Security Audit Report

NodeOps Network

v1.0

June 25, 2025

Table of Contents

Table of Contents **2**
License **4**
Disclaimer **5**
Introduction **6**
Purpose of This Report 6
Codebase Submitted for the Audit 6
Methodology 7
Functionality Overview 7
Audit Summary 8
How to Read This Report **9**
Code Quality Criteria **10**
Summary of Findings **11**
Detailed Findings **13**
1. Flawed time-based minting schedule permits excessive minting 13
2. Cross-chain signature replay vulnerability in AirdropPhase2 and RevenueVault contracts 13
3. Utilizing mintBuffer could lead to wrong NODE amounts being minted 14
4. Using distributeEmissions in StakingV2 could lead to stuck rewards 14
5. Users should be able to unstake if a pool is deactivated 15
6. Re-staking resets the lock period, leading to unfair lock extensions 15
7. NodeVesting will indicate an invalid available balance when vesting schedules get revoked 16
8. The mint buffer in NODE cannot be properly utilized 17
9. Missing initialization of offchainRecipient leads to potential token loss 17
10. Missing input validation in initialize functions 18
11. Inconsistent role assignment for pause and unpause functions 18
12. Unbounded iteration could disable RPC calls 19
13. Global rate limiting per address blocks independent vesting schedule releases 19
14. First token mint could end up locking the minting functionality for MIN_MINT_INTERVAL, without actually minting anything 20
15. Improper scaling in rewardPerTokenStored enables reward calculation anomalies 20
16. The StakingV2 contract will accumulate dust amounts 21
17. claimAllRewards in StakingV2 will revert if the user has unstaked all tokens beforehand 21
18. Lack of input validation in setOffchainRecipient enables misconfiguration 22
19. Centralization risks 22
20. Unbounded iteration in emission share validation risks DoS 23
21. Incomplete burn event metadata 23

22. Use proper scientific notation 24
23. Users can deposit small amounts in RevenueManager to avoid sending funds to the burnPool 24
24. For the first deposit in RevenueManager, burnPoolDepositId will be 0, even with no funds sent to the burnPool 25
25. getBurnSplitPercentage will show 0 for burn splits that are lower than 1% 25
26. Unused errors 25
27. Unused constants 26
28. Unused structs 26
29. Use custom errors in AirdropPhase2 27
30. Unused imports in StakingV2 and NodeVesting 27
31. Unnecessary if statement in NodeVesting 27
32. Usage of inefficient loop operations 28
33. Usage of wide Solidity pragma 28
34. Define and use constant variables instead of using literals 29
35. Unnecessary calls to setRoleAdmin for specific roles and DEFAULT_ADMIN_ROLE 29
36. Inefficient access pattern in _updateUserRewards 30
Appendix: Test Cases 31
1. Test case for “Flawed time-based minting schedule permits excessive minting” 31
2. Test case for “The mint buffer in NODE cannot be properly utilized” 32
3. Test case for “rewardPerTokenStored in StakingV2 can be inflated to very high amounts” 32
4. Test case for “StakingV2 will accumulate dust amounts” 33

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON THE COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by NodeLas Labs Limited to perform a security audit of the NodeOps Network smart contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities that could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/NodeOps-app/contracts-public
Commit	`b5cefb449c76de2cf5404dfb7259f83b235bded5`
Scope	All contracts were in scope.
Fixes verified at commit	`aac69a42ee81e384207394d3a30a75d149974e0d` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The NodeOps Network protocol is centered around the NODE token, which interacts with a suite of smart contracts to manage staking, revenue, vesting, and distribution mechanisms. At its core is a multi-pool staking system (StakingV2), allowing users to lock NODE tokens for various durations in exchange for emissions-based rewards allocated proportionally across different pools.

Revenue generated by the protocol is processed through the RevenueManager contract, which allocates incoming funds between a token burn mechanism and operational revenue, based on adjustable split ratios. Tokens designated for burning are routed to the BurnPool contract, where they are permanently removed from circulation, contributing to deflationary tokenomics.

Additional components include the RevenueVault, which ensures secure fund custody and supports signature-based withdrawals, and the NodeVesting system, designed to distribute tokens according to predefined cliff and linear vesting schedules. The protocol also features a dedicated airdrop mechanism for token distribution.

All essential contracts are upgradeable via the UUPS proxy pattern and incorporate role-based access control and pausability

Audit Summary

This audit resulted in a total of 36 findings, which were categorized based on severity as follows: 2 critical, 5 major, 12 minor, and 17 informational issues.

The vast majority of the issues identified during the audit have been addressed. Specifically, 29 findings were marked as resolved, while the remaining 7 were acknowledged by the development team for future consideration. No issues remain pending or partially resolved at the time of this report.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, leading to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits, and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	The client provided documentation for the largest contract.
Test coverage	Medium-High	`forge coverage` computes a test coverage of `84.68%`

Summary of Findings

No	Description	Severity	Status
1	Flawed time-based minting schedule permits excessive minting	Critical	Resolved
2	Cross-chain signature replay vulnerability in AirdropPhase2 and RevenueVault contracts	Critical	Acknowledged
3	Utilizing mintBuffer could lead to wrong NODE amounts being minted	Major	Resolved
4	Using `distributeEmissions` in StakingV2 could lead to stuck rewards	Major	Resolved
5	Users should be able to unstake if a pool is deactivated	Major	Resolved
6	Re-staking resets the lock period, leading to unfair lock extensions	Major	Resolved
7	NodeVesting will indicate an invalid available balance when vesting schedules get revoked	Major	Resolved
8	The mint buffer in NODE cannot be properly utilized	Minor	Resolved
9	Missing initialization of `offchainRecipient` leads to potential token loss	Minor	Resolved
10	Missing input validation in `initialize` functions	Minor	Resolved
11	Inconsistent role assignment for pause and unpause functions	Minor	Acknowledged
12	Unbounded iteration could disable RPC calls	Minor	Resolved
13	Global rate limiting per address blocks independent vesting schedule releases	Minor	Resolved
14	First token mint could end up locking the minting functionality for `MIN_MINT_INTERVAL`, without actually minting anything	Minor	Resolved
15	Improper scaling in `rewardPerTokenStored` enables reward calculation anomalies	Minor	Resolved
16	The StakingV2 contract will accumulate dust amounts	Minor	Resolved

17	`claimAllRewards` in StakingV2 will revert if the user has unstaked all tokens beforehand	Minor	Resolved
18	Lack of input validation in `setOffchainRecipient` enables misconfiguration	Minor	Resolved
19	Centralization risks	Minor	Acknowledged
20	Unbounded iteration in emission share validation risks DoS	Informational	Acknowledged
21	Incomplete burn event metadata	Informational	Resolved
22	Use proper scientific notation	Informational	Resolved
23	Users can deposit small amounts in RevenueManager to avoid sending funds to the `burnPool`	Informational	Acknowledged
24	For the first deposit in RevenueManager, `burnPoolDepositId` will be `0`, even with no funds sent to the `burnPool`	Informational	Resolved
25	`getBurnSplitPercentage` will show `0` for burn splits that are lower than 1%	Informational	Acknowledged
26	Unused errors	Informational	Acknowledged
27	Unused constants	Informational	Acknowledged
28	Unused structs	Informational	Resolved
29	Use custom errors in AirdropPhase2	Informational	Resolved
30	Unused imports in StakingV2 and NodeVesting	Informational	Acknowledged
31	Unnecessary `if` statement in NodeVesting	Informational	Resolved
32	Usage of inefficient loop operations	Informational	Resolved
33	Usage of wide Solidity pragma	Informational	Resolved
34	Define and use `constant` variables instead of using literals	Informational	Resolved
35	Unnecessary calls to `setRoleAdmin` for specific roles and `DEFAULT_ADMIN_ROLE`	Informational	Resolved
36	Inefficient access pattern in `_updateUserRewards`	Informational	Resolved

Detailed Findings

1. Flawed time-based minting schedule permits excessive minting

Severity: Critical

In `contracts/ERC20/NODE.sol:72-107`, the `mint` function allows the `MINTER_ROLE` to mint tokens depending on a constrained schedule.

However, although it correctly calculates the number of days elapsed since the last mint via the `timeElapsed` and `daysElapsed` variables, it fails to advance the `nextMint` timestamp by the total number of elapsed days.

Instead, it increments `nextMint` by a single day (`MIN_MINT_INTERVAL`) regardless of how many days have passed.

This allows a minter to repeatedly mint the full allowance for multiple days, effectively bypassing the intended constrained schedule.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend advancing `nextMint` by the total elapsed interval, multiplying `MIN_MINT_INTERVAL` by `daysElapsed`.

Status: Resolved

2. Cross-chain signature replay vulnerability in AirdropPhase2 and RevenueVault contracts

Severity: Critical

In `contracts/pools/custom/AirdropPhase2.sol:81` and `contracts/vaults/RevenueVault.sol:126`, a signature-based token claiming mechanism is implemented that utilizes message hashes from `msg.sender`, `claim`, and `amount` parameters.

While the protocol correctly prevents signature reuse within each chain by maintaining used signature mappings and requiring unique claims, the signature verification lacks chain ID validation.

Since the protocol will deploy on Ethereum, Arbitrum, and Base, identical wallet addresses can generate the same valid signatures across all chains for the same claim parameters. This enables users to replay signatures across different chains, effectively claiming more tokens than intended.

Recommendation

We recommend adding `block.chainid` to the message hash or implementing `EIP-712` structured hashing with chain-specific domain separators.

Status: Acknowledged

3. Utilizing `mintBuffer` could lead to wrong `NODE` amounts being minted

Severity: Major

In `contracts/ERC20/NODE.sol`, the `MINTER_ROLE` is permitted to invoke minting up to one hour before the defined `MIN_MINT_INTERVAL` by leveraging the `mintBuffer`.

This early access mechanism introduces a discrepancy in time calculations due to integer division. Specifically, if a minter uses the buffer to mint at the 23-hour mark and then waits for two full `MIN_MINT_INTERVALs` (48 hours), the total elapsed time will only be 47 hours from the perspective of the contract.

Consequently, the `daysElapsed` variable will be computed as `1` instead of `2`, truncating the value due to integer division.

This miscalculation results in the minter receiving only half the expected mintable amount and delays the next eligible minting window by another full interval.

Recommendation

We recommend setting `nextMint` to be `block.timestamp + MIN_MINT_INTERVAL`, or just remove `mintBuffer` to prevent any potential issues due to it.

Status: Resolved

4. Using `distributeEmissions` in `StakingV2` could lead to stuck rewards

Severity: Major

In `contracts/staking/StakingV2.sol:165`, when the `poolEmissions` are being distributed, they are divided by `10000`, which creates an implicit requirement that the sum of all pools' emissions must equal exactly `10000`.

When this assumption is violated, the distributed rewards become mathematically incorrect, leading to unattainable or disproportionate reward allocations across pools.

Additionally, even when the total emissions correctly sum to `10000`, the function continues to transfer rewards to pools with zero staked tokens (`totalStaked == 0`), effectively locking those rewards in pools where no stakers exist to claim them.

Recommendation

We recommend utilizing the `_validateTotalEmissionShares` function every time `distributeEmissions` is used, and to only transfer funds if `totalStaked > 0` for any pool.

Status: Resolved

5. Users should be able to unstake if a pool is deactivated

Severity: Major

In `contracts/staking/StakingV2.sol:213`, users must wait for the `lockEndTime` until they can unstake their tokens from a specific pool.

However, the current implementation allows pools to be deactivated, meaning that no rewards will be accumulated when distributed.

If a user has staked funds in a pool that gets deactivated, they are essentially blocked from utilizing these funds up to the pool's lock period, as they will not receive any rewards for their stake.

Additionally, there is no mechanism in place to reactivate a pool.

Recommendation

We recommend allowing users to unstake without waiting for the full `lockEndTime` if a pool becomes deactivated.

Status: Resolved

6. Re-staking resets the lock period, leading to unfair lock extensions

Severity: Major

In `contracts/staking/StakingV2.sol:184-212` the `stake` function allows users to stake and lock their tokens.

However, the lock period for a user's stake is unconditionally reset on every new deposit. Specifically, the execution overwrites any existing lock time regardless of how long the user

has already been staked. This behavior penalizes users who add to their stake late in the lock period.

For instance, if a user initially stakes in a 30-day pool and adds more tokens 25 days later, their entire stake is locked for another 30 days from the second deposit, extending their effective lock time to 55 days.

This leads to a situation where users lose the benefit of staked time and can be perpetually penalized for increasing their stake.

Recommendation

We recommend redesigning the staking logic to avoid penalizing users on repeated stakes.

Options include:

- Tracking individual lock periods per deposit and calculating a weighted average.
- Offering multiple stake "slots" or tranches per user to allow independent lock durations.

Status: Resolved

7. NodeVesting will indicate an invalid available balance when vesting schedules get revoked

Severity: Major

In `contracts/vesting/NodeVesting.sol:539` and `contracts/vesting/NodeVesting.sol:635`, the `availableBalance` will show a positive value even when all funds have been allocated and released (even the revoked ones).

This is because, when a schedule is revoked, and the revoked funds are re-funded by the admin again, the `totalRevoked` variable is not reduced, leading to double-counting.

This will lead to admins obtaining invalid information regarding the vesting contract's balance and creating schedules that cannot be properly executed.

Recommendation

We recommend adding a boolean flag to the `fundPool` function to indicate if the funds are coming from a revoked schedule, and reducing `totalRevoked` accordingly.

Status: Resolved

8. The mint buffer in NODE cannot be properly utilized

Severity: Minor

In `contracts/ERC20/NODE.sol:81`, the `MINTER_ROLE` is allowed to mint tokens up to one hour before the `MIN_MINT_INTERVAL` by using the `mintBuffer`.

However, once this buffer is utilized, subsequent attempts to use it again will fail unless a full `MIN_MINT_INTERVAL` has elapsed.

Specifically, the condition `block.timestamp < nextMint - mintBuffer` will remain `true` if the minter attempts to mint again at the 23-hour mark on the following day.

This leads to a revert, effectively preventing consistent use of the `mintBuffer` and limiting its intended utility.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend setting `nextMint` to `block.timestamp + MIN_MINT_INTERVAL`.

Status: Resolved

9. Missing initialization of `offchainRecipient` leads to potential token loss

Severity: Minor

In `contracts/core/RevenueManager.sol:101-152`, the `deposit` function calculates and distributes a split of deposited funds between the burn pool and an off-chain recipient.

However, the `offchainRecipient` address is not initialized in the contract's `initialize` function, leaving it at the zero address by default.

Consequently, if any deposits occur before an administrator explicitly sets the `offchainRecipient` via `setOffchainRecipient`, the revenue portion of those deposits will be irreversibly transferred to the zero address, resulting in permanent loss of funds.

Recommendation

We recommend initializing `offchainRecipient` to a valid, non-zero address in the `initialize` function.

Status: Resolved

10. Missing input validation in `initialize` functions

Severity: Minor

The `initialize` functions across the contracts lack essential input validation, exposing the system to potential misconfiguration risks:

- In `NODE.sol`, the `multisig` address is not checked against the zero address, allowing deployment with an invalid admin.
- In `StakingV2.sol`, neither `_admin` nor `_token` addresses are validated. Additionally, the `_token` should be validated against its expected interface to ensure compatibility.
- In `RevenueManager.sol`, `_token`, `_burnPool`, and `_admin` addresses are not checked for nullity. Moreover, `_token` and `_burnPool` should be validated to confirm interface compliance. The `_burnSplit` parameter lacks a check to ensure it does not exceed `10,000` (representing 100%).
- In `BurnPool.sol`, `_token` and `_admin` are assigned without ensuring they are valid (non-zero) addresses.
- In `AirdropPhase2.sol`, the `multisig` and `token` addresses are not validated. Additionally, the `_token` should be validated against its expected interface to ensure compatibility.
- In `RevenueVault.sol`, both `multisig` and `_tokenAddress` addresses are not validated. Additionally, the `_tokenAddress` should be validated against its expected interface to ensure compatibility.

These missing validations could lead to contract deployments with invalid or non-functional roles and dependencies, potentially locking up functionality or funds.

Recommendation

We recommend enhancing the validation logic in the `initialize` functions.

Status: Resolved

11. Inconsistent role assignment for pause and unpause functions

Severity: Minor

The `pause` and `unpause` functions in the contracts are protected by different roles: `pause` requires the `PAUSER_ROLE`, while `unpause` is restricted to the `DEFAULT_ADMIN_ROLE`.

This asymmetry in role assignment may lead to operational inefficiencies or confusion, particularly in emergency scenarios where the same entity might be expected to both pause and resume contract operations.

Recommendation

We recommend aligning the role requirements for both `pause` and `unpause` functions.

Status: Acknowledged

12. Unbounded iteration could disable RPC calls

Severity: Minor

The `BurnPool.sol` and `RevenueVault.sol` contracts implement the view functions `getAllDeposits` and `getAllClaims`, which construct and return arrays of potentially unbounded size through linear iteration.

These functions return full lists of deposit IDs or claim IDs by iterating over `depositCount` or `userClaims[user]`, respectively.

Consequently, these functions may exceed RPC nodes' gas limits (e.g., 10M gas on Infura, 30M on Alchemy, ~50M for go-ethereum), causing read-only calls to fail. Also, RPC providers may reject or timeout such calls, making the function inaccessible.

Recommendation

We recommend implementing pagination in the aforementioned functions.

Status: Resolved

13. Global rate limiting per address blocks independent vesting schedule releases

Severity: Minor

In `contracts/vesting/NodeVesting.sol`, the `release` function enforces a rate limit via `_checkRateLimit`, which relies on a single `lastReleaseTime[beneficiary]` mapping to throttle release frequency. This design applies a uniform cooldown period across all vesting schedules for a given address.

As a result, when a beneficiary invokes `release` for any `scheduleId`, the `lastReleaseTime` is updated globally.

Subsequent attempts to release from any other schedule, regardless of its independent configuration, will be blocked until `minReleaseInterval` elapses.

This unfairly restricts beneficiaries from claiming tokens from multiple schedules.

Recommendation

We recommend applying rate limiting per `beneficiary` and `scheduleId`.

Status: Resolved

14. First token mint could end up locking the minting functionality for `MIN_MINT_INTERVAL`, without actually minting anything

Severity: Minor

In `contracts/ERC20/NODE.sol:98`, during the first invocation of the minting function by the `MINTER_ROLE`, the minter may unintentionally lock the minting mechanism for the full duration of `MIN_MINT_INTERVAL` without actually minting any tokens.

This occurs because `maxMintAmount` is uninitialized at deployment and defaults to zero, resulting in a no-op mint while still updating the `lastMintTimestamp`.

Consequently, the minter is forced to wait an entire interval before being able to mint again.

Recommendation

We recommend either providing an initialization value to `maxMintAmount` or to revert the mint if `allowedMintAmount` is `0`.

Status: Resolved

15. Improper scaling in `rewardPerTokenStored` enables reward calculation anomalies

Severity: Minor

In `contracts/staking/StakingV2.sol:170`, the `rewardPerTokenStored` variable can be severely inflated (by up to `e18` times more) if the pool's `totalStaked` value is in dust amounts.

This is because `poolEmissions` are multiplied by `1e18`, expecting that the `totalStaked` amount will never be in dust amounts, which will not always be the case, as there is no minimum staking amount.

While this inflation currently has no exploitable impact, since `pendingRewards` in `contracts/staking/StakingV2.sol:278` are correctly downscaled by `1e18`, and `rewardPerTokenPaid` is synchronized with `rewardPerTokenStored`, this could become a vector for manipulation if `rewardPerTokenStored` is referenced elsewhere without appropriate scaling or safeguards.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend adding a minimum staking amount requirement as well as a proper unstaking check to ensure that the staked amount is never in the range of dust amounts.

Status: Resolved

16. The StakingV2 contract will accumulate dust amounts

Severity: Minor

In `contracts/staking/StakingV2.sol`, there are multiple places where integer division is taking place, meaning that dust amounts of `NODE` tokens will gradually accumulate.

However, there is no dust sweeping functionality, meaning that this dust amount will remain stuck in the contract.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend adding a dust sweeping function, which would allow admins to pull these dust amounts.

Status: Resolved

17. `claimAllRewards` in StakingV2 will revert if the user has unstaked all tokens beforehand

Severity: Minor

In `contracts/staking/StakingV2.sol:260`, the `claimAllRewards` function checks all `userActivePools` before calculating rewards.

However, when a user unstakes all its tokens from a pool, that pool is removed from the `userActivePools`.

Because of this, the `claimAllRewards` function will not loop over the pending user rewards and will revert when called. The same issue will be observed with the `getUserTotalPendingRewards` function, which will return 0.

Recommendation

We recommend enforcing users to collect their rewards before fully unstaking.

Status: Resolved

18. Lack of input validation in `setOffchainRecipient` enables misconfiguration

Severity: Minor

In `contracts/core/RevenueManager.sol:174-176`, the `setOffchainRecipient` function allows the `DEFAULT_ADMIN_ROLE` to update the `offchainRecipient` address.

However, the function does not validate the provided address. As a result, it is possible to set this recipient to the zero address, accidentally

Since this address is used to forward the revenue portion of deposits, setting it to `address(0)` would result in irreversible token loss on future deposits, undermining protocol integrity and causing potential financial damage.

Recommendation

We recommend validating addresses before storing them in the contract.

Status: Resolved

19. Centralization risks

Severity: Minor

Multiple contracts grant unrestricted authority to a single role, `DEFAULT_ADMIN_ROLE`, enabling critical operations without checks or balances.

In `NODE.sol`, this role can mint tokens without following the schedule.

In `RevenueManager.sol`, the `admin` can alter financial parameters such as `burnSplit`, change burn destination addresses, or redirect off-chain recipients, potentially rerouting entire revenue flows.

Similarly, `AirdropPhase2.sol` and `NodeVesting.sol` allow this role to withdraw any amount of tokens from the contract, including emergency withdrawals, without validation against allocated or vested balances.

These permissions expose the system to abuse by a compromised or malicious admin, undermining trust and security guarantees.

Recommendation

We recommend implementing a multi-signature governance mechanism for all functions currently restricted to `DEFAULT_ADMIN_ROLE`, ensuring actions require consensus among

multiple trusted parties. Also, implementing monitoring mechanisms and a structured rotation of the signing keys is recommended.

Status: Acknowledged

20. Unbounded iteration in emission share validation risks DoS

Severity: Informational

In `contracts/staking/StakingV2.sol`, the internal functions `_validateEmissionSharesNotExceeding100`, `_validateTotalEmissionShares`, and `distributeEmissions` perform linear iteration over all pools (`totalPools`) to calculate the cumulative emission shares of active pools. These functions are used to validate that:

- `_validateEmissionSharesNotExceeding100` ensures the sum of active pool emission shares does not exceed 100% (`10,000` basis points).
- `_validateTotalEmissionShares` requires that the sum of active emission shares equals exactly 100%.
- `distributeEmissions` distributes tokens proportionally to each active pool based on their share, updating internal state and emitting events.

However, because they iterate through `totalPools`, a sufficiently large pool count could cause these validations to exceed block gas limits. This would result in failed transactions during emission updates or pool creation, effectively locking the staking system and creating a denial of service (DoS) vector.

Recommendation

We recommend enforcing a configurable upper limit on `totalPools` to cap computational complexity.

Status: Acknowledged

21. Incomplete burn event metadata

Severity: Informational

In `contracts/pools/custom/BurnPool.sol:110-120`, the `burn` function emits a `Burned` event upon successful token destruction.

However, the event includes hardcoded placeholder values, `0` for the deposit ID and `address(0)` for the depositor address. This omission of meaningful context prevents observers from correlating burn events with specific deposits or users.

Without accurate metadata, it becomes impossible to audit or trace which deposits contributed to each burn operation.

Recommendation

We recommend updating the `burn` function to include relevant deposit and user metadata in the `Burned` event.

Status: Resolved

22. Use proper scientific notation

Severity: Informational

In `contracts/ERC20/NODE.sol:21`, the `MAX_SUPPLY` constant should be written as `1_000_000_000e18`.

Recommendation

We recommend using `e18` instead of `10**18` throughout the code.

Status: Resolved

23. Users can deposit small amounts in RevenueManager to avoid sending funds to the `burnPool`

Severity: Informational

In `contracts/core/RevenueManager.sol:108`, the absence of a minimum amount constraint allows malicious users to send dust-level token amounts that, due to the `burnSplit` calculation, result in zero value being allocated to the `burnPool`.

This behavior can be exploited to bypass the intended burn mechanism entirely. The risk is further amplified on L1/L2 roll-ups where transaction fees are minimal, making such micro-transactions economically viable for abuse.

Recommendation

We recommend enforcing a minimum deposit amount.

Status: Acknowledged

24. For the first deposit in RevenueManager, `burnPoolDepositId` will be 0, even with no funds sent to the `burnPool`

Severity: Informational

In `contracts/core/RevenueManager.sol:133`, the `burnPoolDepositId` is set as part of the `DepositInfo` struct.

However, this value will be `0` for the very first burn, and will be `0` if there was no burn deposit as well, as the `BurnPool` returns `0` for its first deposit.

This is confusing and could lead to invalid off-chain updates.

Recommendation

We recommend changing the burn pool deposit IDs to start from `1`, allowing `0` to be set for deposits that do not invoke the `BurnPool`.

Status: Resolved

25. `getBurnSplitPercentage` will show 0 for burn splits that are lower than `1%`

Severity: Informational

In `contracts/core/RevenueManager.sol:211`, the `getBurnSplitPercentage` function will show 0 for any `burnSplit < 100`, due to integer division.

This means that ranges from 0.1% to 0.9% will be truncated to 0, leading to invalid front-end representations.

Recommendation

We recommend setting a minimum `burnSplit` percentage.

Status: Acknowledged

26. Unused errors

Severity: Informational

There are numerous unused errors throughout the codebase that are redundant and should be removed to reduce contract size and deployment costs.

Recommendation

We recommend removing the following lines of code:

- `contracts/libraries/Errors.sol:9-13`,
- `contracts/libraries/Errors.sol:15-16`,
- `contracts/libraries/Errors.sol:21-32`,
- `contracts/libraries/Errors.sol:34-43`,
- `contracts/libraries/Errors.sol:46-51`,
- `contracts/libraries/Errors.sol:55-56`,
- `contracts/libraries/Errors.sol:58`,
- `contracts/libraries/VestingErrors.sol:133-139`,
- `contracts/libraries/VestingErrors.sol:150-169`

Status: Acknowledged

27. Unused constants

Severity: Informational

There are numerous unused constants throughout the codebase that are redundant and should be removed to reduce contract size and deployment costs.

Recommendation

We recommend removing the following lines of code:

- `contracts/libraries/Roles.sol:5-12`,
- `contracts/libraries/Roles.sol:15`

Status: Acknowledged

28. Unused structs

Severity: Informational

There are numerous unused structs throughout the codebase that are redundant and should be removed to reduce contract size and deployment costs.

Recommendation

We recommend removing the following lines of code:

- `contracts/libraries/VestingTypes.sol:69-88`

Status: Resolved

29. Use custom errors in AirdropPhase2

Severity: Informational

While the codebase generally adopts custom errors consistently, the contract located at `contracts/pools/custom/AirdropPhase2.sol` does not fully adhere to this practice.

Instead, it relies on revert strings or omits error handling in certain cases, which is inconsistent with the rest of the codebase.

Recommendation

We recommend using custom errors everywhere in the AirdropPhase2 contract.

Status: Resolved

30. Unused imports in StakingV2 and NodeVesting

Severity: Informational

Unused imports should be removed from the codebase.

Recommendation

We recommend removing unused imports in `contracts/staking/StakingV2.sol:14` and `contracts/vesting/NodeVesting.sol:13`.

Status: Acknowledged

31. Unnecessary `if` statement in NodeVesting

Severity: Informational

In `contracts/vesting/NodeVesting.sol:523`, there is an `if` statement checking whether the schedule's `vestingDuration > 0`.

This is unnecessary as `vestingDuration` is validated to always be `> MIN_VESTING_DURATION`, which is set to one day and will always be `> 0`.

Recommendation

We recommend removing the unnecessary `if` statement.

Status: Resolved

32. Usage of inefficient loop operations

Severity: Informational

Invoking `SSTORE` operations in loops increases gas costs. Use local variable caching for the following for loop lengths.

Recommendation

We recommend caching array lengths in the following lines code:

- `contracts/staking/StakingV2.sol:128`,
- `contracts/staking/StakingV2.sol:143`,
- `contracts/staking/StakingV2.sol:162`,
- `contracts/staking/StakingV2.sol:348`
- `contracts/vesting/NodeVesting.sol:227`,
- `contracts/vaults/RevenueVault.sol:166`,
- `contracts/vaults/RevenueVault.sol:174`,
- `contracts/vaults/RevenueVault.sol:158`,
- `contracts/pools/custom/BurnPool.sol:170`

Status: Resolved

33. Usage of wide Solidity pragma

Severity: Informational

In `contracts/libraries/Errors.sol:2`, use a specific Solidity pragma instead of a wide one.

Recommendation

We recommend changing the pragma to `0.8.26`.

Status: Resolved

34. Define and use `constant` variables instead of using literals

Severity: Informational

Throughout the code, literals are used to represent values such as total emissions or percentages, which could prove hard to maintain where the same value needs to be changed in different places.

Recommendation

We recommend creating constant state variables and referencing them throughout the contracts:

- `contracts/core/RevenueManager.sol`,
- `contracts/pools/custom/BurnPool.sol`,
- `contracts/staking/StakingV2.sol`,
- `contracts/vesting/NodeVesting.sol`

Status: Resolved

35. Unnecessary calls to `setRoleAdmin` for specific roles and `DEFAULT_ADMIN_ROLE`

Severity: Informational

`DEFAULT_ADMIN_ROLE` is automatically the admin of all roles by default in OpenZeppelin's `AccessControl`.

This means that it is redundant to explicitly call `_setRoleAdmin(role, DEFAULT_ADMIN_ROLE)`.

Recommendation

We recommend removing the redundant logic in:

- `contracts/ERC20/NODE.sol`,
- `contracts/core/RevenueManager.sol`,
- `contracts/pools/custom/AirdropPhase2.sol`,
- `contracts/pools/custom/BurnPool.sol`,
- `contracts/staking/StakingV2.sol`,
- `contracts/vaults/RevenueVault.sol`,

- `contracts/vesting/NodeVesting.sol`

Status: Resolved

36. Inefficient access pattern in `_updateUserRewards`

Severity: Informational

The `_updateUserRewards` function, defined in `contracts/staking/StakingV2.sol:278`, is invoked frequently by the contract.

Before each invocation, `userStakes` and `pools` are typically queried, but the function redundantly queries them again internally.

This leads to unnecessary gas consumption due to repeated storage and memory reads.

Recommendation

We recommend sending `userStakes[user][poolId]` as a storage pointer, and `pools[poolId]` as a memory pointer, any time `_updateUserRewards()` is invoked, instead of just the user address and pool ID.

Status: Resolved

Appendix: Test Cases

1. Test case for "Flawed time-based minting schedule permits excessive minting"

```
function testMintUnlimited() public {
        vm.warp(1749495492);
        vm.startPrank(admin);
        nodeToken.setMaxMintAmount(100 ether);
        nodeToken.grantRole(Roles.MINTER_ROLE, admin);
        nodeToken.mint(address(this), 100 ether);
        vm.stopPrank();

        // Don't mint for 10 days
        vm.warp(block.timestamp + 10 days);

        vm.prank(admin);
        nodeToken.mint(address(this), 1000 ether);

        // From here on, the user can mint with an amount decreasing
with -maxMint until it reaches the days difference since the first mint
        vm.startPrank(admin);
        nodeToken.mint(address(this), 900 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 800 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 700 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 600 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 500 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 400 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 300 ether);

        vm.startPrank(admin);
```

```
        nodeToken.mint(address(this), 200 ether);

        vm.startPrank(admin);
        nodeToken.mint(address(this), 100 ether);

        vm.startPrank(admin);
        vm.expectRevert();
        nodeToken.mint(address(this), 100 ether);
    }
```

2. Test case for [“The mint buffer in NODE cannot be properly utilized”]

```
function testMintBuffer() public {
         vm.warp(1749495492);
        vm.startPrank(admin);
        nodeToken.setMaxMintAmount(100 ether);
        nodeToken.grantRole(Roles.MINTER_ROLE, admin);
        nodeToken.mint(address(this), 100 ether);
        vm.stopPrank();

        // Utilizing the buffer to mint more tokens
        vm.warp(block.timestamp + 1 days - 1 hours);

        vm.prank(admin);
        nodeToken.mint(address(this), 100 ether);

        // Try to utilize the buffer again, but it will revert
        vm.warp(block.timestamp + 1 days - 1 hours);

        vm.startPrank(admin);
        vm.expectRevert();
        nodeToken.mint(address(this), 100 ether);
    }
```

3. Test case for [“`rewardPerTokenStored` in `StakingV2` can be inflated to very high amounts”]

```
function testOverflowDistribution() public {
        vm.prank(poolManager);
        stakingV2.createPool(POOL_30_DAYS, 10000, "30 Day Pool"); // 50%
```

```
        vm.startPrank(alice);
        token.approve(address(stakingV2), 1);
        stakingV2.stake(0, 1);
        vm.stopPrank();

        vm.startPrank(emissionManager);
        token.approve(address(stakingV2), 100e18);
        stakingV2.distributeEmissions(100e18);
        vm.stopPrank();

        // Massive inflation of rewardPerTokenStored
        assertEq(stakingV2.getPoolInfo(0).rewardPerTokenStored, 100e36);
    }
```

4. Test case for "`StakingV2` will accumulate dust amounts"

```
function testDustAmount() public {
        vm.startPrank(poolManager);
        stakingV2.createPool(POOL_30_DAYS, 3000, "30 Day Pool");
        stakingV2.createPool(POOL_90_DAYS, 3000, "90 Day Pool");
        stakingV2.createPool(POOL_180_DAYS, 4000, "180 Day Pool");
        vm.stopPrank();

        vm.startPrank(alice);
        token.approve(address(stakingV2), 33333333333333333333);
        stakingV2.stake(0, 33333333333333333333);
        vm.stopPrank();

        vm.startPrank(alice);
        token.approve(address(stakingV2), 33333333333333333333);
        stakingV2.stake(1, 33333333333333333333);
        vm.stopPrank();

        vm.startPrank(alice);
        token.approve(address(stakingV2), 33333333333333333333);
        stakingV2.stake(2, 33333333333333333333);
        vm.stopPrank();

        vm.startPrank(bob);
        token.approve(address(stakingV2), 50505050505050505050);
        stakingV2.stake(0, 50505050505050505050);
        vm.stopPrank();
```

```
        vm.startPrank(bob);
        token.approve(address(stakingV2), 50505050505050505050);
        stakingV2.stake(1, 50505050505050505050);
        vm.stopPrank();

        vm.startPrank(bob);
        token.approve(address(stakingV2), 50505050505050505050);
        stakingV2.stake(2, 50505050505050505050);
        vm.stopPrank();

        vm.startPrank(charlie);
        token.approve(address(stakingV2), 123e18);
        stakingV2.stake(0, 123e18);
        vm.stopPrank();

        vm.startPrank(charlie);
        token.approve(address(stakingV2), 123e18);
        stakingV2.stake(1, 123e18);
        vm.stopPrank();

        vm.startPrank(charlie);
        token.approve(address(stakingV2), 123e18);
        stakingV2.stake(2, 123e18);
        vm.stopPrank();

        vm.startPrank(emissionManager);
        token.approve(address(stakingV2), 1000e18);
        stakingV2.distributeEmissions(1000e18);
        vm.stopPrank();

        vm.warp(block.timestamp + 200 days);

        vm.startPrank(alice);
        stakingV2.claimAllRewards();
        vm.stopPrank();

        vm.startPrank(bob);
        stakingV2.claimAllRewards();
        vm.stopPrank();

        vm.startPrank(charlie);
        stakingV2.claimAllRewards();
        vm.stopPrank();
```

```
        vm.startPrank(alice);
        stakingV2.unstake(0, 33333333333333333333);
        stakingV2.unstake(1, 33333333333333333333);
        stakingV2.unstake(2, 33333333333333333333);
        vm.stopPrank();

        vm.startPrank(bob);
        stakingV2.unstake(0, 50505050505050505050);
        stakingV2.unstake(1, 50505050505050505050);
        stakingV2.unstake(2, 50505050505050505050);
        vm.stopPrank();

        vm.startPrank(charlie);
        stakingV2.unstake(0, 123e18);
        stakingV2.unstake(1, 123e18);
        stakingV2.unstake(2, 123e18);
        vm.stopPrank();

        vm.startPrank(emissionManager);

        // Dust is accumulated
        assertGt(token.balanceOf(address(stakingV2)), 0);
    }
```