

First thing to mention is the solidity version. While it is the most recent compiler version, hardhat does not fully support it yet, which is a matter of time, of course, for them to add full support. Besides, there might be another struggle with contract verification, same with hardhat, it might take some time to add most recent solidity version support.

Another point to improve contract readability is to use keyword ether instead of 10**DECIMALS multiplication. It does not affect any contract logic, but since you use default DECIMALS value, it might be easier to read through the code with ether keyword.

ReentrancyGuard is also redundant here. Reentrancy attacks rely on the receive function, which is triggered upon receiving **native** tokens, allowing recurrent calls to the vulnerable function. In this case, no native currency involved, also beforeTokenTransfer is not overridden, so it has no real use. It is a common practice to perform transfers after all the calculations done, so moving _transfer call right before the event emitting makes nonReentrant modifier needless.

Related findings:

- **DFM-1**

Decimals multiplication.

- **DFM-2**

Redundant reentrancy guard.

- **DFM-3**

Solidity version too recent.

TokenSale.sol

No major issues were found.
Same as with Moovy token, MoovyTokenSale contract contains several 10**DECIMALS multiplications, while there is ether and other built-in keywords. Speaking about payToken, decimals multiplication is valid here if payToken decimals value is not default. Otherwise, using ether fits here too.

ReentrancyGuard is redundant here as well, while payToken is a generic ERC20.

Moovy.sol and ERC20.sol imports may be changed to a single IERC20 interface import as they both use the same functionality defined in IERC20 interface. Also using SafeERC20 library on top of that would make the contract even more secure.

Also it is a good practice to use the same names for a contract and a source file.

Related findings:

- **DFM-1**

Decimals multiplication.
- **DFM-2**

Redundant reentrancy guard.
- **DFM-3**

Solidity version too recent.
- **DFM-4**

Interfaces may be used instead of contract import.
- **DFM-5**

Difference between contract and file name.
- **DFM-6**

SafeERC20 should be used with ERC20 tokens.

Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are secure and ready for production, but there are few ways code may be improved.

ID	Description	Severity
DFM-1	Decimals multiplication	Informational
DFM-2	Redundant ReentrancyGuard	Informational
DFM-3	Solidity versioning	Informational
DFM-4	Interface instead of contract import	Informational
DFM-5	Different contract and source filenames	Informational
DFM-6	SafeERC20 may be used	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

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
Public function could be external	Passed

Findings

DFM-1 «Decimals multiplication»

Severity: Informational

Description: Both contracts use 10**DECIMALS multiplication to calculate token amounts. In case of Moovy token, these values are calculated during the compilation process and does not affect gas usage, MoovyTokenSale performs such calculations dynamically with decimals return value which slightly increase gas usage as it is an additional function call. Most ERC20 tokens use default **18** decimals, so using build-in keywords (ether in this case) make it rather easier to read the contract source code.

Recommendation: Use built-in syntax sugar where possible to improve overall readability.

DFM-2 «ReentrancyGuard»

Severity: Informational

Description: Reentrancy attacks rely on receive function which is triggered upon receiving native currency. While no native currency transfers performed within the contracts, it is redundant to inherit ReentrancyGuard.

Recommendation: Perform outgoing transfers at the end of the functions instead of using nonReentrant modifier.

DFM-3 «Solidity versioning»

Severity: Informational

Description: While using most recent compiler version, it may lead to unexpected behaviour while working with various tools as they might not fully support it yet.

Recommendation: Consider using less recent solidity version which is fully supported by the tools used for project development.

DFM-4 «Interface instead of import»

Severity: Informational

Description: MoovyTokenSale does not really need to import both Moovy token and OZ ERC20. Since both Moovy and payToken are ERC20 tokens and no Moovy-specific functions being used, it is easier to treat them as IERC20 interfaces, thus only a light-weight interface would be included in MoovyTokenSale instead of 2 ERC20 inherited contracts, reducing the deploy cost.

Recommendation: Use IERC20 instead of Moovy and ERC20 imports.

DFM-5 «Different contract and source file name»

Severity: Informational

Description: MoovyTokenSale contract source file is named differently.

Recommendation: Please, follow the solidity styling guide.

DFM-6 «SafeERC20»

Severity: Informational

Description: It is a good practice to use OZ SafeERC20 library when interacting with ERC20 tokens.

Recommendation: Please, add using SafeERC20 for IERC20 to the MoovyTokenSale contract.

Adherence to Best Practices

1. Use more built-in syntax sugar.
2. Use fully supported compiler versions.
3. Use SafeERC20 when interacting with ERC20 tokens.
4. Don't import contracts without real need, use interfaces where possible.
5. Add comments to the source code to make it easier to read through.

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