defimoon.org
twitter.com/Defimoon_org
linkedin.com/company/defimoon

Rome, Italy, 00165

Smart contract’s
Audit report

SafeYields

January, 2025

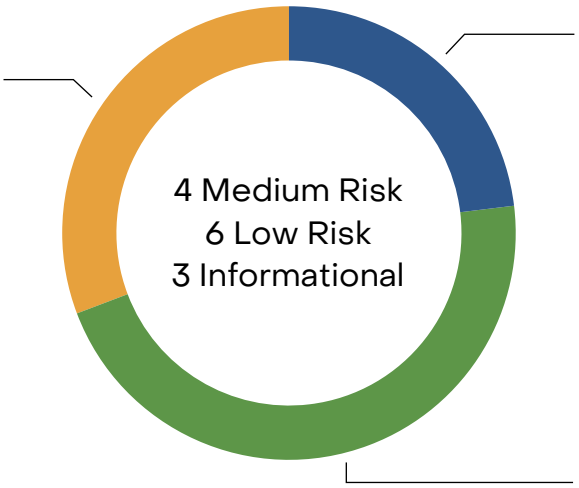
DEFIMOON
be secure

15 January 2025

This re-audit report was prepared by DefiMoon for SafeYields.

Audit information

Description	Token vesting smart contract
Audited files	VestingPreExchange.sol
Timeline	18 Dec 2024 - 15 Jan 2025
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/SafeYields/safe-yields-presale/compare/defimoon-mitigations
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

This audit was conducted to assess the security, functionality, and overall quality of four smart contracts within the SafeYield ecosystem: **SafeYieldVesting**, **SafeYieldStaking**, **SafeYieldPresale**, and **SafeYieldAirdrop**. The contracts aim to facilitate staking, vesting, token presales, and airdrop distribution, leveraging robust mechanisms such as Merkle proofs and modular architecture. The audit focused on identifying vulnerabilities, verifying compliance with best practices, and optimizing the contracts for efficiency.

The codebase exhibits strong modularity and utilizes OpenZeppelin libraries, ensuring compliance with Solidity standards. Error handling is comprehensive, leveraging custom error messages for clear issue tracking. The use of access controls (`Ownable`, `AccessControl`) and modifiers provides essential safeguards for critical operations. However, some functions lack protections against reentrancy and inefficient gas usage, which can affect security and usability at scale.

The **SafeYieldVesting** contract manages token vesting schedules, ensuring users can claim tokens over time. It employs clear logic for vesting calculations and integrates seamlessly with other ecosystem contracts. However, it exhibits vulnerabilities such as missing reentrancy protections and potential inefficiencies in gas usage for complex calculations.

The **SafeYieldStaking** contract handles staking of SAY tokens and distribution of rewards in SafeToken and USDC. Its design includes callback mechanisms for extensibility and precision arithmetic for rewards calculation. While robust, the contract is prone to reentrancy vulnerabilities in critical functions and exhibits inefficiencies in callback processing and reward management.

The **SafeYieldPresale** contract facilitates token purchases and referral-based commissions during the presale phase. It features dynamic token pricing, allocation limits, and integrated referral mechanisms. However, it requires additional safeguards to prevent reentrancy, unsafe price adjustments, and inefficiencies in referral handling.

The **SafeYieldAirdrop** contract distributes tokens to eligible users via Merkle-based proofs. It ensures token vesting and staking integration, offering a seamless user experience. However, it lacks reentrancy protections, token cap enforcement during callbacks, and sufficient validation for Merkle proof parameters.

Summary of findings

ID	Description	Severity	Status
DFM-1	Missing Reentrancy Protection in Vesting Functions	Medium Risk	Resolved
DFM-2	Gas Inefficiency in Vesting Calculations	Low Risk	Acknowledged
DFM-3	Dynamic Max Supply Modification	Informational	Closed
DFM-4	Missing Reentrancy Protection in Staking Functions	Medium Risk	Resolved
DFM-5	Inefficient Callback Handling in Staking	Low Risk	Acknowledged
DFM-6	Potential Mismanagement of Rewards Debt	Low Risk	Acknowledged
DFM-7	Unsafe Token Price Adjustments in Presale	Low Risk	Resolved
DFM-8	Referrer Self-Referral Vulnerability	Low Risk	Closed
DFM-9	Gas Inefficiency in Referral Commission Handling	Informational	Acknowledged
DFM-10	Lack of Reentrancy Protection in Airdrop Functions	Medium Risk	Closed
DFM-11	Insufficient Validation of Merkle Proofs in Airdrop	Medium Risk	Closed
DFM-12	Lack of Token Cap Enforcement in Clawbacks	Low Risk	Closed
DFM-13	Gas Inefficiency in Airdrop Processing	Informational	Acknowledged

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

SafeYieldVesting

DFM-1 «Missing Reentrancy Protection in Vesting Functions»

Severity: Medium Risk

Status: Resolved

Description: The `vestFor` and `unlock_sSayTokens` functions involve external calls (e.g., `safeTransfer`) before updating critical internal state variables. This sequence can allow reentrancy attacks if the external calls invoke malicious code.

Recommendation: Add the `nonReentrant` modifier to the `vestFor` and `unlock_sSayTokens` functions to prevent reentrancy.

DFM-2 «Gas Inefficiency in Vesting Calculations»

Severity: Low Risk

Status: Acknowledged

Description: The `vestedAmount` function involves complex calculations and multiple arithmetic operations. This may result in high gas costs, particularly for users interacting with the contract frequently or under stress conditions.

Recommendation: Optimize the vesting calculation logic by caching results or simplifying the formula to reduce computational overhead.

DFM-3 «Dynamic Max Supply Modification»

Severity: Informational

Status: Closed

Description: The contract allows the owner to modify `maxSupply` after deployment. While this feature adds flexibility, it could lead to trust issues among users if the supply cap is altered unexpectedly.

Recommendation: Make `maxSupply` immutable after deployment or implement a mechanism that enforces transparency, such as requiring user consensus or a delay period before changes take effect.

SafeYieldStaking

DFM-4 «Missing Reentrancy Protection in Staking Functions»

Severity: Medium Risk

Status: Resolved

Description: Functions like `stakeFor`, `unstakeVestedTokens`, and `unStake` involve external calls (`safeTransfer` and `safeTransferFrom`) before updating critical state variables. This sequence exposes these functions to reentrancy attacks if a malicious contract is called during external transfers.

Recommendation: Add the `nonReentrant` modifier to these functions and ensure state variables are updated before external calls where possible.

DFM-5 «Inefficient Callback Handling in Staking»

Severity: Low Risk

Status: Acknowledged

Description: The `stakeFor` and `unstakeVestedTokens` functions iterate through the `callbacks` list twice, once before and once after execution. This double iteration results in gas inefficiency, especially as the list size grows.

Recommendation: Optimize callback handling by batching or reducing redundant iterations to improve gas efficiency.

DFM-6 «Potential Mismanagement of Rewards Debt»

Severity: Low Risk

Status: Acknowledged

Description: Rewards debt calculations in `_stake` and `_unStake` rely on precise arithmetic operations, which could result in rounding issues or misalignments under certain conditions. Mismanagement of rewards debt may lead to incorrect reward allocations.

Recommendation: Include tests for edge cases and add validation checks to ensure consistency in rewards debt calculations.

SafeYieldPresale

DFM-7 «Unsafe Token Price Adjustments»

Severity: Low Risk

Status: Resolved

Description: The `setTokenPrice` function allows the owner to change the token price at any time, including during an active presale. This could lead to exploitation or inconsistencies for investors who depend on stable pricing.

Recommendation: Prevent price adjustments during an active presale or introduce a delay mechanism with notification to ensure transparency.

DFM-8 «Referrer Self-Referral Vulnerability»

Severity: Low Risk

Status: Closed

Description: While the contract prevents self-referrals using the `SYPS__REFERRAL_TO_SELF` error, there is no mechanism to ensure that referrer IDs are unique. This could allow malicious users to create multiple referrer IDs for self-gain.

Recommendation: Enforce uniqueness for referrer IDs and validate them against existing IDs during registration.

DFM-9 «Gas Inefficiency in Referral Commission Handling»

Severity: Informational

Status: Acknowledged

Description: The referral commission mechanism involves nested mappings and array iterations (`referrerRecipients`), which could result in high gas consumption as the number of referrals grows.

Recommendation: Optimize referral data structures and impose size limits or constraints to minimize gas costs.

SafeYieldAirdrop

DFM-10 «Lack of Reentrancy Protection in Airdrop Functions»

Severity: Medium Risk

Status: Closed

Description: The `stakeAndVestSayTokens` function interacts with the staking contract and performs external calls (`approve` and `stakeFor`) before updating the `hasClaimed` mapping. This sequence exposes the function to potential reentrancy attacks.

Recommendation: Add the `nonReentrant` modifier to `stakeAndVestSayTokens` to prevent reentrant calls and ensure state updates occur before external interactions.

Client reaudit comments: hasClaimed is updated before the external calls

DFM-11 «Insufficient Validation of Merkle Proofs»

Severity: Medium Risk

Status: Closed

Description: The `stakeAndVestSayTokens` function does not verify that the `amount` parameter aligns with the allocation defined in the Merkle tree. This opens the door for users to claim incorrect amounts using valid proofs.

Recommendation: Add logic to cross-check the `amount` parameter against the expected allocation encoded in the Merkle tree.

Client reaudit comments: as If an invalid amount is used , the Proof verification will Fail with *`SYA__INVALID_PROOF`*

DFM-12 «Lack of Token Cap Enforcement in Clawbacks»

Severity: Low Risk

Status: Closed

Description: The `clawBackSayTokens` function does not ensure that sufficient tokens remain in the contract for unclaimed allocations before clawing back tokens to the owner.

Recommendation: Add a check to ensure the contract retains enough tokens to cover all unclaimed airdrops.

DFM-13 «Gas Inefficiency in Airdrop Processing»

Severity: Informational

Status: Acknowledged

Description: The `stakeAndVestSayTokens` function performs multiple sequential actions (e.g., `approve`, `stakeFor`), potentially leading to high gas costs for users.

Recommendation: Optimize the function by batching or streamlining operations to reduce gas usage.

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