PUBLIC

# Code Assessment

of the Neulock

Smart Contracts

June 20, 2025

Produced for

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear Lucas,

Thank you for trusting us to help Studio V with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Neulock according to Scope to support you in forming an opinion on their security risks.

Studio V implements a storage contract to be used with an on-chain password manager. The access to the contract is gated through an NFT contract, with a points system attached.

The most critical subjects covered in our audit are access control, functional correctness, and Denial-of-Service vectors.

Security regarding access control is high, after an issue that allowed bypassing the token-gating system has been fixed, see Subscription Can Be Passed Around. Functional correctness is high, after issues with the royalty implementation has been fixed, see Royalty Payments With Native Tokens Break Marketplace Integrations. Security regarding Denial-of-Service vectors is high after the previously implemented refund mechanism was removed, see Refund Mechanism Can Be Abused to DOS a Series.

A general subject covered is gas efficiency. Gas efficiency was improved, see Burning Can Increase Withdraw Gas Cost and Gas Optimizations.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 2 |
| • Code Corrected | 2 |
| Medium-Severity Findings | 3 |
| • Code Corrected | 2 |
| • Specification Changed | 1 |
| Low-Severity Findings | 5 |
| • Code Corrected | 3 |
| • Specification Changed | 2 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Neulock repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 07 May 2025 | e44a32f7ea3f1f2017815c3d0802047a31dec904 | Initial Version |
| 2 | 16 Jun 2025 | 14136b5d1215b5716dd3f4e1a9cfaae21b9494aa | Version after 1st round of fixes |
| 3 | 20 Jun 2025 | c386f625fc60667aa48dfe54aeba43f63c4ac5d5 | Version after 2nd round of fixes |

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The following files of the `contracts/` folder are in the scope of this review:

```
current/
    EntitlementV1.sol
    LockV1.sol
    LogoV2.sol
    MetadataV2.sol
    NeuV2.sol
    StorageV2.sol
utils/
    Utils.sol
```

After Version 2, the scope was updated to the following:

```
current/
    EntitlementV2.sol
    LockV2.sol
    LogoV2.sol
    MetadataV3.sol
    NeuV3.sol
    StorageV3.sol
old/
    EntitlementV1.sol
    LockV1.sol
    MetadataV2.sol
    NeuV2.sol
    StorageV2.sol
utils/
    Utils.sol
```

### 2.1.1 Excluded from scope

Any file not explicitly mentioned in the scope section is excluded from the scope of this review. In particular, third-party libraries like `openzeppelin-contracts` and `openzeppelin-contracts-upgradeable` are assumed to work as intended and are out of the scope of this review. The proxy contracts and their potential admin are out of the scope of this review. The off-chain app, master key and passwords derivation schemes are out of the scope of this review.

## 2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Studio V offers the Neulock Web3 password manager. Encrypted application data is stored on IPFS at some content ID (CID). The core of the on-chain system is a simple key store contract whose role is to store the user-encrypted CIDs. The access to the system is regulated by NFTs representing an entitlement to use it.

### 2.2.1 NeuStorageV2

The `NeuStorageV2` contract is meant to be used behind a UUPS upgradeable proxy and is expected to replace `NeuStorageV1`. The main functionalities of the contract are the `saveData()` and `retrieveData()` functions:

- `saveData(tokenId, data)`: the main purpose of this function is to store the encrypted CID (`data`) on-chain. First, the caller is checked to be entitled to use the system by querying the *NeuEntitlementV1_* contract (details in the next section). Then, the `msg.value` is checked to be `0` if the entitlement is provided by a different token than a `NEU` NFT, or the caller is required to be the owner of the `NEU` NFT with the given `tokenId`. If some `ETH` was passed along with the call, it is used to increase the sponsor points for the `tokenId` (details in NEU NFT and Metadata). Sponsor points can only be accrued when accessing the system using a `NEU` NFT. Next, the `data` is stored in a mapping, where the key is the `msg.sender`.
- `retrieveData(owner)`: this function is a getter function that returns the data stored for the given `owner`.

The contract defines two roles: `DEFAULT_ADMIN_ROLE` and `UPGRADER_ROLE`. The `DEFAULT_ADMIN_ROLE` manages the roles in the contract. `UPGRADER_ROLE` is allowed to upgrade the contract and to reinitialize it with `initializeV2()`.

### 2.2.2 NeuEntitlementV1

The `NeuEntitlementV1` contract is meant to be used behind a UUPS upgradeable proxy. The contract manages a set of token contracts that are allowed to provide entitlement for users to use Neulock. The set is managed with the permissioned functions `addEntitlementContract()` and `removeEntitlementContract()`. The contract is called by `NeuStorageV2` to check whether a user is allowed to use the system (`hasEntitlement()`). The entitlement is checked by querying the user's balance in each of the entitlement contracts until one of them returns a positive balance. By default, the `NeuV2` NFT contract is added to the set of entitlement contracts upon initialization.

The contract defines three roles: `DEFAULT_ADMIN_ROLE`, `UPGRADER_ROLE`, and `OPERATOR_ROLE`. The `DEFAULT_ADMIN_ROLE` manages the roles in the contract. `UPGRADER_ROLE` is allowed to upgrade the contract and to initialize it with `initialize()`. The `OPERATOR_ROLE` is allowed to add and remove entitlement contracts.

### 2.2.3 NEU NFT and metadata

The logic of the `NEU` NFT is split into two contracts: `NeuV2` and `NeuMetadataV2`. The `NeuV2` contract is the token itself, while `NeuMetadataV2` manages the metadata of the token. The `NEU` token also implements the EIP7496 (https://eips.ethereum.org/EIPS/eip-7496), which enables NFT to implement certain traits. The only supported trait is the `points` trait, which allows each NFT to accumulate sponsor points.

The `NeuV2` contract exposes functions for users to buy and burn the token. The NEU token can also be minted by the `OPERATOR_ROLE` for free. For normal users, the NEU token can be bought (`safeMintPublic()`) per series, by paying the price in `ETH`. Public minting is only possible for series that have been set as `available`, but the `OPERATOR_ROLE` can also mint series that are not available. Each series has a finite number of tokens that can be minted at a fixed price. After a token has been bought, it can be refunded during a time period of 7 days after the minting date. Refunded tokens cannot be re-minted.

As mentioned in NeuStorageV2, the storage contract can request the increase of sponsor points of a token when data is saved. To achieve this, the `NeuV2` contract calls the `NeuMetadataV2` contract to increase the sponsor points of the token and transfers the `ETH` to the `NeuDaoLockV1`. Storage contracts must first be granted the `POINTS_INCREASER_ROLE` in the `NeuV2` contract to be allowed to increase the points. The `NeuMetadataV2` contract can be changed in `NeuV2` but this should be done carefully as such a change can break the accounting.

In order to create new series of tokens, the `OPERATOR_ROLE` of `NeuMetadataV2` can call the `addSeries()` function with a name, a price in `Gwei`, the ID of the first token of in the series, the maximum number of tokens in the series, and some color parameters for the SVG rendering of the logo for the token' series. The SVG data of the logo is provided by the `LogoV2` contract, which can be changed in `NeuMetadataV2`. The name must be unique, and the token IDs of the new series must not overlap with token IDs of previously added series. If the name does not start with `WAGMI`, the tokens in the series give governance rights.

Both `NeuV2` and `NeuMetadataV2` are meant to be used behind a UUPS upgradeable proxy.

The `NeuV2` contract defines four roles: `DEFAULT_ADMIN_ROLE`, `UPGRADER_ROLE`, `OPERATOR_ROLE`, and `POINTS_INCREASER_ROLE`. The `DEFAULT_ADMIN_ROLE` manages the roles in the contract. `UPGRADER_ROLE` is allowed to upgrade the contract and to initialize it with `initialize()`. The `OPERATOR_ROLE` is allowed to update the metadata contract, update the DAO lock contract, grant the `POINTS_INCREASER_ROLE`, mint tokens from available and not yet available existing series to arbitrary addresses for free, withdraw any `ETH` balance (such as minting fees) that does not need to be kept for refunds, change the rate of wei/sponsor points, and change the metadata URI for the trait of `NeuMetadataV2`. The `POINTS_INCREASER_ROLE` is expected to be granted to one or more `NeuStorageV2` contracts.

The `NeuMetadataV2` contract defines four roles: `DEFAULT_ADMIN_ROLE`, `UPGRADER_ROLE`, `OPERATOR_ROLE`, and `NEU_ROLE`. The `DEFAULT_ADMIN_ROLE` manages the roles in the contract. `UPGRADER_ROLE` is allowed to upgrade the contract and to initialize it with `initialize()`. The `OPERATOR_ROLE` is allowed to add new series, change the availability status of a series, change the price of the tokens in a series, and update the address of the logo contract. The `NEU_ROLE` is expected to be given only once and to the `NeuV2` contract.

### 2.2.4 NeuDaoLockV1

The `NeuDaoLockV1` contract receives the `ETH` amounts used to increase sponsor points and locks them. It is possible to unlock the tokens for the currently set DAO by having the holders of at least seven `NEU` tokens with governance rights call the `unlock()` function, which will register the token ID as a key token. The holder of a key token can also cancel their participation by calling `cancelUnlock()`. Once the required number of key tokens is reached, anyone can call `withdraw()` to transfer the contract balance to the DAO. The `OPERATOR_ROLE` can change the DAO address. This will also reset the number of key tokens.

The contract defines two roles: `DEFAULT_ADMIN_ROLE` and `OPERATOR_ROLE`. The `DEFAULT_ADMIN_ROLE` manages the roles in the contract. `OPERATOR_ROLE` is allowed to update the address of the DAO.

### 2.2.5 Changes in V2

- All contracts except `NeuLogoV2` have an increased version number.
- An entitlement cooldown period of 1 week is enforced in case of frequent transfer of the `NEU`, see specific rules described in Subscription Can Be Passed Around.
- The refund mechanism has been removed.
- The metadata contract of the `NEU` contract cannot be updated anymore after `initializeV3()` is called.
- Points can be added by anyone to arbitrary existing tokenIds. The `POINTS_INCREASER_ROLE` has been removed.
- A new function `NeuStorageV3.saveDataV3()` has the same functionality as `saveData()`, but skips the points increase.

## 2.3 Trust Model

- Users: not trusted
- Holders of the `DEFAULT_ADMIN_ROLE` on each contract: Fully trusted to manage the contract-specific roles in a non-adversarial manner. In the worst case, critical roles such as `UPGRADER_ROLE` can be granted, which can change the implementation logic of some contracts.
- Holders of the `UPGRADER_ROLE` on each contract: fully trusted to manage the contract upgrades and (re-)initializations in a non-adversarial manner. In the worst case, this role can change the implementation logic of their respective contracts. A compromised `UPGRADER_ROLE` could overwrite the data saved by `NeuStorageV2`, steal the `ETH` locked in `NeuV2`, or simply break the system.
- `OPERATOR_ROLE` of `NeuEntitlementV1`: Partially trusted. In the worst case, a compromised operator could add entitlement contracts, which could be used to grant access to the system to arbitrary users. They can also remove entitlement contracts, which would disallow users from using the system that should be allowed to.
- `OPERATOR_ROLE` of `NeuV2`: Fully trusted as of **Version 2**. In the worst case, a compromised operator could change the royalty recipient address, which could allow them to steal all the royalties paid by trading platforms. Additionally, they can mint tokens for free.
- `OPERATOR_ROLE` of `NeuMetadataV2`: this role is trusted to manage the series, their price and the logo contract address in a non-adversarial manner. In the worst case, this role can create new series with arbitrary parameters or change the price of existing series.
- `OPERATOR_ROLE` of `NeuDaoLockV1`: this role is trusted in setting the new DAO address in a non-adversarial manner. In the worst case, this role can set an arbitrary address and eventually steal the locked funds if it is approved by enough holders.

- POINTS_INCREASER_ROLE: Partially trusted. The POINTS_INCREASER_ROLE is only partially trusted, as the NeuV2 contract checks that any call coming from it sends enough ETH. However, it is trusted to pass the correct tokenId that paid for the call. This role has been removed in Version 2.
- NEU_ROLE: this role is trusted and is expected to be held only by one address, the NEU token contract. If this role is granted to more addresses, it could disrupt the operations in the original NEU token.
- The royalty receiver address is assumed to be able to receive and handle the native token, and to be able to handle arbitrary tokens that can be used as payment for the NFT.

It is assumed that the initialization functions (initialize, initializeVX, ...) are always called in order by the UPGRADER_ROLE.

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |

| | |
|---|---|
| High-Severity Findings | 2 |

- Royalty Payments With Native Tokens Break Marketplace Integrations Code Corrected
- Subscription Can Be Passed Around Code Corrected

| | |
|---|---|
| Medium-Severity Findings | 3 |

- Metadata Contract Can Be Updated Code Corrected
- Refund Mechanism Can Be Abused to DOS a Series Specification Changed
- Royalties in Non-Native Tokens Are Locked Code Corrected

| | |
|---|---|
| Low-Severity Findings | 5 |

- Burning Can Increase Withdraw Gas Cost Specification Changed
- Inconsistency in Initialization Steps Code Corrected
- Inconsistent Refund Period Specification Changed
- Missing Events Code Corrected
- Missing Input Sanitization Code Corrected

| | |
|---|---|
| Informational Findings | 12 |

- Missing Getter for Number of Entitlement Contracts Code Corrected
- NEU Contract Is Not Necessarily the First Entitlement Contract Code Corrected
- EIP7496 Compliance Code Corrected
- Gas Griefing by Adding Many Unlock Keys Code Corrected
- Gas Optimizations Code Corrected
- Loop Iterator Increment Optimization Is Unnecessary Code Corrected
- Missing Natspec Code Corrected
- NEU Can Be Removed From the Entitlement Contracts List Code Corrected
- Storage Variable Without Explicit Visibility Code Corrected
- Uninitialized Dependency Code Corrected
- Unused Role in NeuMetadataV2 Code Corrected
- setStorageContract Does Not Remove Previous Contract Specification Changed

## 6.1 Royalty Payments With Native Tokens Break Marketplace Integrations

Design High Version 1 Code Corrected

*CS-NLCK-001*

The NEU token implements EIP2891 (https://eips.ethereum.org/EIPS/eip-2981), which defines a recipient for royalty payments. As the royalties can be in ETH or other ERC20-like tokens, the recipient address should be able to handle such tokens. In the case of NeuV2, the contract does not have a receive() function, so royalty payments with ETH will revert. This will cause any sale of NEU with ETH on marketplaces that pay royalties to revert. As a result, only marketplaces that do not pay royalties would be able to trade NEU.

---

**Code corrected:**

The contract NeuV3 allows to set a new royalty recipient with initializeV3(), and update it with the function setRoyaltyReceiver() callable only by the OPERATOR_ROLE. The recipient is expected to be able to receive the native token.

## 6.2 Subscription Can Be Passed Around

Design High Version 1 Code Corrected

*CS-NLCK-002*

The NEU NFT represents an entitlement to the Neulock services. However, since it can be transferred, in theory only one user needs to buy the NFT to enable anyone to use the service. This could create secondary markets where the NFT is lent to users to temporarily use Neulock before returning it, without having to pay the full price of the NFT.

---

**Code corrected:**

The NeuV3 NFT has a new associated data entitlementDate that is used to control the entitlement of a token ID. It works as follows:

- on mint, entitlementDate is not touched and has value 0
- on burn, entitlementDate is reset
- on a transfer, there are 3 cases:

  1. entitlementDate > block.timestamp: the transfer happens during the cooldown period, so the previous owner never received entitlement. The cooldown is not changed.
  2. entitlementDate + 1 week <= block.timestamp: the current owner was holding the token for at least 1 week after the entitlement date, the new owner gains entitlement at block.timestamp + 1. The entitlementDate of the token is updated to block.timestamp + 1.
  3. entitlementDate <= block.timestamp < entitlementDate + 1 week: the entitlement date has been reached but the new owner has to wait for entitlementDate + 1 week to gain entitlement. The entitlementDate of the token is updated to entitlementDate + 1 week.

This mechanism allows normal transfer of the token, while making flashloans and other lending services impractical.

## 6.3 Metadata Contract Can Be Updated

Design | Medium | Version 1 | Code Corrected

*CS-NLCK-003*

The `NeuV2.setMetadataContract()` function allows to change the address of the metadata contract. The main intended use is to set it after the contract has been initialized. However, once the address is set for the first time, the `OPERATOR_ROLE` is still allowed to update it. Updating the metadata is risky, as it is tracking some internal accounting such as the refund values.

---

**Code corrected:**

The metadata contract is now set in `NeuV3.initializeV3()` and cannot be changed afterwards.

## 6.4 Refund Mechanism Can Be Abused to DOS a Series

Design | Medium | Version 1 | Specification Changed

*CS-NLCK-004*

When a series is added by the operator, a finite number of tokens and a fixed price for them must be set. This, in conjunction with the refund feature, opens a DOS vector for the series. An attacker could buy all the tokens of a series and have them refunded, effectively making them unavailable for other users and forcing Studio V to issue a new series. Due to the refund, this has no cost aside from gas.

Additionally, the refunded tokens will increase the cost of the privileged `withdraw` function, as described in Burning Can Increase Withdraw Gas Cost.

---

**Spec changed:**

The refund mechanism has been completely removed from the codebase.

## 6.5 Royalties in Non-Native Tokens Are Locked

Design | Medium | Version 1 | Code Corrected

*CS-NLCK-005*

As explained in Royalty Payments With Native Tokens Break Marketplace Integrations, the `NEU` token can receive royalties. However, the current implementation does not offer the possibility to withdraw non-native tokens (e.g. ERC20-like tokens). As a result, any non-native tokens received as royalties will be locked in the contract and cannot be recovered unless it is upgraded.

---

**Code corrected:**

The contract `NeuV3` allows to set a new royalty recipient with `initializeV3()`, and update it with the function `setRoyaltyReceiver()` that callable by the `OPERATOR_ROLE`. The recipient is expected to be able to handle non-native payment tokens.

## 6.6 Burning Can Increase Withdraw Gas Cost

Design Low Version 1 Specification Changed

*CS-NLCK-006*

In NeuMetadataV2, the `sumAllRefundableTokensValue()` function loops through all series and finds the refundable tokens. It loops from the back, terminating when it finds the first non-refundable token. This means it should only look at one token per series if all of them are older than the refund window. However, if the last tokens have been burned, the loop will continue through the series until it finds an unburned token. This means that a burned token at the end of a series will increase the gas cost of all future calls.

This can increase the gas cost of the `sumAllRefundableTokensValue()` function, which is called in the `withdraw()` function. If there are many burnt tokens, it may not be possible to withdraw, as the gas cost may exceed the block gas limit.

---

**Spec changed:**

The refund mechanism has been completely removed from the codebase. Burning tokens at the end of the series can still increase the cost of `NeuMetadataV3._hasRefundableTokens()`, which is used by `NeuMetadataV3.initializeV3()`. This is acceptable as `NeuMetadataV3.initializeV3()` is a permissioned function that will be called only once.

## 6.7 Inconsistency in Initialization Steps

Design Low Version 1 Code Corrected

*CS-NLCK-007*

In the `NeuStorageV2` contract, the `initialize()` function allows to fully initialize a new deployment without calling `initializeV2()`, as it also sets the `_entitlementContract`. In this case, `initializeV2()` allows the `UPGRADER_ROLE` to update the `_entitlementContract` later if `initializeV2()` was not called on deployment.

On the other hand, initializing the `NeuV2` contract requires to call `initialize()` first and then `initializeV2()` to complete the setup.

One of the two options should be chosen for the sake of code consistency.

---

**Code corrected:**

All contracts now require calling the initializers in sequence to fully initialize the contract.

## 6.8 Inconsistent Refund Period

Correctness Low Version 1 Specification Changed

*CS-NLCK-008*

There is an inconsistency between the behaviors of `NeuMetadataV2.getRefundAmount()` and `NeuMetadataV2.sumAllRefundableTokensValue()` at the refund window boundary. While the function `NeuMetadataV2.getRefundAmount()` will revert if the time elapsed since `mintedAt` is exactly `REFUND_WINDOW`, the function `NeuMetadataV2.sumAllRefundableTokensValue()` will take the token into account if the elapsed time is exactly `REFUND_WINDOW`.

In the code:

```
function getRefundAmount(uint256 tokenId) external view returns (uint256) {
    ...
    require(block.timestamp - metadata.mintedAt < REFUND_WINDOW, "Refund window has passed");
    ...
}
```

excludes the equality, but

```
function sumAllRefundableTokensValue() external view returns (uint256) {

    ...
            if (block.timestamp - metadata.mintedAt > REFUND_WINDOW) {
                // All tokens before this one in the series are also expired
                break;
            }

            totalValue += metadata.originalPriceInGwei;
    ...
}
```

will add the `originalPriceInGwei` in case of equality

---

**Spec changed:**

The refund mechanism has been completely removed from the codebase.

## 6.9 Missing Events

Design | Low | Version 1 | Code Corrected

*CS-NLCK-009*

Events should be emitted every time an important storage update is done. Ideally, an observer should be able to reconstruct the contract state by solely looking at the events. The following actions should emit an event:

1. In the `initialize()` of `NeuEntitlementV1`, the `EntitlementContractAdded` event should be emitted after adding the `neuContract` to the set of entitlement contracts.
2. In the `initialize()` function of `NeuMetadataV2`, the `LogoUpdated` event should be emitted after setting the address of the logo contract.

---

**Code corrected:**

The described events are now emitted in the initializers.

## 6.10 Missing Input Sanitization

Design | Low | Version 1 | Code Corrected

*CS-NLCK-010*

1. When a series is added or when the price is updated, the price is not checked to be non-zero. A `0` price will make `isUserMinted()` return `false` even if it is minted by a user.

2. When a series is added, `maxTokens` is not enforced to be at least 1. Setting `maxTokens=0` would create a 0 length series and render the ID `firstToken` unusable.

---

**Code corrected:**

1. FIXED. The price is enforced to be strictly greater than 0.
2. FIXED. The `maxTokens` value is enforced to be strictly greater than 0.

## 6.11 Missing Getter for Number of Entitlement Contracts

Informational | Version 2 | Code Corrected

*CS-NLCK-021*

The `NeuEntitlementV2` contract exposes the `entitlementContractsV2()` function which take an index as a parameter, but there is no easy way to determine the maximum value of the index before the function reverts. This could be solved by adding a function that queries the number of entitlement contracts.

---

**Code corrected:**

The function `entitlementContractsLength()`, returning the number of entitlement contracts has been added.

## 6.12 NEU Contract Is Not Necessarily the First Entitlement Contract

Informational | Version 2 | Code Corrected

*CS-NLCK-022*

The function `NeuEntitlementV2.initializeV2()` assumes the first address in the `entitlementContracts` will be the address of the `NEU` contract, but this is not necessarily true, as the `NEU` contract address might have been removed from the set before the call to `NeuEntitlementV2.initializeV2()` was done. This would have the effect to have some entitlement contract that is not the `NEU` token in the `_neuContract` storage variable.

However, this can only happen if the trusted `OPERATOR_ROLE` removes the contract.

---

**Code corrected:**

The function `NeuEntitlementV2.initializeV2()` has been updated to take the address of the `NEU` contract as an argument. The trusted `UPGRADER_ROLE` is expected to provide the correct address.

## 6.13 EIP7496 Compliance

Informational | Version 1 | Code Corrected

*CS-NLCK-011*

The EIP7496 standard specifies the following:

`The traitKey SHOULD be a keccak256 hash of a human readable trait name.`

(https://eips.ethereum.org/EIPS/eip-7496#keys--names)

However, the `traitKey` used in `NeuMetadataV2._getTraitValue()` is the trait name instead of its `keccak256` hash.

---

**Code corrected:**

The trait key has been replaced by the hash of the human readable trait name "points".

## 6.14 Gas Griefing by Adding Many Unlock Keys

Informational | Version 1 | Code Corrected

*CS-NLCK-012*

In the function `NeuDaoLockV1.setNeuDaoAddress()`, the deletion of `keyTokenIds` will iterate through the whole array to reset its content. The cost of calling the function increases with every key token ID added to the array, and the array does not have an upper bound for its length (aside from the `totalSupply` of NEU tokens). If many token IDs can be added by an attacker, updating the address of the Neu DAO could become expensive or impossible (due to hitting the block gas limit).

---

**Code corrected:**

The `keyTokenIds` array was replaced with an EnumerableSet to allow for efficient removal of keys without iterating through the entire array. This change ensures that the cost of updating the Neu DAO address remains constant, regardless of the number of keys added. The storage layout has changed. The `NeuDaoLockV2` contract will be deployed as a new contract, and the `NeuDaoLockV1` contract will be deprecated.

## 6.15 Gas Optimizations

Informational | Version 1 | Code Corrected

*CS-NLCK-013*

The following possible gas optimizations have been identified:

1. In `NeuEntitlementV1.userEntitlementContracts()`, overwriting the length of `userEntitlements` with `count` would be more gas efficient than looping over the array.
2. In `NeuDaoLockV1.withdraw()`, the `address(0)` check is redundant as there cannot be key tokens if the `neuDaoAddress` is `address(0)`.
3. In `NeuMetadataV2.setSeriesAvailability()`, keeping the `series` as storage pointer would be more efficient than copying the whole struct in `memory`, as only the first word is needed.
4. The availability of series could be stored in a boolean mapping in order to avoid the cost of the loop in `NeuMetadataV2._isSeriesAvailable`.
5. In `NeuMetadataV2.createTokenMetadata()`, an `SLOAD` is done for each `_series[seriesIndex]` access. It would be more efficient to load the `Series` struct in memory once and read its members from there.

6. In `NeuMetadataV2._givesGovernanceAccess()`, an `SLOAD` is done for each `_series[seriesIndex].name[i]` access. It would be more efficient to load the name on the stack once and read each letter from there.
7. In `NeuMetadataV2.setSeriesAvailability()`, the event `SeriesAvailabilityUpdated` is emitted with the same params in both branches of the second `if-else` construct. Emitting the event out of the `if-else` would reduce the size of the bytecode and thus save some gas.
8. In `NeuMetadataV2.addSeries()`, the variables `seriesIndex` and `seriesLength` track the same value and could be merged into one.
9. The values returned by `NeuV2._privateMint()` are never used. Not returning anything would save some gas.
10. The event `TraitUpdated` is emitted twice in a single call to `NeuV2._increaseSponsorPoints()`. Once in `NeuV2._increaseSponsorPoints()` and once in `NeuMetadataV2.increaseSponsorPoints()`.
11. In `Bytes8Utils.toString()`, the routine that increases `i` can be done in an `unchecked` block to avoid unnecessary overflow checks.
12. In `Bytes8Utils.toString()`, the second loop can be done over a new `uint256` iterator, capped to `i` iterations to avoid the redundant `data[i] != 0` check. Having the iterator as `uint256` will also make its increment unchecked (see the first compiler feature of https://github.com/ethereum/solidity/blob/develop/Changelog.md#0822-2023-10-25).

**Version 2**

1. In `NeuEntitlementV2.initializeV2()`, the length of the `entitlementContracts` array could be cached to avoid an SLOAD at each iteration

---

**Code corrected:**

1. FIXED. The result array is resized in assembly.
2. FIXED. The redundant check has been removed.
3. FIXED. Storage pointers are used.
4. FIXED. A bit map is used to track the available series.
5. FIXED. The `createTokenMetadataV3()` function loads the series into memory before reading from it.
6. FIXED. The name is loaded in the context before executing the loop.
7. FIXED. The emission of the event has been moved in specialized functions.
8. FIXED. The variable `seriesLength` has been removed.
9. FIXED. The return values `NeuV3._privateMint()` have been removed.
10. FIXED. The event is only emitted in `NeuV2._increaseSponsorPoints()`.
11. FIXED. The routine is done in an `unchecked` block.
12. FIXED. The loop logic has been simplified.

**Version 2**

1. FIXED. The length of the `entitlementContracts` array is cached before the loop.

## 6.16 Loop Iterator Increment Optimization Is Unnecessary

Informational Version 1 Code Corrected

*CS-NLCK-014*

In `NeuMetadataV2.getTraitValues()`, the `i++` operation can be left in the loop declaration and doesn't need to be explicitly `unchecked`. The compiler automatically does the optimization since `0.8.22`.

---

**Code corrected:**

The index increment has been moved in the loop declaration.

## 6.17 Missing Natspec

Informational Version 1 Code Corrected

*CS-NLCK-015*

The contracts currently do not include any NatSpec comments to document functions, parameters, return values, events, or contract metadata.

NatSpec comments can provide clear and concise documentation for users and developers interacting with the smart contracts. They can also help maintainability of the codebase. As such, it is considered a best practice to include NatSpec comments.

---

**Code corrected:**

NatSpec has been added to all the contracts in `current/`.

## 6.18 NEU Can Be Removed From the Entitlement Contracts List

Informational Version 1 Code Corrected

*CS-NLCK-016*

The `NEU` contract should always stay in the set of entitlement contracts, but a compromised or inattentive `OPERATOR_ROLE` could remove it with `NeuEntitlementV1.removeEntitlementContract()`. Explicitly preventing this case could increase the user's confidence that their `NEU` token can always be used with the system, as long as no upgrade allowing the removal is made.

---

**Code corrected:**

The Entitlement contract now tracks the `NEU` contract separately and it cannot be removed.

## 6.19 Storage Variable Without Explicit Visibility

Informational Version 1 Code Corrected

*CS-NLCK-017*

By default, a storage variable has its visibility set as `internal`. Even though the default behavior is conservative, it is best practice to always explicitly set the visibility of a storage variable.

The following storage variables are missing an explicit visibility:

- `_traitMetadataURI` in `NeuMetadataV2`

---

**Code corrected:**

All the storage variables now have an explicit visibility.

## 6.20 Uninitialized Dependency

**Informational** | **Version 1** | **Code Corrected**

*CS-NLCK-018*

In the `NeuV2` contract initializer functions, `__ERC721Royalty_init()` is never called. Even though in its current state the function is a no-op, it is good practice to call the initializer function in every dependency.

---

**Code corrected:**

The newly added function `NeuV3.initializeV3()` calls `__ERC721Royalty_init()`.

## 6.21 Unused Role in NeuMetadataV2

**Informational** | **Version 1** | **Code Corrected**

*CS-NLCK-019*

In `MetadataV2`, there is an unused role called `STORAGE_ROLE`. This role is not used anywhere.

---

**Code corrected:**

The unused role has been removed.

## 6.22 setStorageContract Does Not Remove Previous Contract

**Informational** | **Version 1** | **Specification Changed**

*CS-NLCK-020*

In NeuV2, the function `setStorageContract` does not remove the previous contract from the `POINTS_INCREASER_ROLE`, but adds the new one. If it should be removed, the `DEFAULT_ADMIN_ROLE` can remove it manually.

---

**Spec changed:**

The `POINTS_INCREASER_ROLE` has been fully removed. Now, `increaseSponsorPoints()` can be called by anyone on the NeuV3 contract.