Smart Contract Audit Report

October, 2023

Skydrome

DEFIMOON PROJECT

Audit and
Development

CONTACTS

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

DEFIMOON

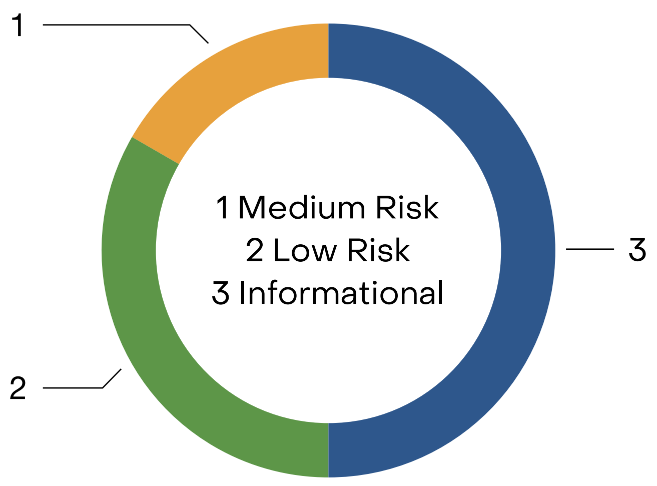
be secure

17 October 2023

This audit report was prepared by DefiMoon for Skydrome Finance.

Audit information

Description	Liquidity AMM
Timeline	16 October 2023 - 17 October 2023
Approved by	Artur Makhnach, Kirill Minyaev
Audit Scope	Pair.sol, PairFactory.sol, Router.sol
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Project Site	https://skydrome.finance/
Source code	https://blockscout.scroll.io/address/0x2516212168034b18a0155FfbE59f2f0063fFfBD9/contracts#address-tabs https://blockscout.scroll.io/address/0xAA111C62cDEEf205f70E6722D1E22274274ec12F/contracts#address-
Network	EVM-like (Scroll)
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

No major vulnerabilities were found.

Non-critical vulnerabilities and comments were found that we recommend paying attention to. If contracts are initialized correctly and administrative functions are called correctly, there should be no issues.

Summary of findings

ID	**Description**	**Severity**
DFM-1	Bribe address check	Medium Risk
DFM-2	Potential loss of owner	Low Risk
DFM-3	Old allowance is not cleared	Low Risk
DFM-4	Redundant variable	Information
DFM-5	Unused variable	Information
DFM-6	Errors description	Information

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	Not 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 «Bribe address check» | Pair

Severity: Medium Risk

Description: The _sendTokenFees function calls the notifyRewardAmount function on the externalBribe contract, however, if the externalBribe address is not set or does not contain the notifyRewardAmount function, then when the swap function is called, it will be reverted with an error. In addition, the externalBribe address can only be set by the voter that is specified when initializing the Pair contract, however there is no guarantee that the voter will be different from address(0).

Recommendation: We recommend adding a check to the _sendTokenFees function that the externalBribe address is set like this:

```
function _sendTokenFees(address token, uint amount) internal {
    if (amount != 0) {
        if (hasGauge && externalBribe != address(0)) {
            IBribe(externalBribe).notifyRewardAmount(token, amount); // transfer
fees to exBribes
            emit GaugeFees(token, amount, externalBribe);
        }
    }
}
```

Additionally, we recommend adding a check that the externalBribe contract matches the desired interface in the setExternalBribe function.

We also recommend prohibiting the creation of a new pair in the PairFactory contract if voter == address(0), so that the pair is not left without a voter.

DFM-2 «Potential loss of owner» | PairFactory

Severity: Low Risk

Description: The PairFactory contract 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: Main functions in your contract 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-3 «Old allowance is not cleared» | Pair

Severity: Low Risk

Description: When changing the externalBribe address in the setExternalBribe function, the allowance for the old externalBribe address is not cleared.

Recommendation: We recommend clearing allowance for the old externalBribe address like this:

```
function setExternalBribe(address _externalBribe) external {
    require(msg.sender == voter, "Only voter can set external bribe");
    address oldBribe = externalBribe;
    _safeApprove(token0, oldBribe, 0);
    _safeApprove(token1, oldBribe, 0);
    externalBribe = _externalBribe;
    _safeApprove(token0, externalBribe, type(uint).max);
    _safeApprove(token1, externalBribe, type(uint).max);
    emit ExternalBribeSet(_externalBribe);
}
```

DFM-4 «Redundant variable» | Pair

Severity: Information

Description: DOMAIN_SEPARATOR is declared as an internal variable, but it is dynamically updated each time and is used only in the permit function.

Recommendation: We recommend using a memory variable to save gas.

DFM-5 «Unused variable» | Pair | PairFactory

Severity: Information

Description: The tank variable is declared and can be set, but is not used anywhere.

Recommendation: We recommend checking the logic associated with tank variable and removing it if it is not used.

DFM-6 «Errors description» | *

Severity: Information

Recommendation: We recommend that you always include a description of the errors in revert and require, or at least abbreviated error codes.

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