BURRA SEC

# Remilux Staking Security Review

Reviewed by: windhustler

July 14, 2025

# Remilux Staking Security Review Report

Burra Security

July 14, 2025

## Introduction

A time-boxed security review of the **Remilux Staking** protocol was done by **Burra Security** team, focusing on the security aspects of the smart contracts.

## Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

## About Burra Security

Burra Sec offers security auditing and advisory services with a special focus on cross-chain and interoperability protocols and their integrations.

## About Remilux Staking

RemiluxStaking is an NFT staking contract that allows users to stake Remilux NFTs for predetermined lock periods (30, 60, or 90 days) with different multipliers for aura calculation rewards. The contract features season-based management where administrators can start new seasons, and users can stake, unstake (after lock periods end), or extend their lock durations while earning rewards based on their staking commitment.

## Severity classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

**Impact** - The technical, economic, and reputation damage from a successful attack

**Likelihood** - The chance that a particular vulnerability gets discovered and exploited

**Severity** - The overall criticality of the risk

**Informational** - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

## Security Assessment Summary

***review commit hash* - d9c1af226e70a49e708c81d4dd6341624f7225e0**

### Scope

The following smart contracts were in the scope of the audit:

- `RemiluxStaking.sol`

---

## Findings Summary

| ID | Title | Severity | Status |
|---|---|---|---|
| M-01 | Incorrect participant count due to global stake tracking | Medium | - |
| M-02 | Participant count inflation via repeated stake/unstake cycling | Medium | - |

| ID | Title | Severity | Status |
|---|---|---|---|
| L-01 | Replace Ownable with Ownable2Step | Low | - |
| L-02 | Ineffective status check for current season | Low | - |
| I-01 | StakedNFT struct not updated after unstaking | Info | - |

## Detailed Findings

### [M-01] Incorrect participant count due to global stake tracking

#### Target

- RemiluxStaking.sol#L178-L180

#### Severity

- Impact: Medium
- Likelihood: Medium

#### Description

When a user stakes, the protocol checks whether their `totalStaked` is zero. If so, it increments the participant count for the current season:

```
// If user is staking for the first time this season, increment
    participants
if (userInfo.totalStaked == 0) {
     seasons[currentSeason].totalParticipants++;
}
```

The issue arises because this condition checks the user's total staking history, not whether they've staked in the current season. For example, if a user staked during Season 1 (increasing `seasons[1].totalParticipants`), then later stakes again in Season 2, the condition fails since `totalStaked` is already nonzero. As a result:

- `seasons[1].totalParticipants = 1`
- `seasons[2].totalParticipants = 0` // even though the user staked in Season 2

The root cause is relying on the global `totalStaked` value instead of tracking staking activity per season.

#### Recommendation

Refactor the logic to track user participation on a per-season basis to ensure accurate participant counts.

## [M-02] Participant count inflation via repeated stake/unstake cycling

#### Target

- RemiluxStaking.sol#L178

#### Severity

- Impact: Medium
- Likelihood: Medium

#### Description

A user can artificially inflate the participant count of a season by repeatedly staking and unstaking:

- The user stakes during Season 1 → `totalStaked` becomes 1, and `totalParticipants` increases by 1.
- After the lock period, the user unstakes → `totalStaked` becomes 0.
- The user then stakes again → `totalStaked` goes back to 1, and `totalParticipants` increases again.

As a result:

- `seasons[1].totalParticipants = 2`
- `userInfo.totalStaked = 1`

This happens because the participant check only verifies if `totalStaked == 0` to decide whether to increment the count:

```
// On staking
if (userInfo.totalStaked == 0) {
    seasons[currentSeason].totalParticipants++;
}
```

```
// On unstaking
userInfo.totalStaked--;
```

The system incorrectly assumes that a zero `totalStaked` means the user has never participated in the season, allowing the same user to be counted multiple times.

### Recommendation

Track participation based on the user's address per season to ensure each user is counted only once.

## [L-01] Replace Ownable with Ownable2Step

### Target

- RemiluxStaking.sol#L16

### Severity

- Impact: High
- Likelihood: Low

### Description

The `RemiluxStaking` contract uses OpenZeppelin's `Ownable` pattern for access control, which implements a single-step ownership transfer mechanism. This design allows the current owner to immediately transfer ownership to any address by calling `transferOwnership()`, without requiring confirmation from the recipient address.

This creates a risk where ownership can be accidentally transferred to an incorrect address. Once ownership is transferred to an inaccessible address, all owner-only functions become permanently unusable, effectively breaking critical contract functionality.

### Recommendation

Replace `Ownable` with `Ownable2Step` to require two-step ownership transfers.

## [L-02] Ineffective status check for current season

### Target

- RemiluxStaking.sol#L137-L140
- RemiluxStaking.sol#L271-L274

### Severity

- Impact: Low
- Likelihood: Low

### Description

The following `require` statement always passes in all relevant functions, because each new season is initialized with `isActive` set to **`true`**. As a result, `currentSeason` will always be active, making this check ineffective:

```
require(
    seasons[currentSeason].isActive,
    "Current season is not active"
);
```

Although the admin sets the previous season's `isActive` to **`false`** when calling `startNewSeason`, the issue lies in the way the current season is initialized:

```
// Start new season
uint256 newSeasonNumber = currentSeason + 1;
seasons[newSeasonNumber] = Season({
    seasonNumber: newSeasonNumber,
    startTime: block.timestamp,
    endTime: 0,
    isActive: true,
    totalParticipants: 0,
});

currentSeason = newSeasonNumber;
```

Since the `isActive` flag for the current season is always set to **`true`**, the `require` check for season activity becomes redundant.

#### Recommendation

Introduce a dedicated function (with appropriate access control) to explicitly manage the active state of `currentSeason`, ensuring better control and reliability in protocol logic.

## [I-01] StakedNFT struct not updated after unstaking

#### Target

- RemiluxStaking.sol#L216

#### Severity

Informational

#### Description

The `RemiluxStaking::unstakeNFT` function only sets the `isStaked` field to **`false`** when unstaking an NFT, but leaves all other fields in the `StakedNFT` struct unchanged.

External contracts calling `getStakedNFT()` on an unstaked token will receive this stale data, which could lead to incorrect assumptions about the NFT's staking history or current state.

#### Recommendation

Update the `unstakeNFT` function to properly clean up the staking state and record the actual unstake time. Consider setting `lockEndTime` to `block.timestamp` to indicate when unstaking occurred.

Alternatively, consider resetting the struct to its default values completely.