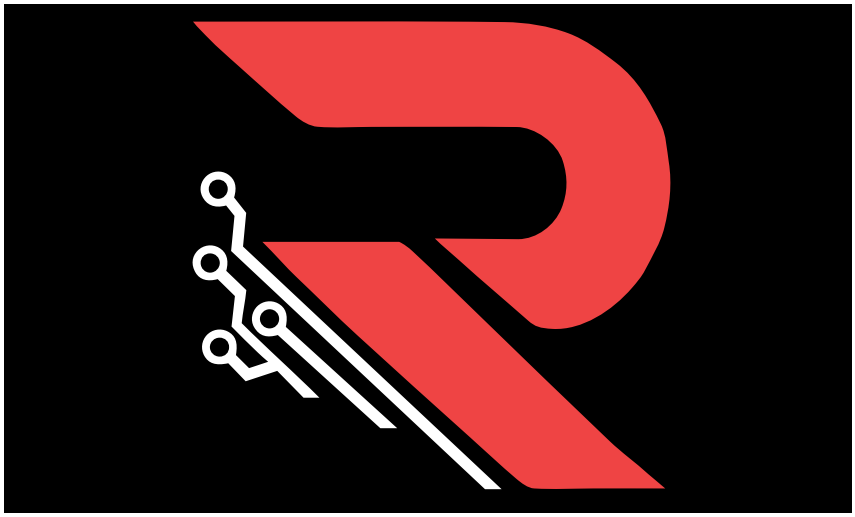
Nuon Security Review



Version 2.0

14.05.2025

Conducted by:

MaslarovK, Lead Security Researcher

radev-eth, Lead Security Researcher

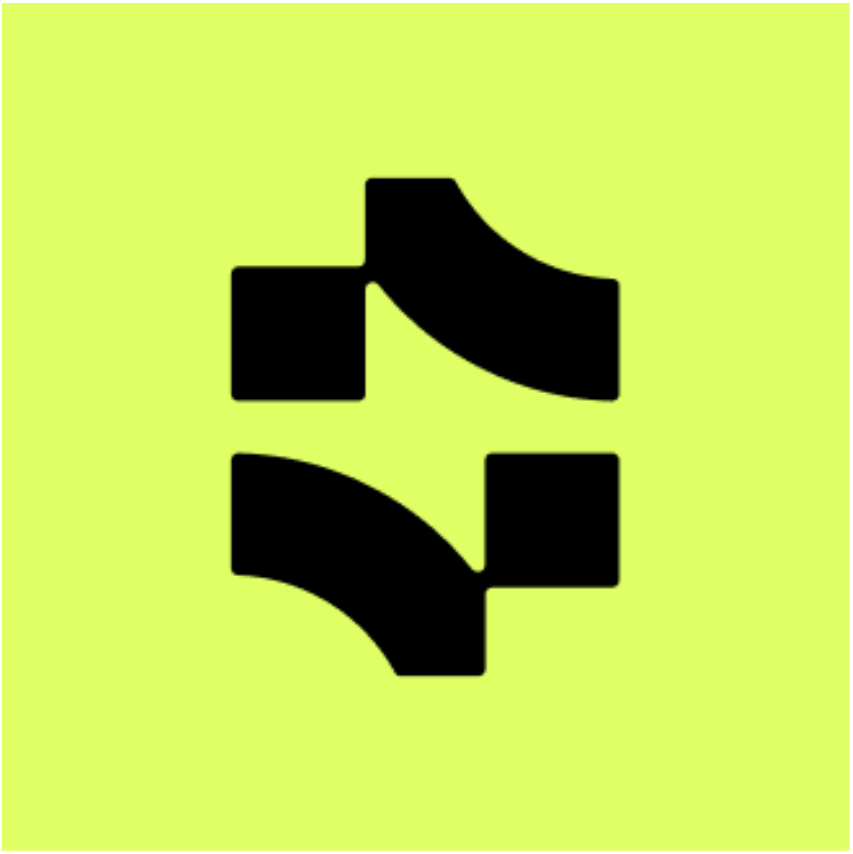


Table of Contents

1 About MaslarovK

MaslarovK a Security Reseacher and Co-Founder of Rezolv Solutions.

2 About radev.eth

radev_eth a Security Reseacher and Co-Founder of Rezolv Solutions.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	Nuon
Repository	https://github.com/nuonfinance/nuon-v2/
Commit hash	c501420ddf56fa7562b3b47193cff13d0b44f63e
Resolution	N/A
Documentation N/A	
Methods	Manual review

Scope

src/libraries/DutchAuction.sol
src/libraries/TreasuryLibrary.sol
src/Tokens/Nuon.sol
src/Tokens/WNuon.sol
src/Treasury/common/TreasuryCommon.sol
src/Treasury/common/TreasuryCommonAccessControl.sol
src/Treasury/common/TreasuryCommonPausable.sol
src/Treasury/common/TreasuryCommonReentrancyGuard.sol
src/Treasury/TreasuryAssetFacet.sol
src/Treasury/TreasuryAuctionsFacet.sol
src/Treasury/TreasuryBufferFacet
src/Treasury/TreasuryDiamondCutFacet.sol
src/Treasury/TreasuryStakingFacet.sol
src/Treasury/TreasuryStorageFacet.sol
src/VoteEscrow.sol
src/Backstop.sol
src/ChainlinkAutomation.sol
src/Diamond.sol
src/Governance.sol
src/InflationOracle.sol
src/NuonStaking.sol
src/OracleRegistry.sol
src/Redistributor.sol
src/ShutdownManager.sol
src/VoteAllocationManager.sol
src/VoteEscrow.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	3
Low risk	10
Informational	2

6 Findings

6.1 Medium risk

6.1.1 Oracle manipulation risk in rebalancing logic

Severity: *Medium risk*

Context: The `TreasuryBufferFacet` performs critical asset rebalancing based on price data fetched through `_getCurrentAssetValue`, which in turn depends on oracles (via `OracleRegistry.getAssetPrice`). These prices directly influence yield calculations, asset buy/sell operations, and Backstop payouts.

Description: The protocol relies heavily on accurate and timely oracle data for asset valuations. If an oracle is manipulated — either by delayed updates, compromised feeds, or flash-loan-based distortions — the following risks arise:

- **Mispriced Rebalances:** Assets could be overbought or undersold.
- **Backstop Exploitation:** Artificially inflated asset values could trigger inflated yield distributions.
- **Underfunding:** Assets may be undervalued, resulting in insufficient collateral for liabilities.

Currently, there is no evidence of TWAP enforcement, price deviation checks, or backup oracle fallbacks.

Recommendation:

- Introduce **oracle anchoring** (e.g., comparing Chainlink price to a TWAP or internal reference).
- Enforce **deviation thresholds** and **max stale periods**.

Resolution: Fixed.

6.1.2 TreasuryAssetFacet.sol - no slippage protection on auctions

Severity: *Medium risk*

Context: Function: `_liquidateAsset()` in `TreasuryAssetFacet` Pattern: Uses DutchAuction._startAuction with static pricing parameters, no lower bound enforcement.

Description: The `_liquidateAsset` function initiates a Dutch auction using static `startMultiplierBps` and `endMultiplierBps` values, without enforcing a minimum sale price tied to real-time market conditions. While the Dutch auction mechanism gradually lowers the price over time to incentivize buyers, in extreme market conditions or low-liquidity periods, this may result in assets being sold significantly below their fair market value.

This becomes particularly critical during liquidation events, where distressed asset sales already carry risk of value loss. Without a price floor, arbitrageurs may exploit stale or misaligned price curves—especially if the asset's oracle price has diverged from the liquidation curve baseline.

Although the team notes that Nuon's listed assets are expected to be stable or safe, the protocol does not explicitly prevent a future governance-approved asset from exhibiting high volatility. The current setup lacks any guardrails to enforce a minimum price based on recent oracle data or deviation tolerance, exposing the Treasury to avoidable fire-sale losses.

Recommendation: Even if current asset risk is low, we recommend implementing optional slippage protections that can be toggled or configured per asset (a dynamic minimum price enforcement during liquidation).

Resolution: Fixed.

6.2 Low risk

6.2.1 Incompatibility with fee-on-transfer and rebasing tokens

Severity: *Low risk*

Context: All auction-related functions in `DutchAuction` rely on standard `IERC20.safeTransfer` and `safeTransferFrom` calls to compute exact transferred amounts. These functions assume 1:1 token movement with no intermediate logic altering the amount (i.e., standard ERC20 compliance).

Description: The current Dutch auction implementation does not account for tokens with non-standard behaviors, such as:

1. **Fee-on-transfer tokens** – These deduct a fee during transfers, leading to lower-than-expected amounts received or sent. The Dutch auction logic does not verify how much was actually transferred, potentially causing accounting mismatches and mispriced trades.
2. **Rebasing tokens** – These can unpredictably alter balances due to internal logic unrelated to transfers, which can break assumptions made in auction accounting (e.g., how much is available or remaining in an auction).

Without explicit checks or restrictions, usage of such tokens could result in incorrect pricing calculations.

Recommendation: You can either:

1. Explicitly document that you do not support tokens with a fee-on-transfer mechanism.
2. Check the `balance before and after the transfer` and validate it is the same as the amount argument provided.

Resolution: Fixed.

6.2.2 Inefficient Loop in `getFirstEndWeekId()`

Severity: Info / Low

Context: The `VoteEscrow.getFirstEndWeekId()` function iterates over `_userToLockIds` to compute the earliest ending lock. However, `_userToLockIds` is not pruned on `unlock()`, which causes stale lock IDs to persist in storage.

Description: Leaving expired or unlocked IDs in `_userToLockIds` leads to unnecessary iterations and gas inefficiencies. While tracking `firstEndWeekId` separately in `VotePowerInfo` was considered as a potential optimization, it was found that such tracking only shifts the burden from `allocate()` to `unlock()` since the list must be re-evaluated after every lock removal.

Despite this, the function could still benefit from:

- On-unlock pruning of stale IDs (where feasible)

- Optimization in iteration logic (as identified by the dev team)

This results in cleaner state and reduced computational overhead.

Recommendation: Continue with the small iteration optimization identified. If full tracking of `firstEndWeekId` proves too complex, at least explore pruning stale lock IDs during `unlock()` where safe and practical.

Resolution: Acknowledged

6.2.3 Unclear Handling of `expirationEpoch` in `allocate()`

Severity: Low

Context: In the `VoteAllocationManager.allocate()` function, the `expirationEpoch` field is hard-coded to `0` during allocation. Its value is updated externally, and not used as the main condition for expiry detection.

Description: This design may confuse reviewers or future contributors since a 0 value may be interpreted as unset or uninitialized. While the devs clarified that `endWeekId` is the authoritative source for determining expiration, this isn't immediately clear in the code.

The `expirationEpoch` field appears to be either legacy or secondary, tied to the rebalancing mechanism. Without inline context, its usage could be misinterpreted or misused.

Recommendation: Document clearly (in code comments) that `expirationEpoch` is not the primary condition for detecting allocation expiration. Note that `endWeekId` is the canonical source, and that epoch mutation is handled elsewhere, such as through rebalancing logic.

Resolution: Acknowledged Design is intentional. Recommend minor doc improvements for clarity.

6.2.4 Precision Loss in Long-Duration Dutch Auctions

Severity: *Low risk*

Context: The Dutch auction pricing decay mechanism in `DutchAuction.sol#_getAuctionPriceMultiplierBps()` linearly interpolates the multiplier between `startMultiplierBps` and `endMultiplierBps` based on elapsed time. It uses Solidity integer division, which introduces rounding behavior when the decay rate per second is fractional.

Description: In long-running auctions (e.g., 30 days = 2,592,000 seconds), if the difference between `startMultiplierBps` and `endMultiplierBps` is relatively small (e.g., 1,000 BPS), the per-second decay becomes a very small fraction:

- Per-second decay: ~0.0003858 BPS/second.
- Due to integer math, the calculated decay remains `0` for the first 43 minutes.
- Result: the multiplier does not change until full BPS step is accumulated, causing visible *step-wise* price drops.

This leads to:

- **Arbitrage opportunities**, as bots can snipe right after a price drop.
- **Jumpy auction curve**, reducing pricing smoothness.

Recommendation: No change is strictly required if auctions remain short-term. However, for future-proofing or if auction durations increase, consider:

- Using fixed-point libraries like `ABDKMath64x64` or `FullMath` for higher-precision interpolation.
- Introducing a non-linear decay curve (e.g., exponential) to reduce step effects in the early stage.

Resolution: Acknowledged.

6.2.5 NuonStaking.2 stake() checks if the referrer is the DEFAULT_REFERRER, but it's later assigned anyway if invalid

Severity: *Low risk*

Description: The function reverts if the user sets DEFAULT_REFERRER explicitly, but silently assigns it anyway if the referrer is ineligible.

Recommendation: Consider allowing DEFAULT_REFERRER directly or enforcing stricter checks.

Resolution: Acknowledged.

6.2.6 Second unlock resets the unlockTimestamp, delaying the first unlock too

Severity: *Low risk*

Context: NuonStaking.reclaimTokens() **Description:** When a user initiates a second unlock, it overrides the previous unlockTimestamp, delaying any already partially unlocked tokens. A more robust solution could allow tracking multiple unlocks or partial claims.

```
if (block.timestamp < unlockTime) {
    revert NuonStakingTokensStillLocked(block.timestamp, unlockTime);
}
```

Resolution: Acknowledged.

6.2.7 Last user to claim rewards may receive less due to compound rounding

Severity: *Low risk*

Context: NuonStaking._claimReward() **Description:** Adjusting rewards downward silently may unfairly penalize the last stakers due to rounding or calculation discrepancies. Consider minting any leftover owed rewards to ensure fair distribution or fail loudly.

```
if (reward > rewardInfo.rewardPool) {
    reward = rewardInfo.rewardPool;
}
```

Resolution: Acknowledged.

6.2.8 NuonStaking._rewardPerToken() may return wrong result and missled users

Severity: *Low risk*

Context: NuonStaking._rewardPerToken() **Description:** If totalStaked == 0, the function returns a cached value, which may no longer be accurate. This could mislead interfaces or users querying real-time earnings. You might consider returning 0 or marking stale data explicitly.

```
if (totalStaked == 0) {
    return rewardInfo.rewardPerTokenStored; // @audit stale data
}
```

Resolution: Acknowledged.

6.2.9 ChainlinkAutomation._checkAssets casting not needed

Severity: *Low risk*

Context: ChainlinkAutomation._checkAssets **Description:** All involved variables (previousPrice, currentPrice, MAX_BPS) are already uint256. The explicit casts are redundant and reduce readability without providing any additional safety.

```
priceDropBps = ((uint256(previousPrice) - uint256(currentPrice)) * MAX_BPS) /
    uint256(previousPrice); //@audit casting not needed, all are in uint256
```

Resolution: Fixed.

6.3 Informational risk

6.3.1 Overbidding risk - no post-trade slippage protection

Severity: *Informational risk*

Context: Function: `_buyAuction()` in `TreasuryAuctionsFacet` Interaction: Relies on DutchAuction library to enforce `maxAuctionPrice`

Description: While the `maxAuctionPrice` is passed to the `DutchAuction._buyAuction()` function to prevent overpayment, there is no post-trade validation in the calling contract to confirm that the final execution price adhered to this limit. This could raise concerns in a system audit or when analyzing safety guarantees locally within `TreasuryAuctionsFacet`.

However, after reviewing the implementation of `DutchAuction._buyAuction()`, it has been confirmed that the check is enforced internally before any token transfers occur. Thus, the system ensures price protection is applied at the auction logic layer.

Still, developers and auditors relying solely on the surface logic of `TreasuryAuctionsFacet` might overlook this protection.

Recommendation: Document explicitly at the call site that `maxAuctionPrice` enforcement is handled within the DutchAuction logic. This improves clarity and maintains future auditability.

Resolution: Fixed.

6.3.2 DEFAULT_MIN_REFERRAL_STAKE & DEFAULT_MIN_INITIAL_STAKE values are missleading

Severity: *Informational risk*

Context: NuonStaking.sol

Description: Using scientific notation like 1e18 makes it clearer that this is referring to a token with 18 decimals. It improves readability and consistency with typical ERC20 logic.

```
uint256 public constant DEFAULT_MIN_REFERRAL_STAKE = 100 ether; // @audit use e18
    for better readability
uint256 public constant DEFAULT_MIN_INITIAL_STAKE = 10 ether;   // @audit use e18
    for better readability
```

Resolution: Fixed.

6.3.3 Partial Cancellation Handling in `cancelVotePower()`

Severity: *Informational risk*

Context: The `VoteEscrow.cancelVotePower()` function currently reverts if the target user's available voting power is less than the specified `amount` to cancel.

Description: Reverting on insufficient voting power may introduce unnecessary friction and UX overhead. A more flexible approach would be to partially cancel what's available:

```
if (userVotingPower < amount) {
    amount = userVotingPower; // proceed with partial cancel
}
```

This allows cancel operations to succeed even when full cancellation isn't possible due to prior deductions or reallocations. The trade-off is behavioral complexity versus user-friendliness.

Partial cancellations may be preferable in cases like:

- Batched governance actions.
- Dynamic voting allocations.
- Situations where full power might not be available due to slashing or reallocation.

Recommendation: If protocol logic allows and UX favors it, implement partial cancellations by reducing the `amount` to `userVotingPower` instead of reverting. Otherwise, document the intended full-only behavior and make that assumption clear for integrators.

Resolution: Acknowledged.

6.3.4 Redundant `user` Parameter in `unlock()`

Severity: *Informational risk*

Context: The `VoteEscrow.unlock()` function accepts both `lockId` and `user`, but the actual ownership is already stored in `lockInfo`.

Description: The function reverts if `lockInfo.owner != user`, but since the caller passes both `lockId` and `user`, this check offers minimal real protection and can be bypassed with matching

values. The cleaner design is to remove the `user` parameter entirely and enforce `msg.sender == lockInfo.owner`.

This:

```
if (lockInfo.owner != user) revert();
```

Can be safely replaced with:

```
if (lockInfo.owner != msg.sender) revert();
```

This change:

- Prevents external manipulation by explicitly tying unlock rights to the caller.
- Reduces function complexity and improves trust assumptions.

Recommendation: Remove the redundant `user` parameter and enforce ownership using `msg.sender`. This simplifies logic and enhances security clarity.

Resolution: Acknowledged.