



Smart Contract Audit Report

November, 2022




DEFIMOON PROJECT

Audit and
Development


CONTACTS

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

 [defimoon_org](#)

 [defimoonorg](#)

 [defimoon](#)

 [defimoonorg](#)

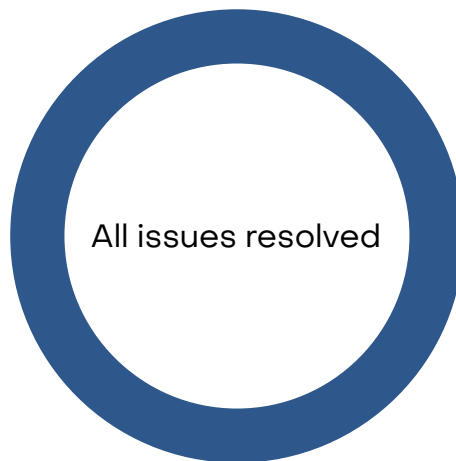


21 November 2022

This audit report was prepared by DefiMoon for Metavault.trade

Audit information

Description	Timelock smart contract for deferred withdrawal of tokens
Audited files	MVXReserveTimelock.sol
Timeline	16 November 2022 – 21 November 2022
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	Initial code Reaudit code
Chain	Polygon
Status	Passed



0

	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.

Audit overview

No major issues were found

The contracts use a very outdated version of the Solidity compiler, namely 0.6.12, which is not relevant at the moment and which forces the use of unnecessary functionality for new versions (safeMath library).

You may notice that the imported contracts, libraries and interfaces are the solution from OpenZeppelin, but they use outdated versions of them. We strongly recommend moving to modern solutions.

There are some risks associated with using the Ownable contract.

There are minimal opportunities to optimize the code to save gas.

Some best practices could also be implemented:

1. Use up-to-date versions of the compiler, but not experimental or too new, as they cannot always be trusted.
2. Use modern solutions from OpenZeppelin.

Summary of findings

According to the standard audit evaluation, Solidity's validated smart contracts are secure and ready for production, but there are not very serious vulnerabilities, and gas consumption can be slightly optimized.

ID	Description	Severity	Status
<u>DFM-1</u>	Possible to lose ownership	Medium Risk	Resolved
<u>DFM-2</u>	Action may not be cleared if function revert	Low Risk	Resolved
<u>DFM-3</u>	Can't create two identical actions	Low Risk	Resolved
<u>DFM-4</u>	Using newer compiler versions	Informational	Resolved
<u>DFM-5</u>	Minor gas consumption optimizations	Informational	Resolved
<u>DFM-6</u>	Using a deadline for actions	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 «Possible to lose ownership»

Severity: Medium Risk

Status: Resolved

Description: The implementation of the Ownable contract includes the `renounceOwnership()` function, with which you can lose the owner of a smart contract if you are not careful. Your smart contract does not imply the refusal of the owner, because all functions are called on behalf of the owner.

You can also lose ownership by calling the `transferOwnership()` function by accidentally transferring ownership to the wrong address.

Recommendation:

The simplest solution would be to override the `renounceOwnership()` function like this:

```
function renounceOwnership() public override onlyOwner {  
    revert("Cant renounce ownership");  
}
```

If necessary, `transferOwnership()` can do the same, or you can use the [pending owner](#) concept to avoid transferring ownership to the wrong address.

DFM-2 «Action may not be cleared if function revert»

Severity: Low Risk

Status: Resolved

Description: Action may not be cleared in case of revert in `MVXReserveTimelock::processWithdraw()`. In `MVXReserveTimelock::processWithdraw()` calls the `_clearAction(action)` method, but the changes may not be applied if the function call fails as a result of `require(withdrawAmount > 0, "...")`. As a result, the action will be stored in the mapping until tokens appear on the contract or the action is manually deleted by the owner

Recommendation: It might make more sense to use `return` instead of `require` to just terminate the function.

DFM-3 «Can't create two identical actions»

Severity: Low Risk

Status: Resolved

Description: There can not be two actions for the withdrawal of the same amounts at the same time. Since only one unique `amount` value is used to form the `bytes32 action` hash, two actions with the same amount cannot exist at the same time.

Recommendation: If there is at least a small chance that you will need to create two identical actions, then it makes sense to add one more variable to form a hash, for example nonce.

DFM-4 «Using newer compiler versions»

Severity: Informational

Status: Resolved

Description: Smart contracts use a very outdated version of the compiler (0.6.12), which is not up-to-date at the moment. Using newer compiler versions is best practice. Due to the use of obsolete versions, you have to use the SafeMath library, although after version 0.8.0 its use is [no longer necessary](#). Also, due to the outdated version, you are running outdated OpenZeppelin solutions that have already been updated to version 0.8.0.

Recommendation: We recommend using current compiler versions starting from 0.8.0. But at the same time, it's better not to use the newest versions of the compiler until they are well tested. It should be limited to version 0.8.13.

DFM-5 «Minor gas consumption optimizations»

Severity: Informational

Status: Resolved

Description: The MVXReserveTimelock::processWithdraw() function and the MVXReserveTimelock::_mvx variable can be changed to save gas.

Recommendation:

1) Operation != costs less than operation >. So it makes sense to change

```
require(withdrawAmount > 0, "MVX-Reserve Timelock: No withdrawable amount!");
```

on the

```
require(withdrawAmount != 0, "MVX-Reserve Timelock: No withdrawable amount!");
```

2) Instead of converting _mvx to the IERC20 type each time, you can immediately declare a variable with this type. Like this:

```
...
IERC20 public _mvx;
constructor(address beneficiary, IERC20 mvx) {
    _beneficiary = beneficiary;
    _mvx = mvx;
}
...
```

DFM-6 «Using a deadline for actions»

Severity: Informational

Status: Resolved

Description: Created actions have a buffer time but no deadline. For the most part, we paid attention to this because the withdraw amount is not strictly fixed, because if the contract balance is less than the requested amount, then the entire available amount is withdrawn. Thus, sometimes it may not be beneficial to call MVXReserveTimelock::processWithdraw() right away, but to wait until the amount reaches the specified one.

Recommendation: Add a deadline if you need it.

Automated Analyses

Slither

Slither's automatic analysis found no additional vulnerabilities, or these results were either related to code from dependencies or false positives.

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