Security Assessment & Formal Verification Report

StakeToken - 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	b06e3fd	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of **StakeToken** which is part of the **Umbrella project** using the Certora Prover and manual code review findings. The work was undertaken from **7 January 2025 to 23 January 2025**.

The following contract list is included in our scope:

- UmbrellaStakeToken.sol
- StakeToken.sol
- ERC4626StakeTokenUpgradeable.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**. (Anyhow we have one informational issue that we list below).

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 StakeToken: An upgraded version of the Aave Stake Token, responsible for holding reserve assets. These assets can be slashed to cover deficits. Each StakeToken maintains an exchange rate tied to its underlying assets, which adjusts based on slashing events. The StakeToken is integrated with the reward system via a hook and the RewardsController contract. Additionally, it includes logic for fund withdrawals through a cooldown mechanism. For each market protected by the Umbrella system, one or more StakeTokens will be created.

Coverage

1. We wrote several rules and invariants, and verified them formally (using Certora's prover). See a detailed description later.
2. The reviewed smart contracts—ERC4626Upgradeable.sol, ERC4626StakeTokenUpgradeable.sol, ERC20Upgradeable.sol, and StakeToken.sol—collectively implement a suite of upgradeable, tokenized vaults with integrated staking functionality, slashing mechanisms, and access control. These contracts leverage established ERC standards, specifically ERC20 for token interactions and ERC4626 for tokenized vault operations, ensuring compatibility and adherence to widely accepted protocols.
3. **Deposit and Withdrawal Mechanisms:**
    a. During the audit, we considered the possibility of a reentrancy attack in the _withdraw(...) function within ERC4626StakeTokenUpgradeable.sol. While this function is designed for standard ERC20 tokens, it can be susceptible to reentrancy if the underlying asset is an ERC777 or any other token standard that supports callback functions.
        i. ERC777 Token Callbacks: Unlike typical ERC20 tokens, ERC777 tokens can trigger a hook (tokensReceived) on the recipient's contract when tokens are sent. If _withdraw(...) transfers ERC777 tokens before finalizing _totalAssets amount, a malicious contract could invoke the callback and attempt to re-enter _withdraw(...) (or another sensitive function), potentially leading to double withdrawals or other unintended state changes.
4. **Transfer Mechanism:**
    a. During transfers, the contract's _update(...) logic dynamically adjusts any existing cooldown snapshots to reflect the correct staked amount. If a user transfers more staked tokens than were originally placed in cooldown, the snapshot is updated or reset accordingly. This prevents a situation where a staker could perpetually keep tokens in a "withdrawal window" state, ensuring the cooldown mechanism remains accurate and cannot be exploited by transferring staked tokens around.
5. **Cooldown Mechanism:**
    a. The cooldown mechanism in the StakeToken contract serves as a protective feature that regulates the unstaking process. It enforces a mandatory waiting period (_cooldown) before a user can initiate the withdrawal of their staked assets, followed by an unstakeWindow period during which the actual withdrawal must occur. This mechanism mitigates abrupt large-scale withdrawals, ensuring system stability and preventing potential liquidity crises.
    b. In the _update(...) function, changing the condition from checking cooldownSnapshot.endOfCooldown != 0 to verifying that the cooldown is still valid (e.g.,

cooldownSnapshot.endOfCooldown > block.timestamp) could avoid unnecessary state updates and event emissions, thereby saving gas.

   c. During the audit we took into consideration few possible problems:
      i. The ability to be on unstakeWindow more than the protocol intends.
         1.
      ii. Unstaking outside of the unstakeWindow period.
         1. Not possible - maxRedeem(...) will always return 0 outside the range.

6. **Slashing Mechanism:**
   a. The slashing mechanism includes a safeguard that ensures the vault retains a minimum number of assets (MIN_ASSETS_REMAINING) even after a slash is executed. During our review, we examined potential inflation attacks where drastically reducing assets might allow an attacker to manipulate the share price or mint disproportionately large amounts of shares with a small deposit. However, because the slashing logic correctly updates both _totalAssets and the exchange rate in real time, any reductions remain fully reflected in the value of each share. Consequently, there is no avenue to artificially inflate share balances, and the remaining asset threshold further protects system integrity.
7. **Exchange Rate Management:**
   a. Implemented by _convertToShares(...) and _convertToAssets(...) which encapsulate few solutions for possible edge cases we considered:
      i. Division by 0.
         1. The contract avoids division-by-zero scenarios by adding small offsets
      ii. Rounding errors.
      iii. Inflation attack
         1. Not possible due to the use of _totalAssets which overrides the ERC4626.sol native totalAssets(...). _totalAssets can't be altered without minting/burning shares thus we cannot inflate the value of the shares.
8. **Reward System Integration:**
   a. We reviewed the integration of the RewardsController.sol to ensure that every contract action affecting reward accrual (e.g., deposits, withdrawals, transfers, and slashing) properly triggers the necessary updates. Through our analysis, we confirmed that each place where new rewards could be gained or modified is correctly accounted for, ensuring users always receive accurate and timely reward distributions.
9. **Pausing Mechanism:**
   a. Pauses are only needed in case something breaks or doesn't go according to plan.
   b. Straight forward implementation. Halts all operations including slashing, which was done intentionally.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical			
High			
Medium			
Low			
Informational	1	1	
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.

Formal Verification Properties

In the table below we specify all the formally verified rules that we wrote for the verification of the stakeToken, and give a detailed description for them. A link to the Certora's prover report can be found here (the invariants) or here (the rules).

P-01. correctness of cooldown data

Status: Verified	Property Assumptions: None

Rule Name	Status	Description	Rule Assumptions
cooldown_data_correctness	Verified	*Invariant of the system:* *When the cooldown amount of a user is nonzero, the cooldown has to be triggered.*	

P-02. cooldown amount is not greater than balance

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
cooldown_amount_not_greater_than_balance	Verified	*Invariant of the system:* *No user can have greater cooldown amount than is their balance*	

P-03. calculated balance is less or equal to the real balance

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
calculated_bal_LEQ_real_bal	Verified	*Invariant of the system:* *The virtual accounting (which is totalAssets()) is always less or equal to the real balance of the vault.*	

P-04. integrity of deposit/mint

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
integrity_of_deposit/mint	Verified	*We check that when the deposit(...) or mint(...) functions are called the following holds:* *- The balances of the involved users and currentContract is as expected.* *- The totalAssets() is as expected*	

P-05. integrity of slashing

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
integrity_of_slashing	Verified	*We check that when the slash(...) function is called the following holds:* *- The balances of currentContract and the destination is as expected.* *- The totalAssets() is as expected* *- The slashing amount doesn't exceed get_maxSlashable().*	

P-06. integrity of withdraw/redeem

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
updateintegrity_of_withdraw/redeem	Verified	*We check that when the withdraw(...) or redeem() functions are called the following holds:* *- The balances of the involved users and currentContract is as expected.* *- The totalAssets() is as expected* *- The withdrawal is indeed possible according to the cooldown info* *- The Cooldown amount is indeed updated as expected.* .	

P-07. integrity of transferFrom

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
integrity_of_transferFrom	Verified	*We check that when the transferFrom(...) function is called the following holds:* *- The balances of the involved users and currentContract is as expected.* *- The totalAssets() remains unchanged* *- The cooldown amounts of the sender and the receiver is as expected.*	

P-08. cooldown always updates cooldown info

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
cooldown_always_updates_cooldown_info	Verified	*We check that when after calling to cooldown the relevant information get updated as expected.*	

P-09. front run

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
front_run__balance	Verified	*The balance of user-A can't be badly affected by any operation of user-B.*	

P-10. calling to handleAction	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions
calling_to_handleAction	Verified	*We check that the function handleAction() (of the rewards-controller) is called in the following cases:* *- The balance of a user was changed.* *- totalAssets() was changed.*	

Detailed Findings

ID	Title	Severity	Status
I-01	**Optimizing Cooldown Snapshot Handling in the _update Function**	Informational	

Informational Issues

I-01 Optimizing Cooldown Snapshot Handling in the _update Function

Severity: **Informational**	Impact: **Less efficient gas cost**

Description:
During our audit, we identified that the _update(...) function may inadvertently update cooldown snapshots even after they have expired when transferring staked tokens. Specifically, the current condition if (cooldownSnapshot.endOfCooldown != 0) does not verify whether the cooldown period is still active, leading to the emission of unnecessary events and incurring additional gas costs. Replacing this condition with a check that ensures the cooldown snapshot is still relevant (e.g., if (endOfCooldown+withdrawalWindow > block.timestamp)) could enhance efficiency by preventing redundant updates.

BGD-labs response:

Was fixed at c1533d7aa4ca0fc5cdc60aa20bd7076bcbe10c70.

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.