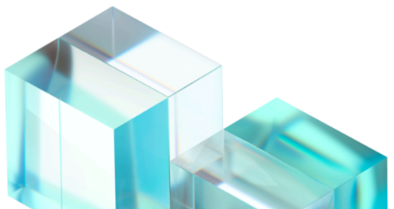
LinumLabs

Flagship Staking- Audit

Prepared by Linum Labs AG

2025-07-02

linumlabs.com



Executive Summary

This document outlines the security review of the Flagship smart contracts. Linum Labs Auditing has identified several vulnerabilities that should be addressed before deploying smart contracts to any live chain. The report outlines these vulnerabilities and advises on how to fix them.

Protocol Summary

Overview

Flagship is an innovative platform creating AI-managed crypto portfolios, aiming for mass adoption with its mobile-first approach. At its core is a "Crypto Brain" that leverages autonomous AI managers, adaptive learning, and real-time data to provide 24/7 crypto portfolio strategizing.

Audit Scope

./src/AgentStaking.sol
./src/AgentStakingFactory.sol
./src/Airdrip.sol
./src/CommunityRewardsTreasury.sol

Audit Results

Summary

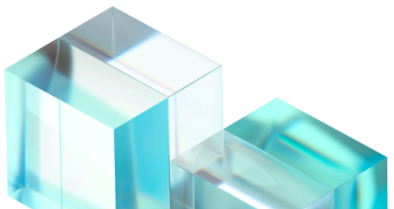
Repository	https://github.com/LinumLabs/flagship-contracts
Commit	55b495f7aab7a0976048cf45452754ec634bd96d
Timeline	June 27th - July 02nd

Issues Found

Bug Severity	Count
Critical	0
High	3
Medium	2
Low	5
Informational	8
Total Findings	18

Summary of Issues

Description	Severity	Status
Double Claiming of Rewards.	High	Resolved
Inaccurate Epoch Principal for Reward Calculation.	High	Resolved
Rewards Stop Accruing After Lock Period Ends.	High	Resolved
Staking Allowed for Inactive Tiers.	Medium	Resolved



Manual APY, Boost Rate, and Pro-Rata Update Mechanism.	Medium	Resolved
Inability to Reactivate Disabled Staking Tiers.	Low	Resolved
Potential Out-of-Gas in emergencyStop().	Low	Resolved
initialClaim Dust Amount Prevents claimWeeklyTokens.	Low	Resolved
Insufficient Balance Check in distributeRewards.	Low	Resolved
Inaccurate Epoch Total Staked and Inefficient Stake Removal.	Low	Resolved
Unexposed Pausability in Factory Contract.	Informational	Resolved
Redundant Merkle Root Check.	Informational	Resolved
Manual APY Update Mechanism.	Informational	Resolved
revokeAirdripSchedule Reclaims to Owner.	Informational	Resolved
Use of memory for Structs in View Functions.	Informational	Resolved
Unlocked Solidity Version Pragma.	Informational	Resolved
Unused OpenZeppelin Imports.	Informational	Resolved
Epoch Advancement Gas Limit Constraint Would Cause Issues During Extended Inactivity.	Informational	Acknowledged

Issues

High Severity

1. Double Claiming of Rewards.

Description: Users can potentially claim rewards multiple times for the same stake and epoch by calling different claiming functions. Specifically, double claiming occurs in the following sequences:

- Calling `claimRewards` (which claims all accumulated rewards for a stake) and then subsequently calling `withdraw`.
- Calling `claimEpochReward` (which claims rewards for a specific epoch) and then subsequently calling `withdraw`.

This indicates a lack of a unified and atomic mechanism to mark rewards as "claimed" across all reward distribution functions. The `claimed` flag within the `EpochStake` struct might not be consistently checked or updated by all relevant claiming pathways, including `withdraw`.

Potential Risk: This vulnerability allows malicious or opportunistic users to drain the reward treasury by repeatedly claiming the same earned rewards. This leads to an over-issuance of tokens, depletion of the reward pool, and can ultimately undermine the entire staking system's economic stability and sustainability. It directly impacts the protocol's ability to maintain its reward distribution model.

Suggested Mitigation: Implement a robust, atomic, and unified system for tracking and marking claimed rewards to prevent any form of double claiming.
Flagship: Fixed in PR.

Linum Labs: Verified.

2. Inaccurate Epoch Principal for Reward Calculation.

Description: The `_calculateEpochReward` function is designed to determine how much reward each staker earns for a specific epoch. It does this by taking the total rewards allocated for that epoch and distributing them proportionally based on each staker's share of `epochTotalStaked[_epoch]`. However, the `epochTotalStaked[_epoch]` variable, which should represent the *total principal staked during a given epoch*, is not consistently updated for all epochs. It primarily increases only when new stakes are made or existing stakes are relocked within that specific epoch. This means that for past epochs where no new staking activity occurred, `epochTotalStaked[_epoch]` might be zero or significantly lower than the actual amount of principal that was actively staked and earning rewards during that period. While `advanceEpochIfNeeded` attempts to carry forward the total staked amount to the *new* `currentEpoch`, this mechanism doesn't correct the historical `epochTotalStaked` values for *already completed* epochs.

Potential Risk: This inaccuracy leads to **incorrect reward calculations and can prevent stakers from claiming their earned rewards for certain periods.**

Suggested Mitigation: A robust system is needed to ensure `epochTotalStaked[_epoch]` accurately reflects the total active principal for *every* epoch.

Flagship: Fixed in PR.

Linum Labs: Verified

3. Rewards Stop Accruing After Lock Period Ends.

Description: The staking system appears to cease calculating or distributing rewards for stakes once their initial lock-up period, defined by `stakeInfo.endTime`, has expired. This occurs even if the principal amount remains staked within the contract and has not been explicitly withdrawn by the user. The reward accrual logic (e.g., within `_calculateEpochReward` or `_calculateCurrentEpochProRataReward`) seems to implicitly or explicitly limit reward generation to the duration of the initial lock-up period, rather than continuing as long as the principal is present.

Potential Risk: Users are financially disadvantaged as they lose out on expected rewards for continuing to keep their principal staked beyond the initial lock-up period.

Suggested Mitigation: Review and adjust the reward calculation logic to ensure that stakes continue to accrue rewards as long as the principal remains within the staking contract, regardless of whether the initial lock-up period has ended.

Flagship: Fixed in PR.

Linum Labs: Verified.

Medium Severity

4. Staking Allowed for Inactive Tiers.

Description: The `stake` function in `AgentStaking` retrieves staking tier information from the `AgentStakingFactory` using `(uint256 stakeDuration, uint256 boost,) = AgentStakingFactory(factory).stakingTiers(_stakingTierID);`.

However, it only checks `require(boost > 0, "AgentStaking__Invalid boost");` and does **not** check the `active` status of the tier. This means users can stake into tiers that the `AgentStakingFactory` owner has explicitly marked as inactive using `disableStakingTier`.

Potential Risk: Users might stake into tiers that are no longer intended for active participation, leading to confusion, unexpected behavior, or a broken user experience. This could also complicate administrative management if inactive tiers continue to accrue stakes.

Suggested Mitigation: Add a `require` statement in the `stake` function to ensure the selected staking tier is active: `require(active, "AgentStaking__Staking tier is inactive");`.

Flagship: Fixed in PR.

Linum Labs: Verified.

5. Manual APY, Boost Rate, and Pro-Rata Update Mechanism.

Description: The `currentAPY` for the staking pool is updated manually by the contract owner via the `updateApy` function. Similarly, boost rates are updated manually through the `updateStakingTier` function, and pro-rata calculations are based on fixed values or are tied to the manually updated APY. There is no automated or dynamic mechanism (e.g., oracle integration, algorithmic adjustment, or governance vote) to adjust these parameters based on real-time market demands, protocol performance, or other external factors.

Potential Risk: This manual update mechanism introduces several risks:

- **Unfair Advantage/Disadvantage:** If the APY or boost rates are changed manually, stakers who claim their rewards at different times (e.g., before versus after an update) might receive disproportionate amounts, leading to unfairness. Users who delay claiming might be disadvantaged if the APY drops, or advantaged if it rises, creating an uneven playing field.

- **Incorrect Reward Computation on Withdrawal:** When a user withdraws, the `calculateTotalRewards` function aggregates rewards across multiple epochs. If the APY or other parameters changed during these epochs, the current calculation might not accurately reflect the rewards earned under the *historical* rates, potentially leading to incorrect total reward computations.
- **Centralization Risk:** Control over these critical parameters is centralized with the owner, making the system susceptible to human error, malicious updates, or delayed responses to market changes.

Suggested Mitigation: This is a fundamental design choice, but its implications should be fully understood and potentially mitigated.

- **For Fairness:** Ensure that reward calculations for past epochs *always* use the APY and boost rates that were active *during those specific epochs*. This means storing historical APY and boost rates per epoch or per time period. The current `updateApy` only changes `currentAPY`, but `_calculateEpochReward` already attempts to use `currentAPY` for past epochs, which is problematic. A historical record of APY changes is crucial.
- **Consider Automation/Decentralization:** If the goal is a more robust and responsive system, explore integrating with a decentralized oracle for dynamic APY adjustments, or implementing a governance module that allows token holders to vote on parameter changes.
- **Clear Communication:** If manual updates persist, ensure transparent and timely communication to stakers about upcoming changes to APY and other reward parameters.

Flagship: Fixed in PR.

Linum Labs:

6. Precision Loss in Reward Calculations.

Description: The formulas used to calculate rewards in `_calculateEpochReward` and `_calculateCurrentEpochProRataReward` involve multiple multiplications and divisions using integer arithmetic. The `REWARD_PRECISION` constant (set to) is intended to help maintain precision, but it is not consistently applied to scale up intermediate results before divisions occur. This leads to **premature truncation of decimal values** during calculations.

Potential Risk: The precision loss results in inaccurate reward distribution and a financial discrepancy for stakers.

Suggested Mitigation: Apply a consistent fixed-point arithmetic strategy across all reward calculations to preserve precision. Thoroughly review and refactor all reward-related formulas (`_calculateEpochReward`, `_calculateCurrentEpochProRataReward`, `calculateTotalRewards`) to ensure `REWARD_PRECISION` is consistently and correctly applied to maintain maximum precision throughout the entire calculation process.

Flagship:

Linum Labs:

Low Severity

7. Inability to Reactivate Disabled Staking Tiers.

Description: The `disableStakingTier` function allows the owner to set the `active` status of a `Staking_Tiers` entry to `false`. However, the `updateStakingTier` function only allows modification of `duration` and `boost`, but does not provide a mechanism to set `active` back to `true`. This means that once a staking tier is disabled, it cannot be reactivated through any existing function.

Potential Risk: Disabled tiers cannot be reactivated without new deployment or a contract upgrade.

Suggested Mitigation: Add a new function, e.g., `enableStakingTier(uint256 _tierId)`, or modify `updateStakingTier` to include a `bool _active` parameter, allowing the owner to explicitly control the active status of a staking tier.

Flagship: Fixed in PR.

Linum Labs: Verified.

8. Potential Out-of-Gas in `emergencyStop()`.

Description: The `emergencyStop` function iterates through all deployed staking pools using a `for` loop (`for (uint256 i = 1; i <= stakingPoolsCount; i++)`) and performs external calls (`setEmergencyStop`, `pause`/`unpause`) on each. As the `stakingPoolsCount` increases, the gas cost of this function will linearly increase.

Potential Risk: The function may become unusable due to exceeding block gas limits with a large number of staking pools, preventing critical actions.

Suggested Mitigation: Consider a design where `AgentStaking` contracts read a global `emergencyStop` flag from the `AgentStakingFactory` or a dedicated status contract. This would decentralize the "stop" logic and remove the need for the factory to iterate through all pools.

Flagship: Fixed in PR.

Linum Labs: Verified.

9. `initialClaim` Dust Amount Prevents `claimWeeklyTokens`.

Description: The `initialClaim` function calculates `initialClaimAmount` as `(schedule.totalAmount * 1) / 100`. If `schedule.totalAmount` is less than 100, `initialClaimAmount` will be `0` due to integer division. In such a scenario, `schedule.claimedAmount` will remain `0` after a successful `initialClaim` transaction. Subsequently, any attempt to call `claimWeeklyTokens` for this schedule will revert with the error "Airdrip__Must claim initial 1% first" because the check `require(schedule.claimedAmount > 0, "Airdrip__Must claim initial 1% first");` will fail, even though the initial claim process was technically completed.

Potential Risk: Beneficiaries with small `totalAmount` allocations may be unable to claim their weekly rewards, as the system incorrectly perceives their initial claim as not having occurred.

Suggested Mitigation: Add a `require` statement in the `addUsersToAirdrip` function to enforce a minimum `_amount` (e.g., `require(_amount >= 100, "Airdrip__Total amount must be at least 100 for initial claim to be non-zero");`). This would prevent the creation of schedules where the initial claim amount would be zero.

Flagship: Fixed in PR.
Linum Labs: Verified.

10. Insufficient Balance Check in `distributeRewards`.

Description: The `distributeRewards` function's balance check `require(fyiToken.balanceOf(address(this)) >= _amount, "RewardsTreasury__Insufficient balance");` does not account for funds that are earmarked for a pending withdrawal via the `proposeWithdrawal` mechanism. If a withdrawal has been proposed, the `withdrawalTimestamp` and the implicit amount to be withdrawn are set. However, these tokens are still included in the contract's `fyiToken.balanceOf(address(this))`. This means `distributeRewards` could successfully transfer tokens, reducing the actual available balance below the amount required for the pending withdrawal, leading to an potential unexpected revert when `executeWithdrawal` is later called.

Potential Risk: An accounting discrepancy that can lead to operational issues.

Suggested Mitigation: Introduce a state variable (e.g., `uint256 public pendingWithdrawalAmount;`) to track the amount of tokens currently locked by a proposed withdrawal.

Flagship: Fixed in PR.

Linum Labs: Verified.

11. Inaccurate Epoch Total Staked and Inefficient Stake Removal.

Description: The contract's epoch tracking for `epochTotalStaked` and `userEpochStakes` suffers from two issues:

1. **Inflated Historical `epochTotalStaked`:** Withdrawals do not fully remove the stake's contribution from `epochTotalStaked` for all past active epochs, leading to artificially high historical values that can carry forward.

2. **Inefficient userEpochStakes Array Management:** _removeStakeFromEpochTracking does not remove EpochStake entries from arrays, causing them to grow indefinitely.

Potential Risk: These issues can lead to **inaccurate reward distribution** (underestimated rewards for active stakers due to inflated denominators), **increased gas costs** for operations iterating over growing arrays, and potential **Denial of Service (DoS)** if arrays become too large.

Suggested Mitigation: The current method of tracking epochTotalStaked and individual userEpochStakes for every past epoch leads to gas inefficiencies and data bloat, particularly for long-term stakers, and can cause Out-of-Gas errors on withdrawal. A more scalable design should avoid unbounded loops over historical data.

Flagship: Fixed in PR.

Linum Labs: Verified.

Informational Severity

12. Unexposed Pausability in Factory Contract.

Description: The AgentStakingFactory contract inherits PausableUpgradeable, but it does not expose public pause() or unpause() functions for itself. This means the contract's own functions (e.g., createNewStakingPool, addNewStakingTier, setAgentStakingImplementation) cannot be directly paused by the owner using the inherited Pausable functionality. Only the individual AgentStaking pools can be paused via dedicated functions.

Potential Risk: If the intention is for the factory's operations to be always available, this is acceptable. Otherwise, it limits the ability to temporarily halt factory-level operations.

Suggested Mitigation: If the intention is for the factory's own operations to be pausable, add the respective functions and modifiers. Otherwise, consider removing `PausableUpgradeable` from the factory's inheritance to avoid confusion if its functionality is not intended for the factory directly.

Flagship: Fixed in PR.

Linum Labs:

13. Redundant Merkle Root Check.

Description: In the `claimWeeklyTokens` function, the check `require(weeklyMerkleRoots[_week] != bytes32(0), "Airdrip__Merkle root not set for this week");` is redundant. The `setMerkleRoot` function already ensures that `_merkleRoot` is not `bytes32(0)` when setting a root. Furthermore, the `require(_week <= currentWeek, "Airdrip__Week not yet available");` check implies that if `_week` is less than or equal to `currentWeek`, a Merkle root for that week must have already been set (and thus be non-zero) by the `setMerkleRoot` function.

Potential Risk: Minor increase in gas consumption due to an unnecessary check.

Suggested Mitigation: Remove the redundant `require(weeklyMerkleRoots[_week] != bytes32(0), "Airdrip__Merkle root not set for this week");` check.

Flagship: Fixed in PR.

Linum Labs: Verified.

14. `revokeAirdripSchedule` Reclaims to Owner.

Description: The `revokeAirdripSchedule` function transfers any `remainingTokens` back to the `owner()` of the `Airdrip` contract.

Potential Risk: The `revokeAirdripSchedule` function transfers any `remainingTokens` back to the `owner()` of the `Airdrip` contract.

Suggested Mitigation: Confirm that returning reclaimed tokens directly to the `Airdrip` contract owner is the desired behavior. If these funds are intended to be re-allocated or returned to a central treasury, the `safeTransfer` target should be adjusted accordingly (e.g., to `address(communityRewardsTreasury)`).

Flagship: The team will transfer the funds back to the treasury from the owner's wallet should a revoke ever happen.

Linum Labs: Acknowledged.

15. Use of `memory` for Structs in View Functions.

Description: In `getClaimableEpochs`, `_calculateEpochReward`, `_calculateCurrentEpochProRataReward`, `calculateTotalRewards`, `getStakeInfo`, and `getUserStakes`, `StakeInfo storage stakeInfo = userStakes[_staker][_stakeIndex];` is used. This declares a storage pointer to a `StakeInfo` struct. Similarly, `EpochStake` structs (when accessed from `userEpochStakes[_staker][epoch]`) are accessed via storage pointers within loops or multiple times. While this directly references storage, repeatedly accessing fields of these structs can incur multiple `SLOAD` operations.

Potential Risk: Increased Gas Consumption for View Calls.

Suggested Mitigation: For structs that are accessed more than once within a `view` function, consider loading the struct into `memory` once, then operating on the memory copy. This replaces multiple, more expensive `SLOAD` operations with a single initial `SLOAD` (or series of `SLOAD`s for the entire struct) followed by cheaper memory reads.

Flagship: Fixed in PR.

Linum Labs: Verified.

16. Unlocked Solidity Version Pragma.

Description: The Solidity contracts use a floating pragma `pragma solidity ^0.8.23;`. This allows the contract to be compiled with any compiler version from 0.8.23 up to (but not including) 0.9.0. While this offers flexibility, it can lead to unexpected behavior or compilation issues if future compiler versions introduce breaking changes or subtle semantic differences that are not immediately apparent during development.

Potential Risk: Unexpected behavior or vulnerabilities could arise if the contract is compiled with a newer, untested Solidity version that introduces breaking changes not accounted for in the current codebase. This can lead to production issues that are hard to debug.

Suggested Mitigation: It is a best practice to lock the Solidity compiler version to a specific, tested version (e.g., `pragma solidity 0.8.23;`). This ensures that the contract always compiles with the exact same compiler version it was developed and tested against, reducing the risk of unexpected behavior due to compiler upgrades. If future upgrades are necessary, the pragma can be explicitly updated after thorough testing with the new compiler.

Flagship: Fixed in PR.

Linum Labs: Verified.

17. Unused OpenZeppelin Imports.

Description: The `AgentStaking` contract imports `BeaconProxy` and `UpgradeableBeacon` from OpenZeppelin. While `AgentStaking` is designed to be an implementation contract for a `BeaconProxy` managed by `AgentStakingFactory`, `AgentStaking` itself does not directly instantiate or interact with `BeaconProxy` or `UpgradeableBeacon` within its own code. The `implementation()` function correctly reads the beacon address from storage, but the direct imports of `BeaconProxy` and `UpgradeableBeacon` are not strictly necessary for the functionality defined within `AgentStaking`.

Potential Risk: Unnecessary imports can slightly increase compilation time and make the contract code appear more complex than it needs to be.

Suggested Mitigation: Remove the unused import statements for `BeaconProxy` and `UpgradeableBeacon`.

Flagship: Fixed in PR.

Linum Labs: Verified.

18. Epoch Advancement Gas Limit Constraint Would Cause Issues During Extended Inactivity.

Description: The `advanceEpochIfNeeded()` function in the AgentStaking contract implements a gas optimization mechanism by limiting epoch advancement to a maximum of `MAX_EPOCHS_PER_ADVANCE` (50 epochs) per transaction. While this prevents out-of-gas errors, it creates a potential issue during extended periods of contract inactivity.

- **Epoch Lag Accumulation**: The `currentEpoch` state variable falls significantly behind the actual epoch calculated by `getCurrentEpoch()`.
- **Incomplete State Updates**: Critical state variables like `epochTotalStaked` and `epochAPY` are not properly initialized for skipped epochs.
- **Multiple Transaction Requirement**: Users must call `advanceEpochIfNeeded()` multiple times (or trigger it through other functions) to catch up to the current epoch.
- **Reward Calculation Dependencies**: Functions like `_calculateEpochRewardPure()` depend on properly initialized epoch state, which may be missing for intermediate epochs.

Potential Risk:

- **Reward Calculation Inconsistencies**: Epochs that haven't been properly advanced would have uninitialized `epochTotalStaked[epoch]` values, potentially leading to incorrect reward calculations.
- **User Experience Degradation**: Users may experience failed transactions or unexpected behavior when interacting with the contract after extended inactivity.
- **Gas Cost Amplification**: Users might need to perform multiple transactions to advance epochs before being able to stake, withdraw, or claim rewards.

Suggested Mitigation: Establish operational procedures to ensure `advanceEpochIfNeeded()` is called at least once every 30-40 epochs.

Flagship: Epoch would be advanced externally on a weekly basis.

Linum Labs: Acknowledged.

Disclaimer

This report is based on the materials and documentation provided to Linum Labs Auditing for the purpose of conducting a security review, as outlined in the Executive Summary and Files in Scope sections. It's important to note that the results presented in this report may not cover all vulnerabilities. Linum Labs Auditing provides this review and report on an as-is, where-is, and as-available basis. By accessing and/or using this report, along with associated services, products, protocols, platforms, content, and materials, you agree to do so at your own risk. Linum Labs Auditing disclaims any liability associated with this report, its content, and any related services and products, to the fullest extent permitted by law. This includes implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Linum Labs Auditing does not warrant, endorse, guarantee, or assume responsibility for any third-party products or services advertised or offered through this report, its content, or related services and products. Users should exercise caution and use their best judgment when engaging with third-party providers. It's important to clarify that this report, its content, access, and/or usage thereof, including associated services or materials, should not be considered or relied upon as financial, investment, tax, legal, regulatory, or any other form of advice.