certora

Security Assessment & Formal Verification Report

aave

Pool v3.3

December-2024

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
v3.3	Github	21c3014	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of **Aave-v3.3** using the Certora Prover and manual code review findings. The work was undertaken from **24 October 2024 to 7 November 2024**.

The following contract list is included in our scope:

- AggregatorInterface.sol
- IEACAggregatorProxy.sol
- BaseParaSwapAdapter.sol
- ParaSwapLiquiditySwapAdapter.sol
- ParaSwapRepayAdapter.sol
- ParaSwapWithdrawSwapAdapter.sol
- StataTokenFactory.sol
- PriceFeedEngine.sol
- PoolInstance.sol
- IPool.sol
- ReserveConfiguration.sol
- BorrowLogic.sol
- ConfiguratorLogic.sol
- Pool.sol
- DataTypes.sol
- ReserveLogic.sol
- LiquidationLogic.sol
- EmissionManager.sol
- IEmissionManager.sol
- RewardsController.sol
- IRewardsController.sol

- RewardsDataTypes.sol
- AaveProtocolDataProvider.sol
- IPoolDataProvider.sol
- WrappedTokenGatewayV3.sol
- IWrappedTokenGatewayV3.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, one bug and one informational issue were discovered - see below.

Protocol Overview

The contracts under review are part of the Aave Protocol and introduce the new v3.3 version. The changes that introduced are:

- Introduction of bad debt accounting, to be compatible with the upcoming Umbrella system.
- Improvement of the liquidations close factor mechanism, and complementary ones.
- Optimisation of ReserveConfiguration bit operations.
- Misc integration changes on Pool, PoolDataProvider, different peripheral contracts.

Coverage

1. We adapted the already existing rules from v3.2 to v3.3, and made sure they all pass.
2. **ParaSwap Adapter** - We have reviewed and verified the modifications to *getReserveAToken*(...), confirming that they align with the intended upgrade and do not introduce any vulnerabilities. This update solely adjusts the information source to the pool, maintaining the same functionality as in the previous adapter version. The pool is considered trustworthy as it operates under strict governance controls. If the pool were to return a malicious or incorrect aToken address, it could lead to unintended consequences, such as transaction reverts or in the worst case asset transfers to a malicious contract. However, new assets and their respective aToken addresses are only added to the pool through governance proposals, ensuring that each address is carefully verified during the asset listing process. Given these security checks, we can confirm that switching to *getReserveAToken*(...) as the source of information is safe and adheres to protocol standards.
3. **StataToken** - the function *createStataTokens*(...) now calls *getReserveAToken*(...) instead of *getReserveData*(...), as only the aToken address is needed for validations and data encoding. The aToken address is verified (non-zero), cached for symbol encoding, and passed to *createDeterministic*(...). For the same reasons as in the *ParaSwap Adapter*, this change is secure;

the aToken address is validated through governance when the asset is listed, ensuring its integrity and safety in this updated approach.

4. **ConfigEngine** - the function _*setPriceFeeds*(...) now uses *AggregatorInterface* instead of the old aggregator interface. This change updates the interface but not the implementation, so there are no concerns regarding *latestAnswer*(...).
5. **IPool.sol** - two events, *DeficitCreated* and *DeficitCovered*, were introduced to signal when bad debt is acknowledged by the system and when it's actually covered. We expect DeficitCreated to be triggered during liquidation when bad debt is discovered, which is confirmed to be emitted in *burnDebtToken*. *DeficitCovered* is intended to be triggered by a new function to eliminate the deficit, and it is indeed emitted in *executeEliminateDeficit*(...). The function *getReserveDataExtended*(...) was removed per specification, potentially causing breaking changes for external integrators. We confirmed that no calls to this function exist within the repo. The new function *eliminateReserveDeficit*(...) was added to handle deficit coverage, and DeficitCovered is correctly emitted at the end of this function. Additionally, *getReserveDeficit*(...) now returns the accrued deficit due to unaddressed bad debt from liquidations. Finally, *getReserveAToken*(...) and *getReserveVariableDebtToken*(...) were added as optimizations, as discussed in the PR.
6. **ReserveConfiguration.sol** - the bit masks were flipped to their complementary values. Previously, masks zeroed out specific bits using *AND* operations with the mask itself for get functions and with the mask's *NOT* for set functions. This pattern is now reversed: get functions *AND* the mask directly, and set functions *AND* the mask's *NOT*. Since all bit operations were logically flipped, this change is functionally equivalent, preserving the original behavior. Given that each flip was verified, we confirm that the change is secure.
7. **BorrowLogic.sol** - *executeRepay*(...) was optimized to first check if the reserve is already set as non-collateral for the user before fetching the user's aToken balance when setting collateral to false. This addition is safe, as it avoids unnecessary balance checks by confirming that if the user's aToken balance is zero and the asset is not set as collateral, no further action is needed to change the collateral setting to false. There is no risk here, as this check does not prevent any inputs from being set to false; it simply optimizes gas usage..
8. **Pool.sol** - The modifications to *Pool.sol* introduce key optimizations and new functionality, all of which were carefully reviewed for alignment with protocol goals and security standards. These updates enhance efficiency while adhering to protocol conventions and ensuring compatibility. The introduction of the *onlyUmbrella*() modifier restricts certain functions to the umbrella role, ensuring these sensitive operations are tightly controlled. Although the umbrella role is currently unset in the address provider, the non-upgradable nature of the pool address provider ensures that a new address can be set securely during deployment. This approach maintains flexibility while preserving protocol integrity.

The removal of the *getReserveDataExtended*(...) function aligns with the specified deprecation plan. While this may cause breaking changes for external integrators, we verified that there are no remaining internal calls to this function within the repository, confirming its safe removal.
Several functions, including *getReserveData*(...), *configureEModeCategory*(...), and *getEModeCategoryData*(...), were optimized to avoid unnecessary memory copies. These changes reduce gas consumption by directly interacting with calldata or storage, ensuring efficiency without altering functionality. For example, *getReserveData*(...) now references storage variables directly, avoiding redundant memory operations.
The *eliminateReserveDeficit*(...) function integrates seamlessly with the *ReserveLogic library's executeEliminateDeficit*(...). It ensures that deficits in reserves can be resolved efficiently. The umbrella contract, as the sole caller, is assumed to have no debt, which is critical for maintaining consistency, particularly with GHO token repayment logic. This safeguard prevents potential under-collateralization issues.
Lastly, the addition of getter functions provides a streamlined way to access specific reserve-related data. Since these getters are purely view functions operating on safe variables like *_reserve*, we concluded that they do not introduce vulnerabilities.

9. **DataTypes.sol** The *_deprecatedStableBorrowRate* field in the *ReserveData* struct has been replaced with the *deficit* field to track accumulated bad debt in the protocol. Stable debt was fully deprecated in version 3.2, so reusing this storage field is safe. No functions in the protocol retrieve the *_deprecatedStableBorrowRate* field. Existing view functions return raw 0 for this field, ensuring consistent behavior.
10. **ReserveLogic Library** - We conducted a thorough review of the modifications to the *ReserveLogic* library, verifying that they align with protocol standards and do not introduce vulnerabilities. The updates enhance functionality while maintaining compatibility with the broader system. Key changes include the addition of deficit management features and optimizations for handling cached reserve data.
The newly added *executeEliminateDeficit*(...) function allows for the resolution of reserve deficits, ensuring that reserves remain balanced and secure. This function requires that the caller (the umbrella contract) holds no debt and that the targeted reserve is active, aligning with protocol safeguards. The implementation accurately updates reserve states and emits the *DeficitCovered* event, ensuring transparency.
The use of *ReserveCache* for optimizing reserve state operations was reviewed to confirm its correctness. The caching mechanism ensures that updates to liquidity and debt indices are based on the latest state of the reserves. For functions like *updateState*(...) and *updateInterestRatesAndVirtualBalance*(...), we verified that cached data accurately reflects the stored state at the time of execution, ensuring consistency and correctness.
We also examined how the library handles virtual accounting for reserves. By checking the *getIsVirtualAccActive* flag, the system dynamically adjusts interest rates and virtual balances,

ensuring that these operations are consistent with reserve configurations. The calculation of indices, such as in *cumulateToLiquidityIndex*(...), was validated to ensure accuracy and proper handling of liquidity growth and debt repayment.
To maintain protocol integrity, the library includes robust event emissions for key actions. Events like *DeficitCreated*, *DeficitCovered*, and *ReserveDataUpdated* provide transparency and traceability for state changes within the system. This ensures that any modifications to the reserve states are auditable.
Overall, the updates to the ReserveLogic library have been implemented securely and efficiently, enhancing reserve management while adhering to protocol standards.

11. **LiquidationLogic Library** - We have thoroughly reviewed the modifications within the LiquidationLogic library and verified that they align with the intended protocol behavior, ensuring no vulnerabilities are introduced. The changes optimize the liquidation process by replacing *userTotalDebt* with *userReserveDebt* for improved specificity and introducing additional safeguards to maintain protocol integrity. The dust prevention mechanism ensures that no leftover collateral or debt falls below the defined threshold, preventing inefficiencies and edge cases that could destabilize user positions.
During the review, we considered potential griefing attacks where a malicious actor might repay small amounts of debt to disrupt a liquidator's transaction by causing it to revert. However, this scenario is mitigated by the fact that front-running liquidators are inherently non-beneficial. A malicious actor attempting to front-run could instead choose to liquidate the position themselves before the targeted liquidator does, which is always a viable and more impactful action. Given this context, the griefing attack lacks practical utility and does not pose a significant risk to the system.
The protocol logic incorporates robust checks to validate user health factors, collateral and debt ratios, and ensure compliance with protocol-defined thresholds. While the liquidation logic operates under complex scenarios, such as isolation mode and eMode categories, the implementation ensures these rules are respected, maintaining consistency across the system.

Findings Summary

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

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

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	Wrong rate calculation that may lead to insolvency	Medium	Fixed
I-01	Repaying small amounts to cause reverts	Informational	Fixed

Medium Severity Issues

M-01 Wrong rate calculation that may lead to insolvency

Severity: **Medium**	Impact: **Low**

Description:
Before we explain this issue here is some background. We fix an asset and denote the following:
- *totSUP* is *aToken.totalSupply()* (where the *aToken* is of the given asset)
- *accrued* is *_reserves[asset].accruedToTreasury * index* (where index is the liquidity index of that asset)
- *totDEBT* is *variableDebtToken.totalSupply()* (where the *variableDebtToken* is of the given asset)
- *VB* is *_reserves[asset].virtualUnderlyingBalance*

Ideally, for every asset the following is an invariant (up to V3.3):

$$totSUP + accrued == totDEBT + VB \quad (*)$$

Usually (*) doesn't precisely hold, but we know that the difference between the left hand side and the right hand side is very small (when compared to the left/right hand sides themselves). For the rest of this explanation we assume that (*) holds.

This issue is in the **DefaultReserveInterestRateStrategyV2** contract. This contract calculates the variable *supplyUsageRatio* which should represent that ratio between the amount of money that was borrowed from the pool (thus gains interests), and the amount of money that was supplied to the pool (thus is eligible for interests). The variable *supplyUsageRatio* is calculated by *totDEBT/(totDEBT+VB)*. *totDEBT* is obviously "the amount of money that was borrowed", and by (*) *totDEBT+VB* is the same as *totSUP+accrued*, namely it is "the amount of money that was supplied to the pool", so the calculation is as we expect.

In V3.3 the notion of deficit was added and (*) no longer holds, rather the following:

totSUP + accrued == totDEBT + VB + deficit (**)

(Where by deficit we mean *_reserves[asset].deficit*.)

Hence, in V3.3 the value *totDEBT+VB* does not represent the "amount of money that was supplied to the pool" rather is smaller than that. Hence the value of *supplyUsageRatio* will be larger than it should, hence the liquidity-rate will be larger than it should, which will result in the bad situation that the pool gives to the suppliers more interests that they deserve. Ultimately it may result in an insolvency of the pool.

BGD-labs response:

Acknowledged. Fixed in #1fa3d570e5885113e930e0223c8f35f019d1b165

Informational Issues

I-01 Repaying small amounts to cause reverts

Severity: **Informational**	Impact: **None**

Description:

This issue is in the **LiquidationLogic** library. It arises from the way the protocol handles dust thresholds (minimal leftover amounts) for debt and collateral during liquidations. Specifically, when a user (liquidator) attempts to liquidate another user's position, the protocol enforces rules to prevent leaving amounts smaller than MIN_LEFTOVER_BASE in either debt or collateral.

This enforcement introduces a discontinuity where certain liquidation amounts fall into a "forbidden range." If the liquidator specifies a liquidation amount that would result in leaving less than the threshold, the transaction will revert. While this ensures the protocol doesn't retain economically meaningless amounts (dust), it opens a vector for griefing attacks.

A malicious actor can exploit this by repaying small amounts of debt (just enough to push the liquidation into the forbidden range) before the liquidator's transaction is executed. This is a front-running attack where the malicious actor strategically adjusts the liquidated user's position to cause the liquidator's transaction to fail. The liquidator is forced to retry with a different amount, incurring significant gas costs and operational inefficiencies.

BGD-labs response: We consider the possibility of front-running as non feasible, as most liquidators utilize private mem-pools and already must assume priority inclusion (otherwise the liquidation could be frontran itself instead of just front-running to break it).That said, we acknowledge the issue that in some edge case scenarios passing `amount = type(uint256).max` will lead to a non-intential revert. It is important to know that this unintentional revert in situations where the close factor is 50% and the position is caused in a very specific way, namely either:
- when the amount of debt you are trying to liquidate(50% of total debt), will leave reserve debt < min threshold, while not repaying the full reserve debt
- when the amount of debt you are trying to liquidate(50% of total debt), will leave reserve collateral < min threshold, while not taking the full reserve collateral

Solving this problem on the liquidationLogic itself is non trivial in a ges-efficient way.
Therefore we opted to expose a new LiquidationHelper contract which can be used to fetch the maximum available debt to liquidate for any given position.

Disclaimer

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.