Security Assessment & Formal Verification Report

Rewards Controller - Umbrella

January-2025

Prepared for:
Aave DAO

Code developed by:
BORED
GHOSTS
DEVELOPING

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Umbrella	Github	62f3850	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of **Rewards Controller** which is part of the **Umbrella project** using the Certora Prover and manual code review findings. The work was undertaken from **23 January 2025 to 2 March 2025**.

The following contract list is included in our scope:

- RewardsController.sol
- RewardsDistributor.sol
- EmissionMath.sol
- InternalStructs.sol

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, **no bug was discovered**.

Protocol Overview

The contracts under review are part of Umbrella which is an upgraded version of the Aave Safety Module, based on a staking and slashing mechanism.

Particularly the contracts under review makes the Rewards distribution mechanism: An upgraded version of the Aave Rewards Controller, responsible for tracking and allowing claiming of rewards. This contract works alongside the Umbrella StakeTokens to provide rewards to their holders for securing Aave against bad debt. These rewards can be arbitrary erc-20 tokens, without unexpected functionality (ERC777, fee-on-transfer, and others).

Coverage

1. We wrote several rules and invariants, and verified them formally (using Certora's prover). See a detailed description later.
2. The reviewed rewards contracts—RewardsDistributor.sol, RewardsController.sol, EmissionMath.sol, and InternalStructs.sol—collectively implement a robust and modular rewards distribution system for tokenized staking protocols. These contracts facilitate dynamic, multi-reward accounting where emissions are tailored through a carefully designed emission curve, ensuring incentives adjust according to asset liquidity and participation. Leveraging standardized access control via role-based permissions and integrating signature-based claim authorization, the system ensures secure and flexible reward claims while maintaining high precision through consistent scaling to 18 decimals.
3. The system's mechanisms were examined to ensure their integrity during our manual audit. Specific details regarding some of the checks performed and our thought processes are outlined below.
4. **Asset and Reward Configuration:**
    a. **Configure the Same Reward Twice:**
        i. The configuration flow differentiates between first-time initialization and subsequent updates. When a reward is set up via the configureAssetWithRewards(...), it checks in _setUpReward(...) whether the reward is already initialized by calling _isRewardInitialized(...) on its stored configuration. If the reward exists, the contract calls _configureReward(...) to update its parameters instead of reinitializing it. This ensures that the reward is not added twice to the asset's rewards list (the rewardsInfo array), thereby preventing duplicate configuration entries.
            1. Once a reward was added, the distribution end cannot be changed to 0 again, thus ensuring that it will forever be uninitialized.

    b. **Delete a Reward:**
        i. The contracts do not provide any administrative function or mechanism to remove a reward from an asset. Once a reward is initialized (and its details recorded in both the mapping and the rewards list), there is no function that supports deletion or removal. This design choice is intentional to protect users' accrued rewards, as removing a reward would disrupt reward tracking and could potentially harm users who might have pending rewards.

    c. **Delete an Asset:**
        i. Similar to rewards, assets are permanently recorded in the system once initialized. The _configureAsset(...) function either updates an existing asset's parameters or, if the asset is new, adds its address to the persistent assets array. There is no mechanism provided to remove an asset from storage. This immutable registration

ensures that all historical reward data remains accessible and that users' positions are safeguarded even if an asset's reward emission is later disabled.

   d. **Overwrite a Reward:**
      i. While reward parameters can be updated through the _configureReward(...), the critical properties (such as the reward's address and its initial decimals scaling) are set only once during the initialization phase (in _initializeReward(...)). Subsequent calls for an already-initialized reward only update adjustable parameters (like maxEmissionPerSecond and distributionEnd) without replacing the underlying reward identity. This prevents an attacker from "overwriting" a reward to point to a malicious token address.
   e. **Have More Than 8 Rewards:**
      i. The system enforces a strict upper bound on the number of rewards per asset. In the _initializeReward(...) function, there is a requirement that checks if the length of the rewardsInfo array is less than MAX_REWARDS_LENGTH (set to 8). If an attempt is made to add a ninth reward, the transaction will revert with a MaxRewardsLengthReached() error. This hard cap protects the contract from potential storage overflows or unexpected behavior caused by an excessive number of rewards.

5. **Dynamic Emission Calculation:**
   a. **Correct and Continuous Endpoint Configuration:**
      i. The EmissionMath library has been carefully designed so that the boundary conditions—the endpoints of each segment of the piecewise linear emission curve—are correctly configured. This means that when transitioning between the three segments (the steep, early-deposit increase; the linear decrease; and the flat emission), the calculated emission values match at the boundaries. This continuity is achieved by precisely defining the formulas in each segment and ensuring that the output from one function seamlessly becomes the input for the next without any jumps or gaps. As a result, the emission curve remains continuous and predictable across all asset levels.
   b. **Ensuring Maximum Emission is Greater Than Flat Emission:**
      i. The configuration explicitly requires that the maximum emission rate (maxEmissionPerSecond) is always higher than the flat emission rate. The flat emission is set as a percentage (80%) of the maximum emission. This is enforced in the calculation logic, particularly within the linear decrease function, where the emission rate decreases smoothly from the maximum down to the flat level. Any attempt to configure the reward with a maxEmissionPerSecond that does not satisfy this condition would fail the validation checks implemented in EmissionMath. This safeguard ensures that the dynamic emission adjustments always follow the

intended behavior and prevent any misconfiguration that could lead to unexpected reward distributions.

6. **Reward Accrual and Indexing:**
    a. **The index didn't increase despite time passing (precision and rounding down):**
        i. The reward index is computed using a fixed scaling factor (typically 1e18) to maintain high precision. However, because Solidity performs integer arithmetic, minor increases that fall below the minimal representable unit will be rounded down. This behavior is intentional to prevent overcompensation from very small, fractional increments. The design ensures that the accrued rewards only update once the accumulated delta exceeds the precision threshold, thus any rounding-down effects remain within acceptable limits and do not materially affect overall reward distribution.
    b. **The rewards index grew during the disabled time period:**
        i. The system prevents the reward index from accumulating rewards once the distribution period has ended. In the _updateRewardIndex(...), if the current block timestamp exceeds the distributionEnd, the emission rate is set to zero, effectively halting any further growth of the index. Moreover, when the reward distribution is resumed, the state is first updated via a call to _updateData(...), which refreshes the global reward index and, crucially, resets the lastUpdateTimestamp to the current block timestamp. This ensures that the period during which rewards were disabled does not contribute to further index increases. As a result, any gap in active distribution is effectively "paused" in the reward calculations, and the accrual accurately resumes only from the moment the state is updated, thereby safeguarding against unintended reward accumulation.
    c. **Ensuring _updateData(...) is called before every state change to maintain an updated state:**
        i. Before any action that might affect a user's balance or the asset's overall state (such as deposits, withdrawals, or transfers), the contract mandates an update of the reward state via the _updateData(...) function. This function recalculates both the global reward index and individual user accrued rewards, ensuring that the state reflects the most recent data before any change is committed. By doing so, the system guarantees that all reward calculations remain current and accurate, thereby preventing any exploitation that might arise from using stale state data during reward claims.
7. **Claiming Flexibility:**
    a. **Claiming Only by an Authorized User:**
        i. The contracts enforce strict authorization for reward claims. A user can claim rewards directly, but if a third party is to claim on their behalf, that party must be

explicitly authorized through the authorizedClaimers mapping. The modifier onlyAuthorizedClaimer(user) ensures that only addresses approved by the user can execute the on-behalf claim functions. Additionally, signature-based permits further secure this mechanism by requiring a valid cryptographic signature from the user, ensuring that only the intended recipient or their designated representative can trigger a claim.

   b. **Safe Default Admin Claim On-Behalf:**
      i. The design includes a provision where the default admin role—typically held by governance—can claim rewards on behalf of any user. This feature is especially useful for contracts that lack the inherent ability to claim rewards themselves. Despite this elevated privilege, the system is designed with safety in mind: the default admin's authority is tightly controlled by role-based access control, ensuring that only a trusted governance entity can exercise this capability. This safeguard prevents arbitrary or malicious claims while providing a necessary fallback mechanism for non-interactive contracts.
   c. **Prevention of Double-Claiming:**
      i. Once rewards are claimed, the contracts immediately update the user's state by reducing the accrued rewards by the claimed amount. In the _claimSelectedRewards(...), after the reward tokens are transferred from the reward payer to the receiver, the user's accrued rewards for that specific reward are decreased accordingly. This design ensures that any rewards already claimed are deducted from the total, making it impossible to claim the same rewards more than once. Each claim operation only allows the user to withdraw the net new rewards accrued since their last update, effectively preventing double-claiming exploits.

8. **Decimals Scaling and Precision Management:**
   a. The rewards system standardizes all internal arithmetic to 18 decimals to maintain high precision and consistency, regardless of the native decimals of individual tokens. This is achieved through scaling functions (commonly referred to as scaleUp and scaleDown) which convert token amounts to a uniform 18-decimal representation before performing calculations such as reward accrual, index updates, and emission computations. By normalizing values in this way, the contract minimizes rounding errors and precision loss, ensuring that even tokens with fewer than 18 decimals (e.g., 6-decimal tokens) are accurately accounted for. This precision management is crucial to fairly distribute rewards and avoid discrepancies that could be exploited or result from imprecise arithmetic.
9. **Access and Rescue Controls:**
   a. To safeguard the protocol, the contracts incorporate robust role-based access control mechanisms. Critical functions—such as asset configuration, reward parameter updates, and administrative tasks—are restricted to trusted roles like the DEFAULT_ADMIN_ROLE

(assigned to governance) and REWARDS_ADMIN_ROLE. Additionally, the integration of rescue controls via the RescuableBase and RescuableACL modules provides a secure means for the designated rescue guardian or authorized admin to recover tokens that might be accidentally locked or misdirected within the contract. This layered security ensures that only verified and trusted entities can perform sensitive operations, thus mitigating the risk of unauthorized access or malicious attempts to drain funds, while maintaining a clear, auditable control over the contract's administrative functions.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0		
High	0		
Medium	0		
Low	0		
Informational	0		
Total			

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Information

Global assumptions: In all the properties that refer to a specific asset, we limit the number of rewards of the asset to one or two (we specify it under the "Property Assumption).
The message sender(s) is not any of the involved contracts (specifically it is not the RewardController).
Solidity version: solc8.27.
Loop unrolling: For most of the rules it's 2. When it's not the case we specify it in the rule table.
Links: Except for the P-03, a link to the Certora's prover report can be found here and here (the rules) or here (the invariants). (For P-03 see below.)
Additional Contracts: We assume that the reward tokens are standard ERC20, and that the StakeToken is a standard ERC4626.

Formal Verification Properties

In the tables below we specify all the formally verified rules that we wrote for the verification of the RewardsController, and give a detailed description for them.

P-01. Bob can't affect the claimed amount of Alice	
Status: Verified	Property Assumptions: Bob isn't the admin, he hasn't a spending allowance on behalf of Alice, Alice doesn't permit Bob to act on her behalf (also not to claim rewards for her). The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
bob_cant_affect_the_claimed_amount_of_alice	Verified	*If Alice is eligible to claim some amount of money, then any action from Bob's side won't affect that amount.* ***Note****:* *1. We exclude the permit functions, and assume that isClaimerAuthorized(alice, bob)==false.* *2. This holds also for the case where Bob is the admin, except for the function configureAssetWithRewards(..)* .	

P-02. Claimed rewards of a user can't decrease with time	
Status: Verified	Property Assumptions: The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
claimed_rewards_cant_decrease_with_time	Verified	*The claimed reward of a user can only increase as time evolves.* ***Note****: When we combine this rule with the previous one ("bob_cant_affect_the_claimed_amount_of_alice") we conclude that the claimed reward of a user can never be decreased (up to the limitations of the previous rule)*	

P-03. Bob can't prevent Alice from claiming her rewards	
Status: Verified	Property Assumptions: Bob isn't the admin, he hasn't a spending allowance on behalf of Alice, Alice doesn't permit Bob to act on her behalf. The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
3 rules: bob_cant_DOS_alice_to_claim/__claimSelectedRewards/__claimAllRewards	Verified	*If the system has enough money to pay both Alice and Bob, then Bob can't make Alice call to claimAllRewards(..) to revert.* ***Note****:* *We exclude the permit and onBehalfOf functions.* ***Links to the prover runs****:* *Here, here, or here.*	

P-04. Integrity of claimAllRewards	
Status: Verified	Property Assumptions: The checked asset contains a single reward or two rewards.

Rule Name	Status	Description	Rule Assumptions
integrity_of_claimAllRewards__1reward/__2rewards	Verified	*We check the following aspects of the function claimAllRewards:* *- The calculateCurrentUserReward(..) is consistent with the actual claimed amount.* *- The balance of the receiver can't decrease.* *- After claiming the rewards, the calculate-reward is 0.* *- The claimable amounts equals the differences of the balances of the payers.*	

P-05. Integrity of claimSelectedRewards

Status: Verified	Property Assumptions: The checked asset contains a single reward or two rewards. The user passes an array with size up-to 2.

Rule Name	Status	Description	Rule Assumptions
integrity_of_claimSelectedRewards__1reward/rewards	Verified	*We check the following aspects of the function claimAllRewards:* *- The calculateCurrentUserReward(..) is consistent with the actual claimed amount.* *- The balance of the receiver can't decrease.* *- After claiming the rewards, the calculate-reward is 0.* *- The claimable amounts equals the differences of the balances of the payers.* *- Claiming from non-existing reward yields 0 amount.* ***Note**: We allow the caller to pass any array of rewards, with size up to 2. It may contain duplicated rewards, and it may contain non-existing rewards.*	

P-06. claimAllRewards(..) can't revert

Status: Verified	Property Assumptions: The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
claimAllRewards_must_succeed	Verified	*Under the following requirements, a call to claimAllRewards(..) must not revert:* *- The payer has enough balance* *- currentContract is allowed to transfer enough money on behalf of the payer* *- The receiver of the rewards can receive the money (namely, its balance won't overflaw)* *- The timestamp is below 2^32.*	

P-07. The index of a reward can’t decrease	
Status: Verified	Property Assumptions: The checked asset contains a single reward or two rewards.

Rule Name	Status	Description	Rule Assumptions
rewardIndex_can_never_decrease	Verified	*The index of a reward can never be decreased*	

P-08. The index of a user can’t decrease	
Status: Verified	Property Assumptions: The checked asset contains a single reward or two rewards.

Rule Name	Status	Description	Rule Assumptions
userIndex_can_never_decrease	Verified	*The index of a user can never be decreased*	

P-09. The function claimAllRewards(..) must revert if there are not enough rewards	
Status: Verified	Property Assumptions: The checked asset contains a single reward or two rewards.

Rule Name	Status	Description	Rule Assumptions
not_enough_rewards_must_revert	Verified	*Assume that a user calls the function claimAllRewardsd(..). If the balance of the payer is below the amount that should be paid the call must revert*	

P-10. Rewards of a user are monotone in balance	
Status: Verified	Property Assumptions: The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
rewards_are_monotone_in_balance	Verified	*Users with more balance (in the asset) are eligible for more rewards.*	

P-11. Current emission can't exceed max emission	
Status: Verified	Property Assumptions: The checked asset contains a single reward.

Rule Name	Status	Description	Rule Assumptions
current_emission_cant_exceed_max_emission	Verified	*The emission can't exceed that value of maxEmissionPerSecondScaled.*	

P-12. The distributionEnd of an asset != 0	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
distributionEnd_NEQ_0	Verified	*Invariant of the system:* *For every reward that exists in the array assetsData[asset].rewardsInfo* *it holds that: assetsData[asset].data[reward].rewardData.distributionEnd != 0*	

P-13. All the rewards of a given asset are unique	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
all_rewars_are_different	Verified	*Invariant of the system:* *The rewards of an asset are unique.*	

P-14. The field distributionEnd (which appears twice) has the same values	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
same_distributionEnd_values	Verified	*Invariant of the system:* *For every reward that exists in the array assetsData[asset].rewardsInfo, the field distributionEnd (that exists in both structs RewardData and RewardAddrAndDistrEnd) has the same value.*	

P-15. The field lastUpdateTimestamp is always less or equal to the current time	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
lastUpdateTimestamp_LEQ_current_time	Verified	*Invariant of the system:* *The field lastUpdateTimestamp can't exceed current timestamp*	

P-16. The field accrued is 0 for non existing rewards

Status: Verified	Property Assumptions: The size of the array rewardsInfo is at most 2.

Rule Name	Status	Description	Rule Assumptions
accrued_is_0_for_non_existing_reward	Verified	*Invariant of the system:* *For every reward that doesn't exist in the array assetsData[asset].rewardsInfo it holds that assetsData[asset].data[reward][user].accrued == 0*	

P-17. The user's index is 0 for non existing rewards

Status: Verified	Property Assumptions: The size of the array rewardsInfo is at most 2.

Rule Name	Status	Description	Rule Assumptions
userIndex_is_0_for_non_existing_reward	Verified	*Invariant of the system:* *For every reward that doesn't exist in the array assetsData[asset].rewardsInfo it holds that assetsData[asset].data[reward][user].index == 0*	

certora aave

P-18. The field distributionEnd is 0 for non existing rewards

Status: Verified	Property Assumptions: The size of the array rewardsInfo is at most 2.

Rule Name	Status	Description	Rule Assumptions
distributionEnd_is_0_for_non_existing_reward	Verified	*Invariant of the system:* *For every reward that doesn't exist in the array assetsData[asset].rewardsInfo* *it holds that assetsData[asset].data[reward].rewardData.distributionEnd == 0*	

P-19. The index of non existing rewards is zero

Status: Verified	Property Assumptions: The size of the array rewardsInfo is at most 2.

Rule Name	Status	Description	Rule Assumptions
rewardIndex_is_0_for_non_existing_reward	Verified	*Invariant of the system:* *For every reward that doesn't exist in the array assetsData[asset].rewardsInfo* *it holds that assetsData[asset].data[reward].rewardData.index == 0*	

P-20. The index of any user is less or equal to the index of the reward	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
userIndex_LEQ_rewardIndex	Verified	*Invariant of the system:* *The index of any user (for a specific reward) can't exceed the index of the reward.*	

P-21. The field targetLiquidity of an asset is never 0	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
targetLiquidity_NEQ_0	Verified	*Invariant of the system:* *If the asset contains rewards (namely rewardsInfo.length>0), then targetLiquidity!=0*	

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.