
# Smart Contract Audit Report

May, 2023

## PekingMuskToken

DEFIMOON PROJECT

Audit and
Development

CONTACTS

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

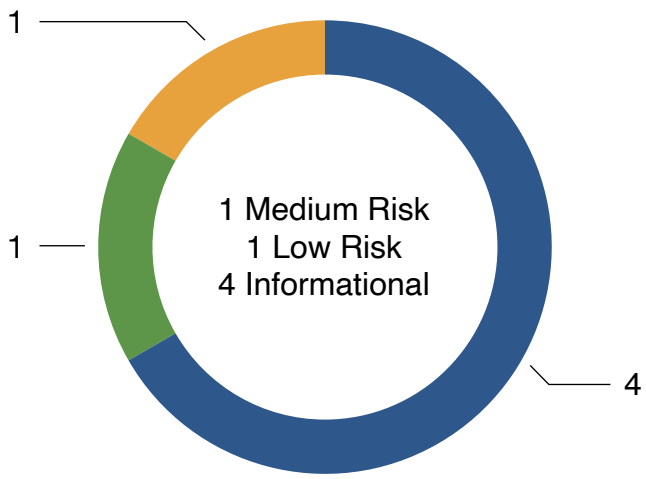
DEFIMOON
be secure

31 May 2023
This audit report was prepared by DefiMoon for PekingMusk.

## Audit information

| | |
|---|---|
| Description | Default ERC20 token smart contract with additional transfer logic |
| Audited files | PekingMuskToken |
| Timeline | 31 May 2023 |
| Audited by | Ilya Vaganov |
| Approved by | Artur Makhnach, Kirill Minyaev |
| Languages | Solidity |
| Methods | Architecture Review, Unit Testing, Functional Testing, Manual Review |
| Source code | https://etherscan.io/address/0x6690e2a46d00e72d87cbadf80627cd3d7565d840#code |
| Chain | Ethereum |
| Status | Passed |

## Disclaimer




| | | |
|---|---|---|
| | 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. |

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

**Found vulnerabilities and contradictions in the logic of the contract.**

There may be a loss of access rights for the owner of the contract, which may make it impossible to use the main administrative functionality.

The token transfer functionality logic cannot guarantee that users will receive locked tokens at a fixed time stamp or be able to dispose of their tokens. This implementation calls into question the reliability of the user experience and cannot guarantee users timely access to tokens.

## Summary of findings

| ID | Description | Severity |
|---|---|---|
| DFM-1 | Possible loss of rights to the contract | Medium Risk |
| DFM-2 | Ability to change LockLiquidity | Low Risk |
| DFM-3 | The vesting address has not been explicitly set | Informational |
| DFM-4 | Incorrect messages in require | Informational |
| DFM-5 | LockLiquidity timestamp | Informational |
| DFM-6 | Anti Bot Mechanism is missing | 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 |

## *Findings*

### DFM-1 «Possible loss of rights to the contract»

**Severity:** Medium Risk

**Description:** The inherited Ownable contract from OpenZeppelin has a renounceOwnership() function that removes the owner of the contract. This function can be accidentally called, leaving the contract without an owner. Considering that many contract functions are available only to the owner, the loss of access can become a critical issue.

**Recommendation:** In your case, you can stop using the renounceOwnership() function or change it like this:

```
function renounceOwnership() public override onlyOwner {
    revert("Safety: Is not allowed");
}
```

## DFM-2 «Ability to change LockLiquidity»

**Severity:** Low Risk

**Description:** The setLockLiquidity() function allows the contract owner to change the LockLiquidity value. So, in theory, it is possible to change LockLiquidity so that it is always greater than block.timestamp, as a result of which transfers for most users and vesting will always be unavailable and users will not be able to dispose of their tokens or will not be able to receive tokens from vesting.

**Recommendation:** This approach cannot guarantee users fixed time limits for unlocking tokens and the possibility of using them. The best practice would be to use a fixed value of LockLiquidity and not be able to change it.

## DFM-3 «The vesting address has not been explicitly set»

**Severity:** Informational

**Description:** The SetVesting() function only sets whitelistedAddresses to false, but does not explicitly store the address of the vesting. As a result, when using the SetMultiSigAuthority() function for a vesting address, whitelistedAddresses can be set to true by accident or on purpose, allowing users to get locked tokens ahead of time.

**Recommendation:** The best practice is to implement a different storage mechanism for the address or addresses of the vestings to avoid undesirable situations.

## DFM-4 «Incorrect messages in require»

**Severity:** Informational

**Description:** In the transfer() and transferFrom() functions, the require check is performed with the error message "Token transfer successful". The second argument to require is the message that will be returned if the condition is false, which is an error message. In your case, there is a message that says the transfer was successful, although it will only be returned if the transfer failed.

**Recommendation:** Modify the message so that it notifies users of a transfer error.

## DFM-5 «LockLiquidity timestamp»

**Severity:** Informational

**Description:** The LockLiquidity parameter is set to 1716397200 (22 May 2024) by default, even though the comment in the code says "Unix timestamp 01 December 2023".

**Recommendation:** Make sure the LockLiquidity value is set correctly and correct or remove the comment.

## DFM-6 «Anti Bot Mechanism is missing»

**Severity:** Informational

**Description:** The antiBotMechanism() function returns the string "Sniper and Front-running bots are restricted", although neither this function nor other functionality provides for restrictions for sniper and front-running bots.

**Recommendation:** The antiBotMechanism() function can be confusing to users, so the best practice is to remove it or extend the functionality of the contract by adding front-running resistance.

## Automated Analyses

**Slither**

Slither's automatic analysis not found vulnerabilities, or these false positives results .

# 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

| | |
|---|---|
| Resolved | 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 |