Smart Contract Audit Report

September, 2022

DEFIMOON PROJECT

Audit and
Development

CONTACTS

https://defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

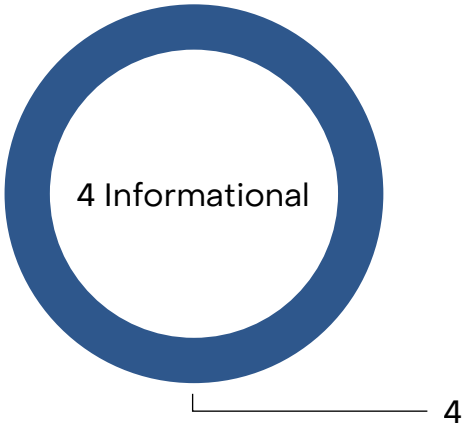
be secure

August 29th 2022

This audit report was prepared by Defimoon for DEXfinance

Audit information

Description	The contract implements the ERC-20 type token and DEX interaction mechanics
Audited files	DexLPZapper.sol, DexStrategy.sol, DexStrategyFactory.sol, DexStrategyHarvester.sol, DexStrategyProfitStorage.sol, DexStrategyZapper.sol, DexTreasury.sol
Timeline	29 August — 2 September
Audited by	Cyrill Novoseletskyi, Ilya Vaganov
Approved by	Artur Makhnach, Cyrill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/dexIRA/strategy-contracts/commit/0f963b63eb82e28e5374f49eb009d3aa3f4c455b
Commit hash	0f963b
Network	BSC
Project Website	https://www.dexfinance.com
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.

Summary of findings

According to the standard audit assessment, the audited solidity smart contracts are secure and can be considered ready for production, however a number of small issues were discovered and it is recommended to address them prior to future development. These issues represent minor points, mostly related with code optimization and do not damage the project or project's reputation directly.

ID	Description	Severity
DFM-1	No "NatSpec" comment format	Informational
DFM-2	Use _msgSender function	Informational
DFM-3	Use a specific value	Informational
DFM-4	Excessive congestion	Informational

Deployed Contracts

General Contracts:

DexLPZapper	0x72de5623C783bc8856B0aDb2A52330153930b02C
DexTreasury	0x395E851524562654c8d3EB4af021Dcd5322810CC

Strategy Contracts:

DexStrategyProfitStorage	0x2C9824D207cEF8c9781e9824F724834EDa7c0F9e
DexStrategyZapper	0x18f0cD0b63CbBcF6C6A21403110379084cb59c9e
DexStrategyHarvester	0xD21C7C0CA1560b0959e4789e08a3Db2Aa3998674
DexStrategy	0x31E77abBcDf985711926786214971CeC3879E3Ea
DexStrategyFactory	0xc73F10364a5423291bB98b72E28F4A27b0F8EF49

Audit overview

No major security issues were found.

Descriptions of functions do not comply with generally accepted standard called NatSpec, which impairs readability and understandability of the code (DFM-1).

The contracts that inherit Ownable.sol access, the `_msgSender` function is a safer replacement for the classic `msg.sender` and also in some cases proves to be more useful (DFM-2).

In the `onlyValidPaths` modifier, it is better to pass one address from the path since the second one is never used (DFM-3).

The `_updateKeeper` function is superfluous since it is not used anywhere else except in a function with an almost similar name `updateKeeper` (DFM-4).

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 «No "NatSpec" comment format»

Severity: Informational

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-2 «Use msgSender function»

Severity: Informational

Description:

In contracts that inherit Ownable.sol access, the `_msgSender` function is a safer replacement for the classic `msg.sender`

Recommendation:

In contracts that inherit Ownable.sol, replace `msg.sender` with `_msgSender.`

DFM-3 «Use a specific value»

Severity: Informational

Description:

In the `onlyValidPaths` modifier, it is better to pass one address from the path since the second one is never used, which makes the execution of functions containing this modifier more expensive

Recommendation:

Replace the data type of an address array to simply an address.

```
modifier onlyValidPaths(address path0, address path1, address first) {
     require(path0 == first && path1 == first, "DexLPZapper: First paths element neq tokenIn");
     _;
}
```

DFM-4 « Excessive congestion»

Severity: Informational

Description:

The function `_updateKeeper` is redundant because it is not used anywhere else except for one occurrence (`updateKeeper`) and based on the name, it is only needed there. Such functions congest the contract and make the code less readable as well as increase the cost of the contract deployment and execution.

Recommendation:

Move the body from the internal function to the public one and delete this internal function.

Automated Analyses

Slither

Slither has reported 181 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.

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