DEFIMOON

be secure

Smart Contract Audit Report

May, 2024

VeryLongSwap Single Staking

CONTACTS

DEFIMOON PROJECT

Audit and Development

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

DEFIMOON

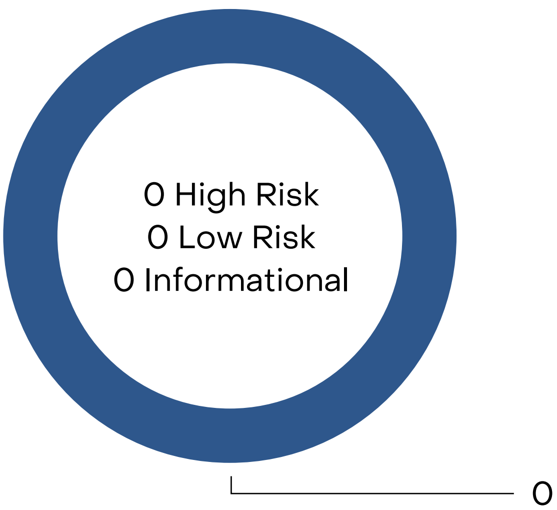
be secure

9 May 2023

This audit report was prepared by DefiMoon for VeryLongSwap.

Audit information

Description	Fork of PancakeSwap PancakeFixedStaking contract
Timeline	2 - 9 May 2024
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/VeryLongSwap/singlestaking-contract
Network	EVM-Compatible
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

The codebase difference assessment showed minor deviations from PancakeSwap original contracts that reduce the original functionality, without introducing new mechanics and **can be considered safe to use**.

This report is based deviation assessment of the client contract and original PancakeSwap contract deployed on BinanceSmartChain at address *0x25217F4eAEBAD142615E9Da5ebbf8751659af738*. While there are a number of key differences of the forked contract to the original, those deviations simplify the original contract and reduce it's functionality. During the security assessment it was concluded that implemented differences do not pose a threat to the contract's operation — key differences are highlighted below.

Library and Interface Usage:

Original Contract: Utilizes upgradeable versions of OpenZeppelin contracts suited for upgradeable smart contracts.
Forked Contract: Employs non-upgradeable standard OpenZeppelin contracts and fewer interfaces, indicating a non-upgradeable design.
Impact: The forked contract may lack the flexibility provided by upgradeable patterns, potentially affecting long-term scalability.

Structs and Data Elements:

Original Contract: Includes additional parameters within the *Pool* struct, supporting complex features like reward boosting.
Forked Contract: Simplifies the *Pool* struct by removing parameters related to boosting features.
Impact: Simplification reduces the feature set of the user reward mechanisms.

Functionality and Feature Set:

Original Contract: Contains complex functionalities for boosting rewards and detailed management of user states and transactions.
Forked Contract: Removes those functionalities, specifically the boosting mechanics and some detailed transaction conditions.
Impact: The reduced complexity may enhance contract performance and ease of understanding but at the cost of losing more sophisticated user reward and management features.

Constant and Variable Definitions:

Original Contract: Defines critical constants and uses private variables appropriately to manage sensitive data.

Forked Contract: Changes the scope of some variables and modifies constant values such as the WBNB address, which might reflect deployment on a different network.

Impact: Altering constants and variable scopes can affect the contract's interaction with external tokens and overall posture.

Security Measures and Modifiers:

Original Contract: Implements security checks using modifiers to restrict access and interactions by smart contracts.

Forked Contract: Continues to use similar security measures but with slight differences in implementation, which do not pose a direct security risk.

Impact: Minor changes in security implementations can affect how secure the contract is against automated attacks and malicious contracts, however implemented changes do not pose a direct risk.

Error Handling and User Feedback:

Original Contract: Provides clear and informative error messages, enhancing debugging and user interaction.

Forked Contract: Contains less descriptive error messages and fewer feedback mechanisms.

Impact: The reduction in clarity of error messages could lower the ease of debugging and user understanding, potentially leading to a poorer user experience.

Overall Complexity and Robustness:

Original Contract: Exhibits a high degree of complexity with robust features designed to handle a variety of scenarios and interactions.

Forked Contract: Reduces overall complexity, possibly to decrease potential security risks or to streamline operations.

Impact: While simplification can reduce potential attack vectors and improve performance, it may also strip the contract of valuable features and flexibility needed for certain use cases.

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

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