Nereus LP Staking v2 Contracts Code Audit and Verification by Ambisafe Inc.

February, 2023

Oleksii Matiiasevych

1. **Introduction.** Nereus Finance. requested Ambisafe to perform a code audit of the LP Staking v2 contracts. The contracts in question can be identified by the following git commit hash:

   `033deeec4e9b73a0a93d88034734f4e3274e8edf`

   The scope of the audit is Staking contract and interface.

   During the initial code audit, Nereus Finance team applied a number of updates which can be identified by the following git commit hash:

   `8b0473691dba3640f028aa89be730361737a7fcf`

   Additional verification was performed after that.

2. **Disclaimer.** The code audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bugfree status. The code audit documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of Ambisafe.

3. **Executive Summary.** There are **no** known compiler bugs for the specified compiler version (0.8.17), that might affect the contracts' logic. There were 0 critical, 0 major, 0 minor, 34 informational and optimizational findings identified in the initial version of the contracts. Some of the findings were addressed while others remained acknowledged.

4. **Critical Bugs and Vulnerabilities.** No critical bugs or vulnerabilities were found.

5. **Initial Line By Line Review. Fixed Findings.**

5.1. IStaking, line 9. Note, the **UserInfo.lastRewardUpdateTime** property is not used.

5.2. Staking, line 12. Note, the **treasuryAddress** state variable could be made public.

5.3. Staking, line 13. Note, the **rewardTokensPerSecond** state variable is not used.

6. **Verification Line By Line Review. Acknowledged Findings.**

6.1. Staking, line 15. Optimization, the **REWARD_TOKEN_PRECISION** state variable could be made constant.

6.2. Staking, line 39. Optimization, the **REWARD_TOKEN_PRECISION** logic could be redesigned to only be used for division on **claim**, when the reward tokens are moved.

6.3. Staking, line 55. Optimization, the **updatePoolRewards()** reads pool properties from storage multiple times.

6.4. Staking, line 56. Optimization, the **updatePoolRewards()** reads **currentEmissionPoint** value from storage multiple times.

6.5. Staking, line 57. Optimization, the **updatePoolRewards()** reads **emissionSchedule[].endTime** value from storage twice.

6.6. Staking, line 76. Note, the **updatePoolRewards()** function has an excessive code duplication based on the emission points equality condition. The loop covers the equal case as well and could be used every time. Only the update of the **currentEmissionPoint** storage variable could be avoided if the emission points are equal.

6.7. Staking, line 125. Note, the **deposit()** function uses standard **transferFrom()** function which might fail on a non-comlpiant ERC20 tokens, consider using **safeTransferFrom()** function instead.

6.8. Staking, line 133. Optimization, the **deposit()** function reads **staker.amount** value from storage multiple times.

6.9. Staking, line 139. Optimization, the **deposit()** function reads **pool.totalSupply** value from storage multiple times.

6.10. Staking, line 142. Optimization, the **deposit()** function reads **pool.accumulatedRewardsPerShare** value from storage multiple times.

6.11. Staking, line 158. Optimization, the **withdraw()** function reads **pool.accumulatedRewardsPerShare** from storage twice.

6.12. Staking, line 166. Note, the **withdraw()** function uses standard **transferFrom()** function which might fail on a non-comlpiant ERC20 tokens, consider using **safeTransferFrom()** function instead.

6.13. Staking, line 174. Optimization, the **withdraw()** function reads **poolStakers[msg.sender].amount** from storage twice.

6.14. Staking, line 179. Note, the **withdraw()** function uses standard **transfer()** function which might fail on a non-comlpiant ERC20 tokens, consider using **safeTransfer()** function instead.

6.15. Staking, line 196. Note, the **emergencyWithdraw()** function uses standard **transfer()** function which might fail on a non-comlpiant ERC20 tokens, consider using **safeTransfer()** function instead.

6.16. Staking, line 207. Optimization, the **claim()** function reads **pool.accumulatedRewardsPerShare** from storage twice.

6.17. Staking, line 219. Optimization, the **claim()** function reads **poolStakers[msg.sender].amount** from storage twice.

6.18. Staking, line 222. Note, the **claim()** function uses standard **transferFrom()** function which might fail on a non-comlpiant ERC20 tokens, consider using **safeTransferFrom()** function instead.

6.19. Staking, line 260. Optimization, the **getClaimableRewards()** function excessively updates the **lastRewardUpdateTime** local variable and then returns.

6.20. Staking, line 264. Note, the **getClaimableRewards()** function has an excessive code duplication based on the emission points equality condition. The loop covers the equal case as well and could be used every time.

6.21. Staking, line 314. Note, the **calculateAccruedRewardsFromTo()** function could return wrong results in case **totalSupply** went to 0 for some time after the **fromDate**.

6.22. Staking, line 323. Optimization, the **calculateAccruedRewardsFromTo()** function should break the loop in case **startTime >= toDate**, because later emission schedules will have **startTime** even higher.

6.23. Staking, line 357. Optimization, the **getTotalUnclaimedRewardsForDate()** could return **calculateAccruedRewardsFromTo(poolStartTime, date)** and avoid extra **getTotalUnclaimedRewards()** call.

6.24. Staking, line 404. Optimization, the **editEmissionSchedule()** function excessively invokes **onlyOwner()** modifier, while it is already present in the **addEmissionsPoints()** function.

6.25. Staking, line 413. Note, the **editEmissionSchedule()** function will revert with an underflow if called when the **emissionSchedule** list is still empty.

6.26. Staking, line 439. Optimization, the **calculateEmissionPoint()** function reads **emissionSchedule.length** value from storage multiple times.

6.27. Staking, line 455. Optimization, the **calculateEmissionPoint()** function reads **emissionSchedule[i]** value from storage twice for every loop iteration except for the first and last.

6.28. Staking, line 477. Optimization, the **getEmissionPoints()** function **fromPoint >= 0** condition is always true.

6.29. Staking, line 505. Note, the **getEmissionPoint()** function will revert if the **emissionSchedule** list is empty.

Oleksii Matiiasevych