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

Rome, Italy, 00165

Smart contract's
Audit report

ValidV2

January, 2025

DEFIMOON

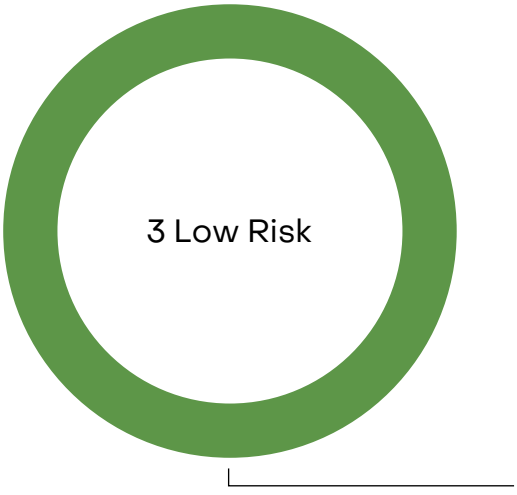
be secure

30 January 2025

This audit report was prepared by DefiMoon for ValidV2.

Audit information

Audited files	/contracts/*, excluding /contracts/interface
Timeline	25 Jan 2025 - 30 Jan 2025
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/yodaplus/valid-v2-contracts/tree/dev/contracts
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

The **ValidV2 Protocol** is a complex smart contract system designed for node staking, reward distribution, and liquid staking derivative (LSD) management. The protocol comprises multiple interconnected contracts responsible for creating and managing nodes, issuing tokens, handling investments, and distributing rewards.
Our audit focused on identifying vulnerabilities in security, access control, gas optimization, and economic stability across the following core contracts:

- **`NodeContract.sol`** – Manages node staking, token issuance, and rewards.
- **`Token.sol`** – Implements a custom ERC20 token with controlled burning and minting mechanisms.
- **`TokenCreator.sol`** – Handles the deployment of new tokens and their ownership transfers.
- **`ValidAdmin.sol`** – Governs the protocol's administrative functions and role-based permissions.
- **`LSDAdmin.sol`** & **`LSDInventory.sol`** – Oversee liquidity management, staking fees, and reward accrual for liquid staking.
- **`XDCValidator.sol`** – Implements validator functions, including voting, staking, and candidate management.

Summary of findings

ID	Description	Severity
DFM-1	Incorrect Handling of Incoming ETH	Low Risk
DFM-2	Inefficient Gas Usage in Withdrawal Tracking	Low Risk
DFM-3	Vulnerability to Front-Running in Staking	Low Risk

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	Not 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	Not passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Findings

DFM-1 «Incorrect Handling of Incoming ETH»

Severity: Low

Description: The `receive()` function in `NodeContract.sol` treats all incoming ETH as staking rewards, adding them to `CURRENT_REWARD_ACCUMALTED`. However, since ETH can be sent to the contract by accident, the contract may mistakenly treat unintended deposits as rewards, leading to inaccurate calculations.

Recommendation:
- Implement a whitelist of expected ETH senders to prevent unexpected deposits.
- Introduce a **fallback function** that **rejects unintended ETH transfers** instead of automatically adding them to rewards.
- Require **explicit function calls** for staking deposits rather than using `receive()`.

DFM-2 «Inefficient Gas Usage in Withdrawal Tracking»

Severity: Low

Description: The function `getWithdrawBlockNumber()` in `NodeContract.sol` iterates over an array of withdrawal block numbers **without optimization**. As the number of withdrawals increases, this function **incurs progressively higher gas costs**, which could **lead to out-of-gas errors**.

Recommendation:
- Use **mapping-based storage** instead of arrays to **enable direct access** rather than iteration.
- Store **indexed references** to track withdrawal block numbers **more efficiently**.
- Implement **batch processing** to handle large arrays in multiple transactions.

DFM-3 «Vulnerability to Front-Running in Staking»

Severity: Low

Description: The `stake()` function in `NodeContract.sol` does not implement any protection against front-running attacks. Malicious actors can observe pending transactions and manipulate staking amounts or execution order, leading to unfair advantages in reward allocation.

Recommendation:

- Implement **commit-reveal mechanisms** for staking to prevent front-running.
- Introduce gas-based anti-front-running measures, such as randomized stake execution times.
- Use batch processing for staking transactions, preventing attackers from predicting order execution.

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