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

Rome, Italy, 00165

Smart contract's
Audit report

Hype

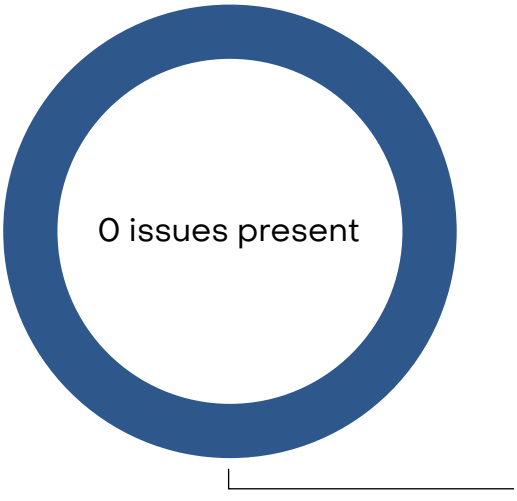
February, 2025

DEFIMOON
be secure

6 Feb 2025
This reaudit report was prepared by DefiMoon for Hype.

Audit information

Description	Solana-based token contract, with USDC-deposit issuance mechanism
Audited files	/src/*
Timeline	1 Jan 2025 - 6 Feb 2025
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Rust
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/hypewatch/smart-contract-v2/commit/4c512d334919ed45f6bedd3111d0dba444f7bdbf
Network	Solana
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 have been found at reaudit

The protocol code implements a Solana-based system for creating and managing user tokens using USDC collateral. The contract includes functionalities for minting, burning, and validating tokens, as well as managing roles such as operators and validators. The system also features an exponential bonding curve for token valuation and commission distribution mechanisms.

The overall structure and functionality of the code follow Solana's best practices and general principles of on-chain token management. However, the analysis has identified several critical issues, including unsafe usage of Rust and unchecked logic in key areas, which pose significant risks to security and functionality.

The following vulnerabilities were discovered:

- High risk issues concerning unsafe operations and unchecked access control.
- Medium risk issues regarding logic inconsistencies in fee calculations and account validation.
- Informational findings about optimization and readability.

All issues have been addressed during the reaudit and were either resolved completely, or mitigated partially, making the project safe for operation.

Summary of findings

ID	Description	Severity	Status
DFM-1	Unsafe use of `unsafe` blocks in critical functions	High Risk	Resolved
DFM-2	Inadequate Access Control for Validator Role	High Risk	Mitigated
DFM-3	Lack of Validation for Input Parameters	Medium Risk	Resolved
DFM-4	Gas Inefficiency in Mint and Burn Logic	Low Risk	Mitigated
DFM-5	Insufficient Error Handling	Low Risk	Resolved
DFM-6	Redundant Data Storage	Informational	Mitigated

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

Program version not specified	Passed
Program 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 loop	Passed
High consumption storage	Passed
“Out of Gas” Attack	Passed

Findings

DFM-1 «Unsafe Operations in Core Functions»

Severity: High Risk

Status: ✅ Resolved

Description: The contract relies heavily on unsafe Rust code, particularly in functions such as `mint` and `burn` within `src/program/processor/mint.rs` and `src/program/processor/burn.rs`. These functions use unsafe blocks to dereference pointers without ensuring memory safety, which could lead to undefined behavior, memory corruption, or data races under specific conditions. This undermines the reliability and security of the protocol.

Recommendation: Refactor the code to eliminate unsafe blocks where possible. Use safe Rust alternatives, such as borrowing and lifetimes, to ensure memory safety. If unsafe code is unavoidable, carefully document and validate its usage with proper runtime checks.

Changes observed: The `unsafe` blocks in mint.rs and burn.rs have been removed, replaced with structured abstractions (`RootState`, `ClientState`, `TokenState`), which handle state management safely.
Functions now rely on safe Rust practices, removing direct pointer dereferencing.

DFM-2 «Inadequate Access Control for Validator Role»

Severity: High Risk

Status: 🟡 Mitigated

Description: In the `change_token_status` function (`src/program/processor/change_token_status.rs`), access control for the validator role is not thoroughly enforced. A malicious actor could potentially bypass validation logic and gain unauthorized access to set or remove the `verified` status of tokens. This poses a significant risk to the integrity of the protocol.

Recommendation: Implement a robust role-based access control (RBAC) mechanism. Ensure that only the designated validator accounts can call this function. Use a whitelist or permissions map to enforce proper validation.

Changes observed:

- The `change_token_status` function in change_token_status.rs now validates that the signer is the designated validator.

- However, the check is still performed inline without a centralized access control mechanism.

- This leaves potential for misconfigurations or oversight in other role-dependent functions.

DFM-3 «Lack of Validation for Input Parameters»

Severity: Medium Risk

Status: ✅ Resolved

Description: Several functions, such as `initialize_holder` (`src/program/processor/initialize_holder.rs`) and `add_operator` (`src/program/processor/add_operator.rs`), do not adequately validate input parameters, such as account keys and instruction data. This lack of validation can lead to unintended behavior or unauthorized actions.

Recommendation: Add strict validation for all input parameters, including checks for account ownership, signatures, and data format. Use helper functions to standardize parameter validation across the codebase.

Changes observed:

- Functions such as `initialize_holder.rs` and `add_operator.rs` now use structured `new()` methods (e.g., `HolderState::new(holder_acc, program_id)`) to validate input parameters before execution.
- Additional error handling and ownership checks were added.

DFM-4 «Gas Inefficiency in Mint and Burn Logic»

Severity: Low Risk

Status: 🟡 Mitigated

Description: The `mint` and `burn` functions rely on iterative calculations to determine costs and fees based on the bonding curve. This approach increases computation costs and could lead to inefficiencies as the number of transactions grows.

Recommendation: Optimize the bonding curve calculations by precomputing reusable values or using mathematical approximations. Store intermediate results in state variables where appropriate to minimize redundant computations.

Changes observed:

- The bonding curve calculations still involve real-time computations in mint.rs and burn.rs.
- However, redundant state reads have been reduced, and intermediary values are cached in `RootState` for reuse.

DFM-5 «Insufficient Error Handling»

Severity: Low Risk

Status: ✅ Resolved

Description: In multiple locations across the contract, such as `log_new_client` (`src/state/log.rs`) and `withdraw_operator_funds` (`src/program/processor/withdraw_operator_funds.rs`), error handling is minimal or absent. This can lead to situations where errors propagate silently, making debugging and maintenance more challenging.

Recommendation: Enhance error handling by using descriptive error messages and adding checks to handle unexpected cases. Implement a centralized error management strategy to maintain consistency across the codebase.

Changes observed:

- Functions such as withdraw_operator_funds.rs now check account state **before executing transfers**.
- Structured error handling via `Result<>` is now present in most processor functions.

DFM-6 «Redundant Data Storage»

Severity: Informational

Status: 🟡 Mitigated

Description: The `RootAccount` structure in `src/state/data_structure.rs` stores several fields, such as `clients_count` and `tokens_count`, that could be calculated on the fly. This redundancy increases the storage cost and complicates data management.

Recommendation: Remove redundant fields and calculate their values dynamically when needed. This approach will reduce storage overhead and streamline the contract's state management.

Changes observed:

- Some redundant fields in `RootAccount` and `HolderAccount` have been removed.
- However, fields like `clients_count` and `tokens_count` **still exist**, despite being derivable from state data.

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