decurity

Smart Contract Security Audit Report

Nuon

1. Contents

2. General Information

This report contains information about the results of the security audit of the Nuon (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 09/12/2024 to 27/12/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: nuon-v2 (commit 1fec33). Re-testing was done for the commit dc61ae.

The following contracts have been tested:

- src/interfaces/IVoteAllocationManager.sol
- src/interfaces/IShutdownManager.sol
- src/interfaces/ITreasuryStakingFacet.sol
- src/interfaces/IDiamond.sol
- src/interfaces/IERC20RebaseableUpgradeable.sol
- src/interfaces/INuMint.sol
- src/interfaces/IWNuon.sol
- src/interfaces/IGovernance.sol
- src/interfaces/IRedistributor.sol
- src/interfaces/IChainlinkAutomation.sol
- src/interfaces/INuon.sol
- src/interfaces/IVoteEscrow.sol

- src/interfaces/IDutchAuction.sol
- src/interfaces/ILendingPool.sol
- src/interfaces/ITreasuryBufferFacet.sol
- src/interfaces/IOracleRegistry.sol
- src/interfaces/INuonStaking.sol
- src/interfaces/IBackstop.sol
- src/interfaces/IInflationOracle.sol
- src/interfaces/ITreasuryStorageFacet.sol
- src/interfaces/ITreasuryAssetFacet.sol
- src/interfaces/ITreasury.sol
- src/tokens/ERC20RebaseableUpgradeable.sol
- src/tokens/WNuon.sol
- src/tokens/NuMint.sol
- src/tokens/Nuon.sol
- src/treasury/TreasuryBufferFacet.sol
- src/treasury/TreasuryStakingFacet.sol
- src/treasury/TreasuryAssetFacet.sol
- src/treasury/common/TreasuryCommonAccessControl.sol
- src/treasury/common/TreasuryCommonPausable.sol
- src/treasury/common/TreasuryCommon.sol
- src/treasury/TreasuryDiamondCutFacet.sol
- src/treasury/TreasuryStorageFacet.sol
- src/Backstop.sol
- src/ShutdownManager.sol
- src/Redistributor.sol
- src/DutchAuction.sol
- src/Diamond.sol
- src/OracleRegistry.sol
- src/VoteAllocationManager.sol

- src/Governance.sol
- src/VoteEscrow.sol
- src/ChainlinkAutomation.sol
- src/NuonStaking.sol
- src/InflationOracle.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). Centralization risks were not a primary focus of this assessment, per the defined scope.

The main possible threat actors are:

- Chainlink Automation
- Aave pool
- Protocol Owner
- User

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

During audit we have detected multiple critical and high issues. Due to the number of identified issues and multiple rounds of retesting, we recommend an additional reaudit to ensure all fixes are properly addressed.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of February 19, 2025.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
User can make protocol insolvent via abusing `performUpkeep` function	src/ChainlinkAutomation.sol	**Critical**	Fixed
Steal user rewards by unlocking his stake	src/VoteEscrow.sol	**Critical**	Fixed
Default referrer change can lead to the DoS and loss of rewards for legitimate referrers	src/NuonStaking.sol	**High**	Fixed
Inflation attack on NuonStaking	src/NuonStaking.sol	**High**	Fixed
Double subtracting is possible in VoteAllocationManager	src/VoteAllocationManager.sol	**High**	Fixed
Incorrect mainValue calculation	src/treasury/TreasuryAssetFacet.sol	**High**	Fixed
Fee is not percent based	src/treasury/TreasuryStakingFacet.sol	**High**	Fixed

Issue	Contract	Risk Level	Status
Double subtracting is possible in VoteEscrow	src/VoteEscrow.sol	High	Fixed
Slashing can be avoided	src/NuonStaking.sol	High	Fixed
Expired allocations are not revoked	src/VoteAllocationManager.sol	High	Fixed
User's locks overslashing	src/VoteEscrow.sol	High	Fixed
CCIP gas limit isn't configurable in the Nuon contract	src/tokens/Nuon.sol	Medium	Fixed
All rewards can be stolen from VoteEscrow in the shutdown mode	src/VoteEscrow.sol	Medium	Fixed
Many assets processing may exceed Chainlink Automation max gas limit	src/ChainlinkAutomation.sol	Medium	Fixed
Incorrect storage location in Redistributor	src/Redistributor.sol	Medium	Fixed
DOS of `lockids`	src/VoteEscrow.sol	Medium	Fixed
Array flooding and incorrect accounting in Redistributor	src/Redistributor.sol	Medium	Fixed
Chainlink stale data	src/OracleRegistry.sol	Medium	Fixed
Unlocked shares account for global calculation	src/NuonStaking.sol	Medium	Fixed
Possible DOS in _buyAuction	src/DutchAuction.sol	Medium	Fixed
Shutdown mode can never be activated	src/TreasuryBufferFacet.sol	Medium	Fixed

Issue	Contract	Risk Level	Status
proposalBounty amount can be locked	srr/Governance.sol	Medium	Fixed
Asset oracle is not updatable	src/treasury/TreasuryAssetFacet.sol	Low	Acknowledged
Hardcoded USDC decimals in the Backstop contract	src/Backstop.sol	Low	Fixed
Improper referrer assignment	src/NuonStaking.sol	Low	Acknowledged
Inconsistent number of veto votes	src/Governance.sol	Low	Acknowledged
No mechanism to resolve failed CCIP messages in the Nuon contract	src/token/Nuon.sol	Low	Acknowledged
Nuon staking uses token balance instead of shares	src/NuonStaking.sol	Low	Acknowledged
Owner of contracts is EOA		Low	Acknowledged
Chainlink extraArgs is immutable	src/Nuon.sol	Low	Acknowledged
Unrestricted Inflation Value Change	src/tokens/Nuon.sol	Low	Acknowledged
_disableInitializers is missing in the `Redistributor`	src/Redistributor.sol	Low	Fixed
Oracle will return stale price	src/InflationOracle.sol	Low	Acknowledged
Public initialize functions	scripts/NuonSetup.sol	Low	Acknowledged

Issue	**Contract**	**Risk Level**	**Status**
Oracle will return the wrong price for asset if underlying aggregator hits `minAnswer`	src/OracleRegistry.sol	Low	Acknowledged
Lack of slashing recover logic		Low	Acknowledged
Lack of auction ending logic		Low	Fixed
DutchAuction to library	src/DutchAuction.sol	Low	Fixed
Token distribution may run out of gas	src/Redistributor.sol	Low	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. User can make protocol insolvent via abusing `performUpkeep` function

Risk Level: **Critical**

Status: Fixed in pull 292

Contracts:

src/ChainlinkAutomation.sol

Location: Function: `performUpkeep`.

Description:

Basic concepts

- Auctions during rebalancing are performed with a loss for a protocol with a certain amount, since they have a discount and takers won't fill auctions without their own profit.
- Rebalancing is done once buffer exceeds threshold
- Buffer is increased when users deposits to treasury contract
- Rebalancing is managed by ChainLink automation via `performUpkeep` with no access control.

Potential attack vector

- User deposits large amount of funds to treasury, thus increasing the buffer.
- User invokes `performUpkeep` and triggers asset rebalance.
- Auctions for assets are open now
- User withdraws from the protocol

Note that this can be done in one transaction. After that auctions will be filled and some funds from already existing buffer will be lost, because of discount in auction.

After that rebalance can be done again, but this time without depositing, because buffer will be lower than the required buffer, due to previously filled auctions. This again results in a certain loss for a protocol.

So basically: deposit -> upkeep -> auction-> withdraw -> auctions are filled -> upkeep -> auction -> auctions are filled -> repeat.

Malicious user will be able to gain profit via filling newly created auctions and buffer will be slowly decreasing due to a certain loss of value at auctions, at some point bringing protocol into insolvency state.

Remediation:

Consider limiting access to `performUpkeep` function and not allowing user to manipulate rebalance actions via deposits. This can be done via introducing % based fee when depositing.

5.2. Steal user rewards by unlocking his stake

Risk Level: **Critical**

Status: Fixed in pull 276

Contracts:

src/VoteEscrow.sol

Location: Lines: `340`. Function: `unlock()`.

Description:

The `unlock()` function in `VoteEscrow` sends rewards to `msg.sender` instead of the actual lock owner. Since `unlock()` can be called by anyone, this allows malicious actors to frontrun legitimate unlock transactions and steal accumulated rewards.

```
// Distribute lock rewards
$._voteAllocationManager.payUserRewardsPerLock(
    msg.sender, lockInfo.startEpoch, lockInfo.endWeekId, totalVotePowerOfLock
);

// Unlock the nuMINT tokens
$._nuMintToken.safeTransfer(user, lockInfo.amount);
```

Remediation:

Consider changing `msg.sender` to `user` to prevent rewards loss.

5.3. Default referrer change can lead to the DoS and loss of rewards for legitimate referrers

Risk Level: High

Status: Fixed in commit 8ea063

Contracts:

src/NuonStaking.sol

Location: Lines: 193. Function: `setDefaultReferrer`, `reclaimTokens`.

Description:

Initially, users without referrers are assigned to a default referrer address which is excluded from reward calculations. However, when this address is changed through the `setDefaultReferrer` function, users remain linked to the old default referrer address which now becomes eligible for rewards as a regular address. This unexpectedly dilutes the referral reward pool as the old default referrer starts competing for rewards, reducing the share of legitimate referrers.

In plus, the `reclaimTokens` function is vulnerable to a DoS because it contains a subtraction of the `unlockingShares` from the `referrerShares[referrer]`, which wasn't initialized during the stake and is equal to 0.

```
    // Update referrer shares
    if (referrer != $._defaultReferrer) {
        $._referrerRewardInfo.totalShares -= stakeInfo.unlockingShares;
        $._referrerShares[referrer] -= stakeInfo.unlockingShares;
    }
```

Test:

```
function testReclaimTokensWithDefaultReferrerChange() public {
    uint256 stakeAmount = 100 ether;
    uint256 unlockShares = 50 ether;
    address newDefaultReferrer = address(0x123);

    // Transfer NUON tokens to USER1
    vm.prank(ADMIN);
    nuonToken.transfer(USER1, stakeAmount);

    // USER1 stakes tokens with default referrer
    vm.startPrank(USER1);
    nuonToken.approve(address(nuonStaking), stakeAmount);
    nuonStaking.stake(stakeAmount, ZERO_ADDRESS); // Uses default referrer

    // Initiate unlock
    nuonStaking.initiateUnlock(unlockShares);
    vm.stopPrank();

    // Change default referrer
    vm.prank(GOVERNOR);
    nuonStaking.setDefaultReferrer(newDefaultReferrer);

    // Fast-forward time beyond the unlock period
    vm.warp(block.timestamp + unlockPeriod + 1);

    // Try to reclaim tokens - should revert due to underflow
    vm.startPrank(USER1);
    vm.expectRevert(); // Will revert due to arithmetic underflow
    nuonStaking.reclaimTokens();
    vm.stopPrank();
}
```

Remediation:

Consider fixing the logic related to default referrer.

5.4. Inflation attack on NuonStaking

Risk Level: High

Status: Fixed in pull 291

Contracts:

src/NuonStaking.sol

Location: Lines: 238-240.

Description:

The NounStaking contract is vulnerable to an inflation attack. An attacker can make a minimal initial deposit (e.g., 1 wei) and then directly transfer tokens to the contract to manipulate share values. This allows them to frontrun other users' deposits and steal their funds.

Proof of concept:

```
 function testFirstDepositInflationAttack() public {
     // Initial setup
     uint256 attackerInitialStake = 1 wei;
     uint256 attackerSecondAmount = 10 wei;
     uint256 victimAmount = 10 wei;
```

```
      // Transfer NUON tokens to attacker (USER1) and victim (USER2)
      vm.startPrank(ADMIN);
      nuonToken.transfer(USER1, attackerInitialStake + attackerSecondAmount);
      nuonToken.transfer(USER2, victimAmount);
      console.log("Attacker initial balance:", nuonToken.balanceOf(USER1));
      console.log("Victim initial balance:", nuonToken.balanceOf(USER2));
      console.log("\n");
      vm.stopPrank();

      // Attacker stakes minimal amount first
      vm.startPrank(USER1);
      nuonToken.approve(address(nuonStaking), attackerInitialStake);
      nuonStaking.stake(attackerInitialStake, ZERO_ADDRESS);
      console.log("Attacker stake:", attackerInitialStake);
      vm.stopPrank();

      // Attacker front-runs victim's transaction by transferring more tokens
directly to the contract
      vm.prank(USER1);
      nuonToken.transfer(address(nuonStaking), attackerSecondAmount);
      console.log("Attacker direct transfer to contract:",
attackerSecondAmount);

      // Victim stakes their tokens
      vm.startPrank(USER2);
      nuonToken.approve(address(nuonStaking), victimAmount);
      nuonStaking.stake(victimAmount, ZERO_ADDRESS);
      console.log("Victim stake:", victimAmount);
      vm.stopPrank();

      // Calculate shares
      uint256 attackerShares = nuonStaking.getStake(USER1).shareBalance;
      uint256 victimShares = nuonStaking.getStake(USER2).shareBalance;

      // Verify actual token balances vs shares
      uint256 attackerBalance = nuonStaking.stakedBalance(USER1);
      uint256 victimBalance = nuonStaking.stakedBalance(USER2);

      // Log the results
      console.log("\n");
      console.log("Results:");
      console.log("Attacker shares:", attackerShares);
      console.log("Attacker staked balance:", attackerBalance);
      console.log("Victim shares:", victimShares);
      console.log("Victim staked balance:", victimBalance);

  }
```

Remediation:

Consider minting dead shares at the beginning and checking `sharesToMint != 0` to avoid this attack vector.

References:

- https://mixbytes.io/blog/overview-of-the-inflation-attack
- https://medium.com/@vinicaboy/vault-inflation-attack-a-well-known-yet-persistent-threat-part-one-deecd79c9267

5.5. Double subtracting is possible in VoteAllocationManager

Risk Level: **High**

Status: Fixed in pull 283.

Contracts:

src/VoteAllocationManager.sol

Description:

The `VoteAllocationManager` contract has a potential issue where `_totalVotePowerAllocated` can be deducted twice under certain conditions. This occurs when there is a mismatch between the epoch duration and the week duration (`expirationWeekId`).

The issue arises because the allocation expiration logic depends on the `expirationWeekId`, which is determined as:

```
expirationWeekId = currentWeekId + $._maxAllocationDuration;
if (expirationWeekId >= firstEndWeekId) expirationWeekId = firstEndWeekId;
```

If the epoch lasts longer than a week, the following sequence can lead to double deduction:

1. A stale week is cleared using `processTotalVotePowerAllocatedUpdates`.
2. A new allocation is created in the same epoch, triggering `_revokeAllocation`.
3. The `_revokeAllocation` function deducts vote power again for the same week, as it does not check whether the week has already been cleared.

This results in `_totalVotePowerAllocated` being deducted twice for the same week, leading to potential underflow and incorrect accounting.

Remediation:

Introduce checks to ensure that `processTotalVotePowerAllocatedUpdates` and `_revokeAllocation` do not deduct vote power for the same week more than once.

5.6. Incorrect mainValue calculation

Risk Level: **High**

Status: Fixed in pull 289

Contracts:

src/treasury/TreasuryAssetFacet.sol

Description:

The protocol miscalculates the `currentAssetValue` due to an incorrect handling of decimal normalization between the asset price, oracle decimals, and stablecoin decimals.

The `assetPrice` returned from the `_oracleRegistry.getAssetPrice` function is always normalized to 18 decimals. However, the `priceScalingFactor` is computed as follows:

```
assetInfo.priceScalingFactor =
    _getAssetDecimals(asset) + _getOracleDecimals(asset) -
_getAssetDecimals($._depositStablecoin);
```

This results in an inconsistency during the `currentAssetValue` calculation:

```
currentAssetValue = (currentAssetBalance * assetPrice) / (10 **
yieldBearingAsset.priceScalingFactor);
```

For example, if:

- The asset has **6 decimals**,
- The oracle has **8 decimals**, and
- The stablecoin has **6 decimals**,

The calculated `priceScalingFactor` will equal **8 decimals**. When calculating `currentAssetValue`, the multiplication of a **6-decimal asset balance** with an **18-decimal asset price**, divided by an **8-decimal scaling factor**, results in a final value normalized to **16 decimals** instead of the expected 18 decimals. This discrepancy can lead to incorrect asset valuations and downstream issues in protocol accounting.

Remediation:

Ensure each value is normalized before calculations.

5.7. Fee is not percent based

Risk Level: **High**

Status: Fixed in pull 292

Contracts:

src/treasury/TreasuryStakingFacet.sol

Description:

The current implementation of the deposit fee in the protocol is a fixed value (`_mintingFee`) rather than a percentage-based fee. This design introduces a vulnerability that can be exploited through rebasing attacks, where the attacker can manipulate token supply and profits as follows:

1. The attacker deposits a large amount before a rebase event occurs.
2. The token undergoes a rebase (adjusting its supply).
3. The attacker withdraws their deposit with profits, bypassing the intended fee mechanism.

```
        // Exchange depositStablecoin for aToken
        _depositTokenForAToken($, amount);

        uint256 depositAmount;
        // Only mint if receiver is not address(0)
        if (receiver != address(0)) {
            if (_isShutdown($)) {
                revert TreasuryDepositingDisabledInShutdown();
            }
            // Apply minting fee
>           depositAmount = amount - $._mintingFee;
            // Adjust deposit amount based on decimals difference and mint
Nuon tokens to receiver
            uint256 adjustedDepositAmount = _adjustAmount(depositAmount,
$._decimalsDifference);
            $._nuonToken.mint(receiver, adjustedDepositAmount);
        } else {
            depositAmount = amount;
        }
```

Remediation:

The deposit fee should be calculated as a percentage of the deposit amount to prevent rebasing attacks.

5.8. Double subtracting is possible in VoteEscrow

Risk Level: High

Status: Fixed in pull 283

Contracts:

src/VoteEscrow.sol

Description:

The `VoteEscrow` contract uses two layers of accounting for vote power: **user-specific vote power** and **global vote power**.

When a user clears old vote power using the `unlock` function, the protocol should invoke `processTotalVotePowerUpdates` to adjust the global vote power by deducting the aggregated values from the `_totalVotePowerToDeduct` mapping. However, there is an issue where `_totalVotePower` can be deducted twice under certain conditions.

Specifically, during slashing operations, the protocol deducts vote power from both user-specific vote power and global vote power. If outdated user vote power values have not been cleared prior to slashing, the global vote power reduction can result in a double subtraction for the same expired lock. This can lead to incorrect global vote power calculations and, in edge cases, an underflow error.

```
function slashUser(address user, uint48 epoch, uint256 allocationId, uint256
assetIndex)
        external
        onlyRole(SLASHER_ROLE) {
                ...
                uint256[] storage lockIds = $._userToLockIds[user];
        uint256 lockIdsLength = lockIds.length;
        uint256 totalAmountSlashed;
        uint256 totalVotePowerSlashed;
        for (uint256 i; i < lockIdsLength; ++i) {
            LockInfo storage lockInfo = $._lockIdToLockInfo[lockIds[i]];

            unchecked {
                uint256 amountSlashed = (lockInfo.amount * lossBps) / MAX_BPS;
                uint256 votePowerSlashed = (amountSlashed *
lockInfo.votePowerMultiplierBps) / MAX_BPS;
```

```
                // Update the locked amount
                lockInfo.amount -= amountSlashed;
                // Update the vote power reductions
                votePowerInfo.votePowerToDeduct[lockInfo.endWeekId] -=
votePowerSlashed;
                if (_lastProcessedWeekId <= lockInfo.endWeekId) {
                    $._totalVotePowerToDeduct[lockInfo.endWeekId] -=
votePowerSlashed;
                }
                // Update accumulators
                totalAmountSlashed += amountSlashed;
                totalVotePowerSlashed += votePowerSlashed;
            }

            // Remove week from tracking if no more power to deduct for the
lock for this week
            if (votePowerInfo.votePowerToDeduct[lockInfo.endWeekId] == 0) {
                _removeWeekIdFromDeductionList(votePowerInfo,
lockInfo.endWeekId);
            }
        }

        // Check if we actually are able to slash anything
        if (totalAmountSlashed == 0) revert VoteEscrowNoLocksToSlash();

        // Reduce the user's total vote power
        votePowerInfo.totalVotePower -= totalVotePowerSlashed;
        // Reduce the total vote power
        $._totalVotePower -= totalVotePowerSlashed;
                ...
}
```

Remediation:

Introduce a check during the slashing process to verify that locks being processed have not expired. This will ensure that `_totalVotePowerToDeduct` is not deducted for outdated locks, preventing double subtraction and potential underflows.

5.9. Slashing can be avoided

Risk Level: High

Status: Fixed in pull 315

Contracts:

src/NuonStaking.sol

Description:

The vulnerability lies in the fact that anyone can create a lock for another user using the `lockFor` function with a minimal amount (`amount = 1 wei`). This introduces several issues related to denial-of-service (DoS) attacks on slashing mechanisms.

Denial of Service for Slashing:

- The `slashUser` function iterates over all of a user's locks to apply penalties. Spamming locks with minimal amounts makes it computationally expensive or impractical for the administrator to perform slashing operations, effectively preventing penalties from being applied.

Attack Vector:

- An attacker calls the `lockFor` function multiple times with minimal amounts (`1 wei`) on behalf of a target user (`receiver`).
- The attacker creates a large number of locks, resulting in:
  - An inflated `lockIds` array for the target user.
  - Increased computational overhead in `slashUser` function.

```
function lockFor(address receiver, uint256 amount, uint48 lockPeriod) external
whenNotPaused returns (uint256) {
        return _lockFor(receiver, amount, lockPeriod);
    }


function _lockFor(address receiver, uint256 amount, uint48 lockPeriod)
internal returns (uint256) {
        ...


        // Add lock ID to user locks
        $._userToLockIds[receiver].push(lockId);


        emit Locked(receiver, lockId);


        return lockId;
    }
```

```
function slashUser(address user, uint48 epoch, uint256 allocationId, uint256
assetIndex)
        external
        onlyRole(SLASHER_ROLE)
    {
        ...

        // Get total amount of veMint to slash
        uint256[] storage lockIds = $._userToLockIds[user];

        // @audit iteration through all locks

        uint256 lockIdsLength = lockIds.length;
        uint256 totalAmountSlashed;
        uint256 totalVotePowerSlashed;
        for (uint256 i; i < lockIdsLength; ++i) {
        ...
```

Remediation:

Consider adding a minimum amount of lock to prevent spam from being sent.

5.10. Expired allocations are not revoked

Risk Level: **High**

Status: Fixed in pull 280

Contracts:

src/VoteAllocationManager.sol

Description:

The vulnerability in the allocation mechanism stems from the improper handling of expired allocations when a new allocation is applied. Specifically, the contract does not update certain variables that track vote power and deductions when an allocation is marked as expired. This oversight results in an incorrect calculation of total distributed vote power, leading to cascading issues such as imbalanced funds, incorrect reward distribution, and inaccurate assessment of liquidated assets.

Detailed Problem:

1. **Allocation Expiry and Vote Power Deduction:**

- When an existing allocation is set to expire (`latestAllocation.expirationEpoch` and `latestAllocation.expirationWeekId` are updated), the contract fails to:
    - Update the asset-related deductions (`assetInfo.amountToDeduct[allocation.expirationWeekId]`) for the old `expirationWeekId`.
    - Update `$._totalVotePowerAllocatedToDeduct[allocation.expirationWeekId]` for the old `expirationWeekId`.

2. **Incorrect Total Allocated Vote Power:**
    - When the new allocation is applied, the contract increments `$._totalVotePowerAllocated` and `$._totalVotePowerAllocatedToDeduct[allocation.expirationWeekId]` for the new allocation without properly reducing these values for the expired allocation.
        - This results in an inflated total vote power, causing miscalculations during rebalancing and rewards distribution.
3. **Cascading Effects:**
    - The mismanagement of vote power deductions leads to:
        - Improper rebalancing of assets in `_rebalanceAssets`, as the targeted values are calculated based on incorrect total vote power.
        - Incorrect profit and reward distribution among users due to inaccurate tracking of allocated power.
        - Faulty evaluations of liquidated assets, which rely on the rebalanced state.

```
function allocate(address[] calldata assets, uint256[] calldata portionsBps)
external whenNotPaused {
        ...

        AllocationInfo[] storage allocations =
$._userToAllocationInfos[msg.sender];

        // Check if there is an existing allocation
        if (allocations.length != 0) {
            AllocationInfo storage latestAllocation =
allocations[allocations.length - 1];
            // Check if user has already applied allocation for the next epoch
            if (latestAllocation.allocationEpoch == $._globalEpochCounter + 1)
{
                // Remove the latest allocation information and revoke
allocated votes
                _revokeAllocation($, latestAllocation);
                allocations.pop();
            } else {
                // Set expiration epoch and week ID for the latest allocation
if it is active
                if (latestAllocation.expirationWeekId >= currentWeekId) {
                    latestAllocation.expirationEpoch = $._globalEpochCounter;
                    latestAllocation.expirationWeekId = currentWeekId;
                    // @audit-issue
assetInfo.amountToDeduct[allocation.expirationWeekId] and
$._totalVotePowerAllocatedToDeduct[allocation.expirationWeekId] is not updated
                }
            }
        }

        ...

        // Apply new allocation
        _applyAllocation($, allocations[allocations.length - 1],
currentWeekId);

        emit Allocated(msg.sender);
    }
```

Remediation:

Consider updating `assetInfo.amountToDeduct[allocation.expirationWeekId]` and `$._totalVotePowerAllocatedToDeduct[allocation.expirationWeekId]`.

5.11. User's locks overslashing

Risk Level: **High**

Status: Fixed in pull 283

Contracts:

src/VoteEscrow.sol

Location: Function: `slashUser()`.

Description:

The vulnerability in the `slashUser()` function arises from the way it applies slashing indiscriminately across all of a user's locks, even those that were created after the allocation was made or after the token in the allocation was liquidated. This introduces the following issues:

1. **Slashing Locks Created After the Allocation:**

   The function does not differentiate between locks that were active during the voting allocation and those created afterward. As a result:

   - Users may have locks created **after the allocation period**, which were never used for voting on the liquidated asset.
   - These locks are still subjected to slashing, which is unfair and inconsistent with the purpose of the punishment.

2. **Slashing Empty or Expired Locks:**

   The function also processes locks that are already **expired** or have an amount of zero. This leads to unnecessary computation and potentially incorrect behavior.

Problematic Aspects:

- **Indiscriminate Slashing:**

  The function loops through all locks associated with the user (`lockIds`), calculating the amount to slash and reducing the locked amount (`lockInfo.amount`) and voting power (`votePowerInfo.totalVotePower`). This logic does not check whether the lock was active during the allocation period.

- **Potential Over-Slashing:**

  Locks that were created after the allocation could have higher amounts due to new deposits. Applying slashing to these locks results in **over-penalizing users** for actions unrelated to the allocation.

- **Lack of Temporal Validation:**

  There is no validation to ensure that the locks being slashed existed and were active during the time the user voted on the now-liquidated allocation.

Impact:

- Users with multiple locks, including those unrelated to the allocation in question, face excessive penalties.

- Creates distrust in the system, as users are penalized unfairly for actions they did not take during the allocation period.

Remediation:

Consider adding a check to ensure that only locks active during the allocation period are subject to slashing and skip processing for locks that have already expired or have zero amount.

5.12. CCIP gas limit isn't configurable in the Nuon contract

Risk Level: Medium

Status: Fixed in pull 306

Contracts:

src/tokens/Nuon.sol

Location: Lines: 396. Function: `_buildCCIPMessage()`.

Description:

The gas limit for CCIP message is hardcoded in the `_buildCCIPMessage()` function of the Nuon contract. The gas limit should be configurable, since there may be changes in future CCIP and destination chains versions, which can affect the consumed gas.

Remediation:

Make the gas limit value configurable in the contract

References:

- https://docs.chain.link/ccip/best-practices#using-extraargs

5.13. All rewards can be stolen from VoteEscrow in the shutdown mode

Risk Level: Medium

Status: Fixed in pull 307

Contracts: src/VoteEscrow.sol

Location: Function: `_lockFor(), unlock()`.

Description:

In shutdown mode, the `unlock()` function bypasses the lock period check to allow immediate fund withdrawals. However, users can still lock tokens through the `lock()` and `lockFor()` functions since the internal `_lockFor()` function has no shutdown mode restrictions. This vulnerability allows users to repeatedly lock tokens for extended periods and unlock them immediately, enabling them to steal rewards from the contract.

Remediation:

Disable the `_lockFor()` function, when the protocol is in the shutdown mode.

5.14. Many assets processing may exceed Chainlink Automation max gas limit

Risk Level: Medium

Status: Fixed in pull 380

Contracts:

ChainlinkAutomation.sol

Location: Function: `performUpkeep()`.

Description:

According to the Chainlink Automation documentation the maximum gas limit that can be set on Arbitrum and Ethereum is 5_000_000.

The `performUpkeep()` function in the ChainlinkAutomation contract handles Treasury rebalancing and solvency restoration when needed. Since both operations involve extensive processing, the gas limit may be insufficient when rebalancing or restoring solvency with a large number of assets (e.g., 35-40 assets).

Remediation:

Calculate the maximum number of assets that can be processed within the given gas limit and restrict the number of assets that can be added to the protocol accordingly.

References:

- https://docs.chain.link/chainlink-automation/overview/supported-networks/#ethereum-1

- https://docs.chain.link/chainlink-automation/overview/supported-networks/#arbitrum-one

5.15. Incorrect storage location in Redistributor

Risk Level: Medium

Status: Fixed in commit b3250e

Contracts:

- src/Redistributor.sol

Location: Lines: 45.

Description:

The `Redistributor` contract has next storage layout:

```
/// @dev keccak256(abi.encode(uint256(keccak256("nuon.storage.Redistributor"))
- 1)) & ~bytes32(uint256(0xff))
bytes32 private constant REDISTRIBUTOR_STORAGE_LOCATION =
    0xf2f1cdd2928bdf6b3a3da7269b9cdaaa8ee556c7e04b070b1bba11f5e89af600;
```

However, actual hash from `keccak256(abi.encode(uint256(keccak256("nuon.storage.Redistributor")) - 1)) & ~bytes32(uint256(0xff))` is equal to `0x09860f4eda054c0ba8d6e0f27794d12d21bae82d34fa9316befc3fd1503eb900`.

Remediation:

Consider to fix the storage location to match the documentation and prevent collisions in future.

5.16. DOS of `lockid`s

Risk Level: Medium

Status: Fixed in pull 315

Contracts:

src/VoteEscrow.sol

Description:

A malicious user can spam the system by creating many small locks for other users, leading to a Denial of Service (DOS) attack. This prevents valid users from calling `allocate()` in the `VoteAllocationManager` because the system cannot process all the locks efficiently.

Remediation:

Consider limiting minimum amount for creating a lock.

5.17. Array flooding and incorrect accounting in Redistributor

Risk Level: Medium

Status: Fixed in commit ed2ec8

Contracts:

src/Redistributor.sol

Location: Lines: 95. Function: `updateRecipients`.

Description:

The `Redistributor` contract is vulnerable to DoS due to the lack of checks for duplicate recipient addresses in the `updateRecipients` function with zero `basisPoints`.

Test:

```
function testDoSWithDuplicateRecipients() external {
    vm.startPrank(GOVERNOR, GOVERNOR);

    // Initial setup with duplicate recipients
    address[] memory recipients = new address[](4);
    uint256[] memory basisPoints = new uint256[](4);

    recipients[0] = address(0x1111);
    recipients[1] = address(0x2222); // Duplicate recipient
    recipients[2] = address(0x2222);
    recipients[3] = address(0x2222);

    basisPoints[0] = 6000;
    basisPoints[1] = 0;
    basisPoints[2] = 0;
    basisPoints[3] = 4000;

    redistributor.updateRecipients(recipients, basisPoints);

    // Mint tokens to contract
    uint256 totalTokens = 1000 * 1e18;
    nuon.mint(address(redistributor), totalTokens);

    // Wait for distribution period
    vm.warp(block.timestamp + redistributor.distributionFrequency());

    // This should revert because the contract will try to distribute more
tokens than it has
    vm.expectRevert(
        abi.encodeWithSelector(
            IERC20Errors.ERC20InsufficientBalance.selector,
            address(redistributor),
            0,
            400 ether
        )
    );
    // Distribute tokens
    redistributor.distributeTokens();

    vm.stopPrank();
}
```

Remediation:

Consider declining duplicate recipients and base points equal to 0.

5.18. Chainlink stale data

Risk Level: Medium

Status: Fixed in commit c82a26ca

Contracts:

src/OracleRegistry.sol

Description:

The calls to `AggregatorV3Interface::latestRoundData` lack the necessary validation for Chainlink data feeds to ensure that the protocol does not ingest stale or incorrect pricing data that could indicate a faulty feed.

Remediation:

Consider adding checks for stale data. Please review my below code for example, and make changes that suit you.

```
(uint80 roundID, int256 answer, uint256 timestamp, uint256 updatedAt, ) =
registry.latestRoundData(
    token,
    USD
    );
require(updatedAt >= roundID, "Stale price");
require(timestamp != 0,"Round not complete");
require(answer > 0,"Chainlink answer reporting 0");
```

5.19. Unlocked shares account for global calculation

Risk Level: Medium

Status: Fixed in commit 9e0f8ed

Contracts:

NuonStaking.sol

Description:

The `NuonStaking.sol` contract has three key functions for interacting with shares:

1. `stake()`
2. `initiateUnlock()`
3. `reclaimTokens()`

Step 1: Staking

When Alice calls `stake()`, she deposits Nuon tokens and receives shares in return. This increases `_rewardInfo.totalShares`, meaning all future rewards are now distributed among a larger share pool. As a result, the rewards per share decrease.

Step 2: Unlocking

When Alice decides to withdraw her shares, she calls `initiateUnlock()`. At this point:

- The shares are deducted from her balance, so she no longer earns rewards on the shares she intends to withdraw.
- However, `_rewardInfo.totalShares` remains unchanged at this stage. This means that rewards continue to be allocated to Alice's shares even though she is no longer receiving them.

Step 3: Claiming Tokens

After the unlocking period, Alice calls `reclaimTokens()` to burn the shares she withdrew in step 2 and receive the corresponding Nuon tokens.

- Only at this point does `_rewardInfo.totalShares` decrease, restoring the reward rate per share to what it was before step 1.
- Since the shares were still included in `_rewardInfo.totalShares` during the unlock period, rewards were effectively allocated to Alice's burned shares, even though she was not benefiting from them.

Remediation:

To prevent rewards from being allocated to burned shares:

- `_rewardInfo.totalShares` should be decreased in `initiateUnlock()`, rather than in `reclaimTokens()`.
- However, this change requires additional modifications in the contract to ensure correct reward calculations and prevent unintended side effects.

5.20. Possible DOS in _buyAuction

Risk Level: Medium

Status: Fixed in pull 293

Contracts:

src/DutchAuction.sol

Description:

The `_buyAuction` function in the DutchAuction contract reverts when the requested purchase amount exceeds the remaining asset amount:

```
if (auctionedTokenAmount > auction.remainingAssetAmount) {
    revert AuctionNotEnoughAssetRemaining(auction.remainingAssetAmount);
}
```

A malicious user can intentionally buy a minimal amount (e.g., 1 token), leaving a small residual balance. This causes subsequent buyers who attempt to purchase the full or large portion of the auctioned tokens to encounter reverts, effectively resulting in a DoS.

Remediation:

Allow buyers to purchase the remaining asset amount if their requested amount exceeds the available supply.

5.21. Shutdown mode can never be activated

Risk Level: Medium

Status: Fixed in commit dc61aef.

Contracts:

src/TreasuryBufferFacet.sol

Description:

The vulnerability in the `attemptToRestoreSolvency` function lies in two key issues:

1. **Missing Verification for Slashing NUON Stakes:**
    - The function does not check whether the `nuonStakingBalance` is sufficient to cover the `insolvencyAmount`. If the slashed NUON stakes are insufficient, the function still returns prematurely without addressing the insolvency.

- **Insufficient Backstop Liquidation Check:**
    - When attempting to sell backstop assets (`backstopValuation > 0`), the function does not verify whether the `backstopValuation` is sufficient to cover the entire `insolvencyAmount`. If the backstop funds are inadequate, the function still exits early, failing to activate the required **shutdown mode**.

Detailed Problem Analysis:

1. **Logic Breakdown in the Function:**
    - The function follows three steps to resolve insolvency:
        1. Slash NUON stakes.
        2. Liquidate backstop assets.
        3. Activate shutdown mode if the above two options fail.
2. **Slashing NUON Stakes Issue:**
    - The function calculates the `slashAmount` based on the insolvency amount but does not validate whether the slashed amount can fully cover the `insolvencyAmount`. If it cannot, the function returns prematurely, leaving the protocol partially insolvent.
3. **Backstop Liquidation Issue:**

- The function triggers backstop asset liquidation without verifying whether the `backstopValuation` is sufficient to resolve the `insolvencyAmount`.
- If the valuation is insufficient, the system does not transition to shutdown mode, as the function exits after starting the liquidation process.

4. **Unresolved Insolvency:**
    - In cases where neither slashing stakes nor liquidating backstop assets fully resolves the insolvency, the protocol remains in an insolvent state, leading to financial instability and user losses.

```
function attemptToRestoreSolvency()
        external
        onlyRole(SOLVENCY_MANAGER_ROLE)
        returns (uint256 slashDifference, uint256[] memory backstopAuctionIds,
bool shutdownModeActivated)
    {
        TreasuryStorage storage $ = _getTreasuryStorage();
        (uint256 totalValue, uint256 totalNuonSupply, uint256
insolvencyAmount) = _calculateInsolvencyAmount($);

        // Ensure the protocol is insolvent
        if (insolvencyAmount == 0) {
            revert TreasuryProtocolIsSolvent();
        }

        // Option 1: Attempt to slash NUON stakes
        uint256 nuonStakingBalance =
$._nuonToken.balanceOf(address($._nuonStaking));
        if (nuonStakingBalance > 0) {
            uint256 backedNuonAmount =
                _adjustAmount((totalValue * MAX_BPS) /
$._insolvencyThresholdBps, $._decimalsDifference);
            uint256 slashAmount = totalNuonSupply - backedNuonAmount;
            slashDifference = $._nuonStaking.slash(slashAmount);
            return (slashDifference, backstopAuctionIds,
shutdownModeActivated);
        }

        // Option 2: Attempt to sell backstop assets
        uint256 backstopValuation = $._backstop.getBackstopValuation();
        if (backstopValuation > 0) {
            backstopAuctionIds =
$._backstop.assetsLiquidation(insolvencyAmount);
            return (slashDifference, backstopAuctionIds,
shutdownModeActivated);
        }
        // Option 3: Activate shutdown mode
        $._shutdownManager.activateShutdown();
        shutdownModeActivated = true;
        return (slashDifference, backstopAuctionIds, shutdownModeActivated);
    }
```

Remediation:

Consider not returning the function if the solvency in not restored.

5.22. proposalBounty amount can be locked

Risk Level: Medium

Status: Fixed in pull 365

Contracts:

src/Governance.sol

Description:

The vulnerability lies in the absence of checks during proposal creation to ensure that the proposed asset is either already supported by the treasury or a proposal to support it already exists. This can lead to multiple identical proposals for the same asset being created and approved, but only the first finalized proposal can successfully execute. Subsequent proposals will revert during finalization, resulting in the permanent locking of the bounty amounts contributed by the proposers.

Detailed Problem:

1. **Proposal Creation Without Asset Duplication Check:**
    - There is no mechanism to check if:
        - The asset is already supported by the treasury.
        - Another active proposal exists for supporting the same asset.

This allows multiple users to create proposals for the same asset.

5. **Finalization of Duplicate Proposals:**
    - When multiple proposals for the same asset pass the voting stage:
        - The first finalized proposal successfully calls `treasury.supportAsset(_proposal.asset)` and adds the asset to the treasury.

  - Subsequent proposals for the same asset fail at the `treasury.supportAsset` call because the asset is already supported. This reverts the transaction and prevents the proposal's bounty from being returned or burned.

6. **Permanent Locking of Bounty Amounts:**
   - Proposers of failed finalizations lose their bounty amounts, which remain locked in the contract as there is no mechanism to handle this scenario.

```
function finalizeProposal(uint256 proposalId) external {
    ...

    passed = _proposal.totalVeto < (getTotalVotingPower() *
$._minVetoPercentageBps) / MAX_BPS;
    if (passed) {
        // Transfer bounty to the proposer
        $._nuon.safeTransfer(_proposal.proposer, _proposal.bountyAmount);

        // Update the oracle registry with the new oracle for the asset
        $._registry.setOracle(_proposal.asset, _proposal.oracle);

        // Add the asset to the Treasury's supported assets
        ITreasury treasury =
$._voteEscrow.voteAllocationManager().nuonTreasury();
        treasury.supportAsset(_proposal.asset); // @audit-issue reverts if
asset are supported already
    } else {
        // Burn the bounty amount from the contract's NUON balance
        $._nuon.burn(address(this), _proposal.bountyAmount);
    }

    emit GovernanceProposalFinalized(proposalId, passed);
}
function supportAsset(IERC20 asset) external onlyRole(GOVERNANCE_ROLE) {
    if (address(asset) == address(0)) revert
ITreasury.TreasuryZeroAddressUnsupported();

    // Validate oracle by attempting to retrieve the asset's price
    _getAssetPrice(asset);

    TreasuryStorage storage $ = _getTreasuryStorage();

    // @audit reverts if asset are supported already
    if ($._supportedAssetsIndex[address(asset)] > 0) revert
TreasuryAssetAlreadySupported(asset);
```

```
    // Add ERC20 token
    _addAsset($, asset);
}
```

Remediation:

Consider checking if this asset is already supported and if there is an equally active proposal for this asset.

5.23. Asset oracle is not updatable

Risk Level: Low

Status: See customer response below

Customer Response: The supportAsset() function is for adding assets to the Treasury contract, not managing oracle addresses. If an oracle address needs to be updated for a supported asset, it should be done via the setOracle method in the OracleRegistry.

Contracts:

src/treasury/TreasuryAssetFacet.sol

Description:

The bug in the code snippet is that once an asset is marked as supported, it cannot be updated or re-added if the oracle changes or requires an update. This is because the check if `($._supportedAssetsIndex[address(asset)] > 0)` prevents re-adding an asset that is already in the supported assets list, even if the oracle address needs to be updated.

```
function supportAsset(IERC20 asset) external onlyRole(GOVERNANCE_ROLE) {
        if (address(asset) == address(0)) revert
ITreasury.TreasuryZeroAddressUnsupported();

        // Validate oracle by attempting to retrieve the asset's price
        _getAssetPrice(asset);

        TreasuryStorage storage $ = _getTreasuryStorage();

        if ($._supportedAssetsIndex[address(asset)] > 0) revert
TreasuryAssetAlreadySupported(asset);
        //@audit in case if pricefeed updates, set oracle address update will
not be possible

        // Add ERC20 token
        _addAsset($, asset);
    }
```

Remediation:

Make it possible to update an existed address of an oracle linked to the asset

5.24. Hardcoded USDC decimals in the Backstop contract

Risk Level: Low

Status: Fixed in pull 407

Contracts:

Backstop.sol

Location: Lines: `340, 437`.

Description:

In the Backstop contract, lines 340 and 437 use a hardcoded multiplier of 1e12, assuming USDC will always have 6 decimals. However, USDC can have different decimal places on various chains. Instead, the contract should use the `decimals()` function of the USDC contract to determine the correct decimal places.

Remediation:

Calculate multiplier based on the value of USDC `decimals()` function instead of hardcoded values

5.25. Improper referrer assignment

Risk Level: Low

Status: See customer response below

Customer Response: We want referrers to be active participants before promoting the protocol. This intended behavior ensures referral rewards are properly distributed and discourages misuse.

Contracts:

src/NuonStaking.sol

Description:

When users register with a provided `referrer`, the `_isEligibleReferrer` function checks if the referrer meets certain criteria, such as having a minimum stake. If the referrer fails to qualify (e.g., due to a protocol change increasing the minimum stake requirement), the protocol automatically assigns the `$._defaultReferrer`. This behavior can lead to:

1. Users unknowingly having their referrals redirected to the default referrer.
2. Potential loss of legitimate referral incentives for both the user and the referrer.

The issue arises because `_isEligibleReferrer` simply returns `false` instead of reverting when an invalid `referrer` is provided.

Remediation:

Update `_isEligibleReferrer` to **revert** if the user explicitly provides a referrer who does not meet the eligibility criteria.

5.26. Inconsistent number of veto votes

Risk Level: Low

Status: See customer response below

Customer Response: The ability to shift outcomes by acquiring and staking tokens is a strength, not a flaw. Restricting this would limit legitimate participation. If last-minute voting shifts become problematic, we will address it by extending the voting period if major vote shifts occur within 24 hours of a vote completing.

Contracts:

src/Governance.sol

Description:

The proposal finalization logic uses the current total voting power to determine veto success:

```
_proposal.totalVeto < (getTotalVotingPower() * $._minVetoPercentageBps) /
MAX_BPS;
```

This allows an attacker (or regular users accidentally) to inflate the total voting power at the last moment by locking tokens in `VoteEscrow`, potentially reducing the relative veto percentage and causing a proposal to pass even with sufficient veto votes.

Remediation:

Store the total voting power at the time of proposal creation and use the lesser of the stored voting power or the current total voting power during finalization.

5.27. No mechanism to resolve failed CCIP messages in the Nuon contract

Risk Level: Low

Status: See customer response below

Customer Response: Bridge messages should not fail, and in the rare case that a rebase message fails, it will be resolved in the next update without loss of funds.

Contracts:

src/token/Nuon.sol

Description:

The Nuon contract does not implement any refund/message resend mechanism to resolve failed CCIP messages, which may result in funds lost by users.

Remediation:

Consider implementing a way to resolve failed messages.

5.28. Nuon staking uses token balance instead of shares

Risk Level: Low

Status: See customer response below

Customer Response: This is an intended feature of the protocol. The staking contract is designed to accept direct Nuon transfers to increase the rewards pool, enabling redistribution of excess APY earned from the treasury to Nuon stakers.

Contracts:

src/NuonStaking.sol

Description:

The current implementation of `NuonStaking` uses the contract's token balance (`$._nuonToken.balanceOf(address(this))`) for accounting purposes instead of tracking shares properly. While `$._rewardInfo.totalShares` should ideally match `totalNuon` if users always interact through the `stake` function, direct token transfers to the contract (accidental or malicious) can disrupt the accounting. This results in:

- Incorrect calculation of `sharesToMint` in `stake`, leading to potential user losses.
- Misaligned reward distribution in `_rewardPerToken`, as incoming tokens are counted inappropriately.

This flawed design increases the risk of incorrect reward allocation and inconsistency in staking logic.

Remediation:

Replace the reliance on `balanceOf(address(this))` with strictly share-based calculations. Update the logic to account for staking and reward distribution using only `$._rewardInfo.totalShares` and internal tracking.

5.29. Owner of contracts is EOA

Risk Level: Low

Status: See customer response below

Customer Response: Immediately after deploying we will move to a multi-sig wallet

Description:

Since in the config for deploy this address was set as an initial owner - `0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266`

It is recommended to use a multi-sig wallet to handle ownership over contracts to minimize the risk of private key compromise.

Remediation:

Set a initial owner in the configs of deploy to a multi-sig wallet.

5.30. Chainlink extraArgs is immutable

Risk Level: Low

Status: See customer response below

Customer Response: The likelihood of Chainlink breaking backward compatibility is low. Contracts will be upgraded if necessary.

Contracts:

src/Nuon.sol

Description:

In the `_buildCCIPMessage` function in the `Nuon.sol` ,

the `extraArgs` parameter, as described in the Chainlink documentation, is designed to enable compatibility with future CCIP upgrades. To fully leverage this benefit, extraArgs should be mutable in production deployments. If extraArgs is immutable, the contract may not be adaptable to potential future changes in the Chainlink CCIP.

Remediation:

Consider making the extraArgs mutable.

References:

- https://docs.chain.link/ccip/api-reference/client#evmextraargs

5.31. Unrestricted Inflation Value Change

Risk Level: **Low**

Status: See customer response below

Customer Response: Those checks are already being done in the InflationOracle contract.

Contracts:

src/tokens/Nuon.sol

Description:

The `rebase` function allows for unrestricted changes to the `inflationValue`, which can lead to significant and potentially harmful fluctuations in the token's supply due to human error or malicious actions.

Remediation:

Implement a restriction on the `inflationValue` change, limiting it to a maximum percentage (e.g., 5%) from the previous value within a set period. This can be done by storing the last `inflationValue` and ensuring any new value does not exceed the allowed percentage change. If the change exceeds the limit, the transaction should be reverted.

5.32. _disableInitializers is missing in the `Redistributor`

Risk Level: Low

Status: Fixed in commit d52351

Contracts:

src/Redistributor.sol

Description:

`disableinitializers` is not called in Redistributor’s constructor

All initialize calls in the implementation contract should be blocked to protect against various attacks.

Remediation:

Add the following constructor:

```
constructor() {
    _disableInitializers();
}
```

5.33. Oracle will return stale price

Risk Level: Low

Status: See customer response below

Customer Response: This issue has been addressed with 5.22

Contracts:

src/InflationOracle.sol

Description:

If a new inflation value isn’t added successfully (e.g., due to a revert in the addRound function), the oracle will return the previous value, even if the update failed. This can lead to incorrect data being returned.

Remediation:

Track whether the last inflation value update was successful.

5.34. Public initialize functions

Risk Level: Low

Status: See customer response below

Customer Response: There is no gain for whoever attempts to do that.

Contracts:

scripts/NuonSetup.sol

Location: Lines: 83-86.

Description:

The `initialize` functions in multiple contracts (Nuon, WNuon, NuMint, etc.) are publicly accessible and vulnerable to front-running attacks. An attacker can monitor the deployer's address, detect the deployment transaction, and front-run the initialization call with malicious parameters. This forces the deployer to redeploy the contracts, incurring additional gas costs.

Below is the part of the `NuonSetup.sol` script, which allows front-running of initializers:

```
function run(string memory network_) external {
    // broadcast and get deployer PK
    deployerPrivateKey = vm.envUint("PRIVATE_KEY");
    vm.startBroadcast(deployerPrivateKey);

    // add dot to network
    network = _concat([".", network_]);

    // read config json file
    string memory configFile = vm.readFile(CONFIG);

    // load deploy flag and deploy contracts
    string[] memory contracts = vm.parseJsonKeys(configFile, network);
    _loadAndDeployContracts(contracts);

    // call initializers
    _initializeContracts(configFile); // **@audit can be front-run**

    vm.stopBroadcast();
}
```

Remediation:

Restrict initializer access to the deployer only to prevent front-running attacks.

5.35. Oracle will return the wrong price for asset if underlying aggregator hits `minAnswer`

Risk Level: Low

Status: See customer response below

Customer Response: Adding price boundaries would introduce unnecessary complexity and maintenance overhead, as the team would need to manage min/max price limits manually. The current implementation aligns with how Chainlink handles price feeds, and we do not see a practical benefit in modifying this behavior.

Contracts:

src/OracleRegistry.sol

Description:

Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow user to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded. When `latestRoundData()` is called it request data from the aggregator. The aggregator has a `minPrice` and a `maxPrice`. If the price falls below the minPrice instead of reverting it will just return the min price.

```
File: OracleRegistry.sol

    function getAssetPrice(IERC20 asset) external view returns (uint256) {
        OracleRegistryStorage storage $ = _getOracleRegistryStorage();

        AggregatorV3Interface oracle = $._registry[asset];
        int256 answer;
        (, answer,,,) = oracle.latestRoundData();
        if (answer <= 0) revert OracleRegistryInvalidOracle();

        return (uint256(answer) * 1e18) / 10 ** oracle.decimals();
    }
```

Remediation:

Add price boundaries:

```
if (answer >= maxPrice or answer <= minPrice) revert();
```

5.36. Lack of slashing recover logic

Risk Level: Low

Status: See customer response below

Customer Response: This is intended behavior and a core strength of the protocol. Stakers have the opportunity to earn strong returns by allocating to profitable strategies, but they also bear the risk of losses if decisions are poor. This risk/reward structure is designed to encourage responsible governance and proper asset allocation within the protocol.

Description:

The vulnerability lies in the protocol's primary mechanism for addressing insolvency, which is slashing tokens from the staking pool. Each time a slashing event occurs, stakers lose their funds without any mechanism to restore their stakes when the protocol generates profits.

Impact

1. **Disincentive for staking:**
   - Users are likely to avoid staking their funds as they risk consistent losses due to slashing, even when the protocol eventually becomes profitable.
7. **Decreased staking participation:**
   - A decline in staking participation undermines the protocol's security, governance, and solvency mechanisms, as staking pools are often a critical component for these functionalities.
8. **Protocol sustainability risks:**
   - The lack of a recovery mechanism for stakers may lead to long-term sustainability challenges for the protocol, as fewer participants engage with staking.
9. **User trust erosion:**
   - Continuous losses for stakers without a recovery mechanism erodes trust and reduces the protocol's attractiveness to both existing and potential participants.

Lack of Update Mechanism for `nuonStaking`**:**

- The protocol lacks a mechanism to update the `nuonStaking` variable. This becomes problematic when staking rewards are exhausted, as users lose incentives to stake their tokens, further diminishing staking participation.

Remediation:

Consider repaying slashed amount when the protocol gets profits.

5.37. Lack of auction ending logic

Risk Level: **Low**

Status: Fixed in commit 93413b.

Description:

The vulnerability arises from the protocol's dependency on the state of liquidation auctions for its core functionality. If an auction remains active for an extended period, the protocol's operation can effectively halt. This situation can occur due to the following reason:

Paused token:

- If a token involved in the auction is paused for transfers or other actions, it becomes impossible to complete the auction, leaving the protocol in a blocked state.

Impact:

- **Protocol freeze:**
  - Several critical functions in the protocol depend on there being no active liquidation auctions. If an auction cannot be resolved, these functions become unusable, leading to a partial or complete halt in the protocol's operations.
- **Economic impact:**
  - Locked liquidity and an inability to perform essential operations may result in loss of user trust, financial damage, and operational inefficiency.

Remediation:

- Implement a mechanism to handle prolonged or unresolvable auctions, such as:
  - Automatically canceling or invalidating auctions after a certain duration.
  - Allowing fallback mechanisms to resolve auctions through other means (e.g., seizing remaining assets or distributing losses).

- Introduce checks to ensure that paused tokens do not block the protocol, such as:
  - Skipping or replacing auctions involving paused tokens.
  - Allowing governance or emergency roles to override auctions involving problematic tokens.

5.38. DutchAuction to library

Risk Level: Low

Status: Fixed in pull 293

Contracts:

src/DutchAuction.sol

Description:

In the current implementation, some facets of the contract architecture inherit from the `DutchAuction` contract. This leads to unnecessary gas consumption during contract deployment, as the `DutchAuction` logic is replicated in every facet that inherits from it.

Remediation:

By converting the `DutchAuction` contract into a library, we can eliminate the need to inherit it in every facet. This will allow the `DutchAuction` logic to be deployed once, reducing the overall gas cost when deploying the facets.

5.39. Token distribution may run out of gas

Risk Level: Low

Status: See customer response below

- **Customer Response:** Tests show distribution to 500 recipients stays within gas limits. Exceeding this is highly unlikely, as even 20 recipients would be an extreme case.

Contracts:

src/Redistributor.sol

Location: Lines: 148. Function: `distributeTokens`.

Description:

In the `Redistributor` contract, `distributeTokens` function it goes through a list and sends tokens to users. In case if there will be a lot of users to receive tokens, transaction may run out of gas and fail.

```
 function distributeTokens() public whenNotPaused {
        ...
        for (uint256 i = 0; i < length; i++) {
            address recipient = $._recipients[i];
            uint256 accountBasisPoints = $._basisPoints[recipient];
            // transfer if basis points greater than 0
            if (accountBasisPoints > 0) {
                uint256 amount = (balance * accountBasisPoints) / MAX_BPS;
                //@audit we might have stuck tokens in the end, cuz more
tokens come,
                $._nuonToken.safeTransfer(recipient, amount);
            }
        }
    ...
    }
```

Remediation:

Consider making it possible to execute this array by smaller parts. For example, by passing initial index and final index to the `distributeTokens()` as arguments.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.