DEFIMOON

be secure

Smart Contract Audit Report

September, 2023

Rivera

DEFIMOON PROJECT

Audit and
Development

CONTACTS

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

DEFIMOON

be secure

28 September 2023

This audit report was prepared by DefiMoon for RiveraMoney.

Audit information

Description	Auto-compounding Vault for Liquidity V3 management
Timeline	12 September 2023 - 28 September 2023
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Project Site	https://testnet.rivera.money/
Source code	https://github.com/RiveraMoney/farming-vault-factory/tree/c4771b07201f67b02e03936c419f6aaf0f905884
Reaudit Source code	https://github.com/RiveraMoney/farming-vault-factory/tree/ea349aa08b42f93ce5e91cf424cc5b19ca008380
Network	EVM-like
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.

Audit overview

Major vulnerabilities have been found.

The vault contract has various implementations that control access to deposit and withdraw functions. However, in some cases, access can be restricted for users who have already made a deposit.

When liquidity is burned, reward tokens are not converted into deposit tokens.

There is an invalid panic function that cannot be called and an incorrect path setting where existing arrays are extended instead of modified.

The commission taken from the withdraw amount is not sent anywhere and remains on the contract balance. The owner of the vault can bypass the payment of commissions.

There is an inflation vulnerability in the contract.

The contract could potentially remain without an owner.

Lack of checks when installing a manager can lead to paralysis of some contract functions.

Summary of findings

ID	**Description**	**Severity**	**Status**
DFM-1	Restricted access to the vault	High Risk	Resolved
DFM-2	When liquidity is burned, reward tokens are not withdrawn	Medium Risk	Resolved
DFM-3	Invalid panic function	Medium Risk	Resolved
DFM-4	Incorrect path setting	Medium Risk	Resolved
DFM-5	The commission is not sent anywhere	Low Risk	Acknowledged
DFM-6	Bypassing the payment of commissions	Low Risk	Resolved
DFM-7	Inflation vulnerability	Low Risk	Acknowledged
DFM-8	Potential loss of owner	Low Risk	Partially Resolved
DFM-9	Lack of address check	Low Risk	Resolved
DFM-10	Other/ Optimisations	Information	Partially 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 «Restricted access to the vault» | RiveraAutoCompoundingVaultV2

Severity: High Risk

Status: Resolved

Description: There are various implementations of the Vault contract (private, public, whitelisted) that override the _restrictAccess function, controlling access to the _deposit and _withdraw functions, however shares tokens (ERC20 tokens of the RiveraAutoCompoundingVaultV2 contract) can be transferred or in the case of public and whitelisted implementations access can be restricted for a user who has already made a deposit, but now cannot use the withdraw function.

Recommendation: We recommend removing the restriction from the _withdraw function.

DFM-2 «When liquidity is burned, reward tokens are not withdrawn» | RiveraConcLpStaking

Severity: Medium Risk

Status: Resolved

Description: The _burnAndCollectV3 function calls chef.withdraw, which also collects reward tokens, but the reward tokens are not converted into deposit tokens, as in the harvest function.

Recommendation: We recommend adding the conversion of reward tokens to deposit tokens in the _burnAndCollectV3 function.

DFM-3 «Invalid panic function» | RiveraConcLpStaking

Severity: Medium Risk

Status: Resolved

Description: The panic function could not be called.

The first step is to call the pause function, which removes the allowances, causing the ERC20InsufficientAllowance error to be returned when _lptoDepositTokenSwap is called.

Recommendation: We recommend calling pause() after IERC20(depositToken).safeTransfer(vault, _lptoDepositTokenSwap(amount0, amount1))

DFM-4 «Incorrect path setting» | RiveraConcLpStaking

Severity: Medium Risk

Status: Resolved

Description: The _setAddressArray and _setUint24Array functions extend existing arrays rather than modifying them.

Recommendation: We recommend calling pause() after IERC20(depositToken).safeTransfer(vault, _lptoDepositTokenSwap(amount0, amount1))

DFM-5 «The commission is not sent anywhere» | RiveraConcLpStaking

Severity: Low Risk

Status: Acknowledged (protocol logic)

Description: The withdraw function takes a commission from withdrawAmount, but the commission is not sent anywhere and remains on the contract balance. As a result, this commission will be used on your next deposit.

Recommendation: If this is not part of the protocol logic, add sending a fee.

DFM-6 «Bypassing the payment of commissions» | RiveraConcLpStaking

Severity: Low Risk

Status: Resolved

Description: The harvest function is used to reinvest rewards and charges a fee, however, the owner of the vault can use the changeRange function, which will recreate the position taking into account the collected pool fees (but without using reward tokens - more details in DFM-2).

Recommendation: We recommend changing the _burnAndCollectV3 function so that you first call chef.withdraw and NonfungiblePositionManager.collect and charge a fee, and only then call NonfungiblePositionManager.decreaseLiquidity and NonfungiblePositionManager.burn.

DFM-7 «Inflation vulnerability» | RiveraAutoCompoundingVaultV2 | ERC4626

Severity: Low Risk

Status: Acknowledged

Description: There is a vulnerability in the inherited ERC4626 contract (described here). Despite the fact that the RiveraAutoCompoundingVaultV2 contract is positioned as a personal Vault, this vulnerability is still relevant and can be reproduced by the owner accidentally or by another user in a Public or Whitelisted Vault implementation.

Recommendation: We recommend that you become familiar with this vulnerability and use _decimalsOffset at least equal to 8 (like this).

DFM-8 «Potential loss of owner» | RiveraAutoCompoundingVaultV2 | AbstractStrategyV2

Severity: Low Risk

Status: Partially Resolved

Description: The AbstractStrategyV2 and RiveraAutoCompoundingVaultV2 contracts inherit the Ownable contract from OpenZeppelin which includes the renounceOwnership function. This function resets the owner of the contract without the possibility of restoring it, which can lead to irreparable consequences if this function is called, since most of the functionality of contracts is available only to the owner.

Also, the Ownable::transferOwnership function is not safe either, because it does not check the address of the new owner.

Recommendation: Most of the functions in your AbstractStrategyV2 and RiveraAutoCompoundingVaultV2 contracts require owner permissions, and as a result, loss of permissions can become critical. The best solution would be to stop using OpenZeppelin's renounceOwnership function. For example, like this:

```
function renounceOwnership() public override onlyOwner {
    revert("Renounce ownership disabled");
}
```

It's also best practice to use transfer the owner in two steps, like this.

DFM-9 «Lack of address check» | AbstractStrategyV2

Severity: Low Risk

Status: Resolved

Description: The setManager function does not check the address when the manager changes. If address(0) is sent, then Vault will be partially paralyzed, since when trying to send a commission to address(0), an ERC20InvalidReceiver error will be returned.

Recommendation: We recommend that you disable setting address(0) as a manager. In addition, in order not to lose access to some functionality and receive fees if the address is incorrectly specified, we recommend implementing two-step transfer of rights (as stated in DFM-8).

DFM-10 «Other/ Optimisations»

Severity: Information

Status: Partially Resolved

Description:

Unused imports:

RiveraAutoCompoundingVaultV2.sol:

- import "@openzeppelin/security/ReentrancyGuard.sol";

DexV3Calculations.sol:

- import "@rivera/strategies/staking/interfaces/libraries/ILiquidityMathLib.sol";

We recommend using structures explicitly when a function returns a structure:

For example (DexV3Calculations.sol):

```
pool.Slot0 memory slot0 = pool.slot0();
```

instead of

```
(uint160 sqrtPriceX96, int24 tick, , , , , ) = pool.slot0();
```

In owner-only functions, you can use msg.sender instead of owner():

For example (RiveraAutoCompoundingVaultV2.sol):

```
function setTotalTvlCap(uint256 totalTvlCap_) external {
    _checkOwner();
    require(totalTvlCap != totalTvlCap_, "Same TVL cap");
    emit TvlCapChange(msg.sender, totalTvlCap, totalTvlCap_);
    totalTvlCap = totalTvlCap_;
}
```

instead of

```
function setTotalTvlCap(uint256 totalTvlCap_) external {
    _checkOwner();
    require(totalTvlCap != totalTvlCap_, "Same TVL cap");
    emit TvlCapChange(owner(), totalTvlCap, totalTvlCap_);
    totalTvlCap = totalTvlCap_;
}
```

Try to reuse existing memory variables:

For example (RiveraAutoCompoundingVaultV2.sol):

```
      function maxMint(address receiver) public view virtual override returns
(uint256) {
          // ...
          uint256 userCap = userTvlCap[receiver];
          uint256 maxFromUserTvlCap = userCap > 0 ? convertToShares(userCap) -
balanceOf(receiver): type(uint256).max;
          // ...
```

```
}
```

instead of

```
function maxMint(address receiver) public view virtual override returns
(uint256) {
    // ...
    uint256 userCap = userTvlCap[receiver];
    uint256 maxFromUserTvlCap = userCap > 0 ?
convertToShares(userTvlCap[receiver]) - balanceOf(receiver): type(uint256).max;
    // ...
}
```

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