Formal Verification Report of Aave Vault

Summary

This document describes the specification and verification of Aave's Vault using the Certora Prover. The work was undertaken from March 29th to June 18 2023. The latest commit reviewed and run through the Certora Prover was 23366cc.

The scope of our verification includes the following contracts:

- ATokenVault.sol
- ATokenVaultV2.sol

The Certora Prover proved the implementation correct with respect to the formal rules written by the Certora team. During verification, the Certora Prover discovered bugs in the code which are listed in the tables below. All issues were promptly addressed. The fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The following section formally defines the high-level specifications of Aave Vault. All the rules are publicly available in the GitHub repository.

Certora also performed an independent manual review of the code between March 6th and March 23rd, allocating two security researchers.

List of Main Issues Discovered

Severity: Medium

Issue:	DoS - incorrect handling when depositing underlying tokens
Description:	When depositing an amount of underlying token that isn't a whole multiplication of the liquidity index to the vault, the contract may reach a dirty state that keeps reverting undesirably on every method that calls `accrueYield()`. This occurs due to an inaccurate increment of `lastVaultBalance` that doesn't correspond to the actual increment or decrement in the vault's assets following

Issue:	DoS - incorrect handling when depositing underlying tokens
	`deposit()` or `mint()`. In such cases, `lastVaultBalance` ends up being greater than `ATOKEN.balanceof(vault)`, which causes a revert when the new yield is being calculated. The state can be corrected by increasing the existing assets relative to `lastVaultBalance`. This may occur naturally due to the fact that the token accrue yield from the pool, but it can also be initiated by sending a gift to the contract.
Example:	**initial state:** Consider a valid state where `index = 10 Ray, totalSupply = 100 Ray, totalAsset = 101Ray`. **Action 1:** A user deposits a sum of 91 underlying tokens through the contract. The first action invoked is accrual, which updates the state to `lastVaultBalance = _AToken.balanceOf(vault), lastUpdated = now`. **Action 2:** The amount of shares that the assets are worth is calculated through `previewDeposit()`. The simulation returns `shares = (assets * totalSupply)/totalAssets ~= 90.099`, but after rounding down, the result will be `shares = 90`. **Action 3:** From the amount of shares, the number of assets supplied to the pool is recalculated with `_convertToAssets` rounding **up**. The result of this calculation will be `assets = (shares * totalAssets)/totalSupply = 90.9 = 91`. **Action 4:** The contract sends the 91 assets to the pool, which will mint 9 aToken with a worth of 90 underlying tokens. However, `lastVaultBalance` will be incremented by the full 91 assets that were passed to the function. **Post State:** At this point, the state of the contract is `index = 10 Ray, totalSupply = 190 Ray, totalAsset = 101 + 90 = 191Ray, lastVaultBalance = 101 + 91 = 192` In this state, any call to `accrueYield()` will perform the calculation `newYield = newVaultBalance - _s.lastVaultBalance` which will immediately revert due to underflow.
Mitigation/Fix:	Fixed in PR#70, merged in commit 32edfe6.

Severity: Medium

Issue:	Grifting - an attacker can prevent other users from withdrawing for a duration of a block
Description:	An attacker can prevent other users from withdrawing part of their funds, or otherwise force revert by gifting assets to the vault. A gift can be given by directly transferring tokens to the vault at a block where `accrueYield()` is called. By doing this, the malicious player takes `lastVaultBalance` out of sync with `_AToken.balanceof(vault)`. At this stage, the share-to-asset ratio

Issue:	Grifting - an attacker can prevent other users from withdrawing for a duration of a block
	used to determine the amount of assets a user deserves for redeeming their shares is using `_AToken.balanceof(vault)`. However, upon `redeem()`, the withdrawn amount is deducted from `lastVaultBalance`. This mismatch in balance values may lead to reverting cases when the victim tries to withdraw an amount greater than the recorded `lastVaultBalance`. The amount of money that the attacker needs to gift the system in order to execute the attack successfully is determined by the following formula (it does not take rounding into account): `giftAmount > totalAsset(t0) * [(totalShares(t0)/BobSharesToRedeem) - 1]` [1] where `totalAssets(t0)` is the `_AToken.balanceOf(vault)` before the gift, `totalShares(t0)` is the amount of shares in the vault before the gift, `BobSharesToRedeem` is the amount of shares the victim desires to redeem, and `giftAmount` is the amount of assets needed to be gifted to the system by the attacker. Simply put, the gift is proportional to the total amount of reserves in the vault prior to the gift. A simple assignment shows that even for a victim that holds a significant share of the pool which is 50%, the amount needing to be gifted is greater than the total reserves that the pool backs up. **Note:** This is only valid for the same block the gift was transferred. In the next accrual, `lastVaultBalance` is synced, and the user can withdraw their funds.
Example:	**initial state**: Consider the valid state `totalAssets = 200, totalShares = 200`, where the entire 200 shares belong to a victim. **Step 1**: An honest user deposits 1 asset, which grants them 1 share. This brings the state of the contract to `totalAssets = 201, totalShares = 201, lastVaultBalance = 201, lastUpdated = now` **Step 2**: The attacker gifts 3 assets to the vault and takes `lastVaultBalance` out of sync with the `_AToken.balanceOf(vault)`. This brings the state of the contract to `totalAssets = 204, totalShares = 201, lastVaultBalance = 201`. **Post State**: If the victim tries to redeem all their shares, the share-to-asset ratio will evaluate their 200 shares as `200 * 204/201 = 202` assets. When the code gets to the point where it updates `lastVaultBalance`, the function will revert due to underflow: `lastVaultBalance = 201 - 202`.
Mitigation/Fix:	Fixed in PR#82 merged in commit 385b397.

The following 3 issues are derived from the same sequence of initial states and transactions and are a result of the same vulnerability.

Severity: Medium

Issue:	Insolvency - lack of reserves to backup the vault's shares
Rules Broken:	property #14 - `getClaimableFees_LEQ_ATokenBalance` property #15 - `positiveSupply_imply_positiveAssets`
Description:	Given some initial state, an attacker can cause the protocol to deserve fees amounting to a larger value than its reserves. This means a state of insolvency. The coordinated transaction sequence described below relies on two things: 1. An optimization in the code that omits computation of the fees owed by protocol if such computation was already performed in the same block. 2. Although the vault uses the `_AToken.balanceOf[vault]` to track its reserves, it allows users to bypass the `lastVaultBalance` update (accrual) when gifting money to the protocol. The amount of money that the attacker needs to gift the system in order to execute the attack successfully is determined by the following formula (it does not take rounding into account): `(totalAssets(t0) - withdrawnAmount)/feePercentage <= giftAmount` [2] where `totalAssets(t0)` is the `_AToken.balanceOf(vault)` before the gift, `withdrawnAmount` is the amount of assets being withdrawn by the attacker to cause the insolvency, `feePercentage` is the is the fee percentage charges by the vault, and `giftAmount` is the amount of assets needed to be gifted to the system by the attacker. During this griefing attack, the attacker loses a sum of: `giftAmount * (1 - withdrawn_amount/totalAssets(t0))`. We can immediately see that the max loss to the attacker is the gift amount.[2]
Example:	The following scenario assumes that a 5% fee is deducted by the vault. **initial state**: Consider the valid state `_accumulatedFees = 700, _AToken.balanceOf(vault) = 1700, lastVaultBalance = 1700, totalSupply = 1000`. From the definition, `totalAssets() = 1700 - 700 = 1000`, the ratio of share-to-asset is `1:1`. **Step 1**: An attacker is gifting the vault `620` assets by a call to `withdrawATokens()` with the **vault** as the recipient. This updates the state of the contract to be: `_accumulatedFees = 700, _AToken.balanceOf(vault) = 1700, lastVaultBalance = 1080, totalSupply = 380`, `lastUpdated = now`, which implies the share-to-assets ratio is now around `1:2.63`. **Step 2**: On the same block, the attacker, which holds an adequate amount of shares in the pool, withdraws `970` assets by redeeming `~369` shares. Due to optimization on the update block, `totalAssets()` is using a stored, outdated amount of fees deserved by the protocol instead of computing the value using `lastVaultBalance`. The recorded value

Issue:	Insolvency - lack of reserves to backup the vault's shares
	is `totalSupply = 1700 - 700 = 1000`. **Post State:** Following step 2, the state of the contract is now: `_accumulatedFees = 700, _AToken.balanceOf(vault) = 730, lastVaultBalance = 110, totalSupply = ~11`. In the next block (`time > now`), the function `getClaimableFees()` returns `700 + 0.05*(730 - 110) = 731`, while `_AToken.balanceOf(vault) = 730`. A call to `withdrawFees()` with the entire reserve sum (or less) will cause insolvency, meaning shareholders have no assets to backup their shares. **Note:** this broken state is "eternal". Even without withdrawing fees, once `_accrueYield()` is being performed, the state variable `_accumulatedFees` will be updated to the "bad" value, 731.
Mitigation/Fix:	Fixed in PR#82 merged in commit 385b397.

Severity: Medium

Issue:	Complete DoS of the contract due to revert of `totalAssets()`
Rules Broken:	property #14 - `getClaimableFees_LEQ_ATokenBalance`
Description:	Following the scenario described in the issue above, the final state constitutes `getClaimableFees() > _AToken.balanceOf(vault)`, which, as we explained, will remain eternal once an accrual is being performed at one of the next blocks. This state results in a revert of any call to `totalAssets()` due to the underflow of the definition - `ATOKEN.balanceOf(vault) - getClaimableFees()`. This practically DoS the system completely since every function calls `totalAssets()` through `convertToAssets` or `convertToShares`.
Mitigation/Fix:	Fixed in PR#82 merged in commit 385b397.

Severity: Low

Issue:	Misinformation - `previewRedeem()` returns a larger amount of assets than an immediate `redeem()`
Rules Broken:	property #14 - `getClaimableFees_LEQ_ATokenBalance`
Description:	Following the scenario described in the previous issue, if instead of performing Step 2, an honest user calls `previewRedeem()` at the same block, the returned value of the preview will be calculated according to the broken ratio (`1:2.63` shown in the example). If the user calls `redeem()` at the next block, the amount of assets transferred to them will be smaller than expected, according to a lower ratio (`1:2.55` shown in the example). This is because upon

Issue:	**Misinformation - `previewRedeem()` returns a larger amount of assets than an immediate `redeem()`**
	redemption, `_accrueYield()` is called, which causes a reduction in `totalAssets()` by `feePercentage * (ATOKEN.balanceOf(vault) - lastVaultBalance)`.
Mitigation/Fix:	Fixed in PR#82 merged in commit 385b397.

Severity: Low

Issue:	**Loss of fees, overcharge of fees due to rounding**
Rules Broken:	property #13 - `lastVaultBalance_OK`
Description:	The storage variable `_s.lastVaultBalance` marks the portion of reserves for which the vault has already charged fees. In every call to `accrueYield()`, the vault charges fees only from the new yield accrued since the last fee charge - `ATOKEN.balanceOf(Vault) - _s.lastVaultBalance`. Thus, it is expected that after every accrual, `_s.lastVaultBalance` will be equal to `ATOKEN.balanceOf(Vault)`. However, the system may reach a mismatch between the two values when depositing to or withdrawing from the vault due to different update mechanisms. While `_s.lastVaultBalance` is being updated with the exact `assets` amount passed to the function, aToken uses `rayMath` to update the `ATOKEN.balanceOf(Vault)`. While the former is exact, the latter is subject to rounding and may differ from the passed `assets` amount. At the end of a `deposit()` or `withdraw()`, the vault may reach a state where `_s.lastVaultBalance == ATOKEN.balanceOf(Vault) ± 1`. Since this scenario may repeat itself, the vault generally may reach a state where `_s.lastVaultBalance == ATOKEN.balanceOf(Vault) ± k`, where `k` is the number of such occurred deposits or withdraws. For the `+` case, the next time the Vault accrues yield, it will lose its fee from that `k` unaccounted tokens. For the `-` case, the next time that the Vault accrues yield, it will gain money it does not deserve on account of the Vault's users. In some extreme cases, the system may even enter an insolvency similar to the one explained in the `insolvency` bug above.
Mitigation/Fix:	Fixed in PR#86 merged in commit 385b397.

Severity: Low

Issue:	Undesired revert upon depositing aTokens
Description:	When users call `deposit` / `mint`, a check is performed to guarantee that the amount of `assets` they want to deposit does not surpass `maxDeposit()` / `maxMint()`. Both `max` functions are determined by `_maxAssetsSuppliableToAave()`. This function returns one of the following three values: 1. If the market is inactive, frozen or paused, it returns 0. 2. If the market is not limited (no supply cap), no restrictions are imposed and returns `uint256.max`. 3. If the two conditions above aren't met, it returns the remaining margin until reaching the cap. Simply put, `_maxAssetsSuppliableToAave()` returns the maximum amount of **new assets** that can be supplied to the pool, given the market/asset restrictions configured in the pool. Since calling `depositATokens` and `mintWithATokens` only converts aTokens to vault tokens, it does not introduce new money to the pool and hence should not be checked. If the converted aTokens' underlying value surpasses the cap margin, an unjust revert may occur.
Mitigation/Fix:	Fixed in PR#80, merged in commit 34ad6e3.

Severity: Recommendation

Recommendation:	Adding check for `owner ≠ address(0)` in `initialize`
Description:	The function `initialize` does not check that `owner ≠ address(0)`. Allowing the owner to be `address(0)` will result in all `onlyOwner` functions being unreachable, resulting in the transfer of ownership being impossible.
Mitigation/Fix:	Fixed in PR#71, merged in commit 3927afd.

Severity: informational

Issue:	Frontrun - avoiding fee charges for gifts given to the protocol
Description:	The vault intends to charge fees for any yield generated in the Aave pool. This is done by tracking the vault's balance internally at each state-changing method (`lastVaultBalance`) and ensuring that only changes in aTokens' value are accounted for when calculating the fee. However, due to the "same block" optimization that is mentioned in the insolvency issue above, a user can front-run a `withdrawFees()` call and ensure that the vault does not charge fees for any gifts given to the protocol at a block where accrual has already occurred.

Issue:	Frontrun - avoiding fee charges for gifts given to the protocol
Detailed Attack:	1. A user sees that the vault owner wants to withdraw fees. 2. The user invokes an action that will trigger `accrueYield()` and update the state to `lastUpdated = now`. This can be done cheaply by depositing dust in the vault. At this point, there are a few ways that the user can gift money to the protocol, which will not be counted when calculating fees: 3.a. The user can transfer aTokens directly to the vault. 3.b. The user can redeem shares and send the gains directly to the vault. 3.c. The user can gift money directly to the pool by using the `backUnbacked` functionality, for example, and increase the liquidity index. 4. When `withdrawFees()` is invoked, `getClaimableFees()` returns the stored `accumulatedFees` instead of recalculating the fee and taking the new yield generated in this block into account.
Mitigation/Fix:	Fixed in PR#82 merged in commit 385b397.

Severity: Informational

Issue:	Non-compliance of the preview methods with the EIP4626 standard
Rules Broken:	properties: #2 - `previewDeposit_has_NO_threshold` #4 - `previewMint_has_NO_threshold` #6 - `previewWithdraw_has_NO_threshold` #8 - `previewRedeem_has_NO_threshold`
Description:	As per EIP4626, all the preview functions must not take into account any limitation of the system, like those returned by the `max()` methods. In the contract, the preview methods do take into account system limitations. For example let `m` be the value returned by `maxDeposit()`. Then value returned from `previewDeposit(m1)` is identical to the value returned from `previewDeposit(m)` for every `m1>m`.
Mitigation/Fix:	As per EIP4626, all the preview functions may revert due to other conditions that would also cause primary functions to revert. Relying on Aave is acceptable, given that primary functions are impacted by its limitations (e.g. users cannot withdraw if there is no available liquidity in the Aave Pool).

Severity: Informational

Issue:	Non-compliance with EIP4626 standard - non-reverting functions
Rules Broken:	properties: #9 - `must_not_revert` #10 - `must_not_revert_unless_large_input`
Description:	As per EIP4626, the functions `totalAssets`, `maxDeposit`, `maxMint`, `maxWithdraw`, and `maxRedeem` must not revert by any means. In the contract, however, these functions may revert due to over/underflows of arithmetical computations. The EIP also states that the methods `convertToShares()` and `convertToAssets()` must not revert unless due to integer overflow caused by an unreasonably large input. However, these functions may revert even with relatively small inputs due to arithmetical calculations in `totalAssets()`.
Mitigation/Fix:	Although conforming to a standard is important and even essential to a degree, there is likely little to be gained from modifying and altering the core code's functionality to adapt to the minutiae of the standard. As the vault relies inherently on the Aave Protocol, it is acceptable to revert due to Aave-specific calculations.

Disclaimer

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

Though we hope that this information is useful, we do not provide 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.

Overview of Aave Vault

The vault token is a tokenized vault that complies with the EIP-4626 standard. It takes a specific aToken as a reserve asset and mints the corresponding wrapped aTokens to indicate a user's deserved share of the reserves. The contract manages the supply and withdrawal of ERC20 assets in Aave and allows a vault manager to take a fee on yield earned.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- We unroll loops. Violations that require executing a loop more than once will not be detected.
- We do not verify the cryptographic correctness of the WithSig functions. This means that the validity of permits and signatures are not checked.
- A majority of the rules are verified under the assumption that the following values never exceed maxUint64() (the maximal value of a 64-bit unsigned int): - `totalSupply()` - `UNDERLYING.totalSupply()` - `ATOKEN.scaledTotalSupply()`

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit.

✔️ * indicates that the rule is verified on the simplified assumptions described above in "Assumptions and Simplifications Made During Verification".

❌ indicates that the rule was violated under one of the tested versions of the code.

⏱️ indicates the rule is currently timing out and therefore was not proved and no violations were found.

Verification of Aave Vault

EIP4626 Properties

previewDeposit

✔️ 1. previewDeposit_amount_check

- EIP: `previewDeposit()` MUST return as close to, and no more than, the exact amount of Vault shares that would be minted in a `deposit()` call in the same transaction.
- Finding: `previewDeposit()` returns the exact amount of shares minted by `deposit()`.

❌ 2. previewDeposit_has_NO_threshold

- EIP: `previewDeposit()` MUST NOT account for `maxDeposit()` limit or the allowance of asset tokens.

- Finding: We checked that the return value of `previewDeposit()` only depends on `totalSupply()` and `totalAssets()`. The value returned by `previewDeposit()` is NOT independent of `maxDeposit()`.

previewMint

✔ 3. previewMint_amount_check

- EIP: `previewMint()` MUST return as close to, and no more than, the exact amount of Vault shares that would be minted in a `deposit()` call in the same transaction.
- Finding: `previewMint()` returns the same amount of shares that are being minted by `deposit()` with a deviation of up to 1 share in either direction.[3]

❌ 4. previewMint_has_NO_threshold

- EIP: `previewMint()` MUST NOT account for `maxMint()` limit or the allowance of asset tokens.
- Finding: We checked that the return value of `previewMint()` only depends on `totalSupply()` and `totalAssets()`. The value returned by `previewMint()` is NOT independent of `maxMint()`.

previewWithdraw

✔ 5. previewWithdraw_amount_check

- EIP: `previewWithdraw()` MUST return as close to, and no fewer than, the exact amount of Vault shares that would be burned in a `withdraw()` call in the same transaction.
- Finding: `previewWithdraw()` returns the exact amount of shares burnt by `withdraw()`.

❌ 6. previewWithdraw_has_NO_threshold

- EIP: `previewWithdraw()` MUST NOT account for withdrawal limits like those returned from `maxWithdraw()`.
- Finding: We checked that the return value of `previewWithdraw()` only depends on `totalSupply()` and `totalAssets()`. The value returned by `previewWithdraw()` is NOT independent of `maxWithdraw()`.

previewRedeem

✔ 7. previewRedeem_amount_check

- EIP: `previewRedeem()` MUST return as close to, and no more than, the exact amount of assets that would be withdrawn in a `redeem()` call in the same transaction.

- Finding: `previewRedeem()` returns the same amount of assets that are being transferred by `redeem()` with a deviation of up to 1 asset in either direction.[4]

❌ 8. previewRedeem_has_NO_threshold

- EIP: `previewRedeem()` MUST NOT account for redemption limits like those returned from `maxRedeem()`.
- Finding: We checked that the return value of `previewRedeem()` depends only on `totalSupply()` and `totalAssets()`. The value returned by `previewRedeem()` is NOT independent of `maxRedeem()`.

Non-revertable functions

❌ 9. must_not_revert

- EIP: the following functions MUST NOT revert: `assets()`, `totalAssets()`, `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()`.
- Finding:
  - ✔ `asset()` does not revert by any means.
  - ❌ the methods `totalAssets()`, `maxDeposit()`, `maxMint(address)`, `maxWithdraw()`, and `maxRedeem()` may revert due to arithmetic calculations.

❌ 10. must_not_revert_unless_large_input

- EIP: the following functions MUST NOT revert unless there is an integer overflow caused by an unreasonably large input: `convertToShares()` and `convertToAssets()`.
- Finding: the functions may revert even with relatively small inputs. Regardless of the input to the function, a call to these functions may revert due to arithmetical calculations done in the function `totalAssets()`.

Additional Properties

✔ 11. sumAllBalance_eq_totalSupply

The sum of all balances of the Vault equals `totalSupply()`.

✔ 12. balanceOf_leq_totalSupply

Every user's balance is less or equal to `totalSupply()`.

✔ 13. lastVaultBalance_OK

❌ - found issues in previous commits At any time, the storage variable `_s.lastVaultBalance` is either less than or equal to `ATOKEN.balanceOf(Vault)` [5]

✔ 14. getClaimableFees_LEQ_ATokenBalance

❌ - found issues in previous commits The value returned by `getClaimableFees()` must always be less or equal to `ATOKEN.balanceOf(theVault)`.[6]

✔ 15. positiveSupply_imply_positiveAssets

❌ - found issues in previous commits If there exist any vault shares, there must be some non-zero amount of aTokens reserves held by the vault, i.e. `totalSupply()!=0 ⇒ totalAssets()!=0`.[6]

✔ 16. accrueYieldCheck

`_s.accumulatedFees` should monotonically increase with every call to `accrueYield()`.

✔ 17. changeInContractBalanceShouldCauseAccrual

Checks that `_accrueYield()` is called every time a balance change function is executed.

1. The formula was derived by demanding: `lastVaultBalance < BobSharesToRedeem * (totalAsset(t0) + giftAmount) / totalShares(t0)` where `lastVaultBalance`, in the worse case for the attacker, is synced with the `_AToken.balanceOf(vault) := totalAsset(t0)` ↩

2. The formulas are a bound to a couple of restrictions that must be met: a. `withdrawnAmount <= totalAssets(t0)` b. `_AToken.balanceOf(vault) - withdrawnAmount >= giftAmount` ↩ ↩2

3. Verified under the assumption that the function `convertToAssets()` left inverts the function `convertToShares()`. That is: `∀x. convertToAssets(convertToShares(x)) == x`. More specifically, we only need the above assumption for `x == maxDeposit()`. ↩

4. Verified under the assumption that the function `convertToAssets()` left inverts the function `convertToShares()`. That is: `∀x. convertToAssets(convertToShares(x)) == x` More specifically, we only need the above assumption for `x == maxAssetsWithdrawableFromAave()`. ↩

5. Verified under the assumption that the `index` always satisfies `1 ≤ index ≤ 2`. ↩

6. Verified under the following assumptions: a. The `index` always satisfies `1 ≤ index ≤ 2`. b. `∀x ∀ind ∀z. rayMul(rayDiv(x,ind)+z),ind) == x+rayMul(z,ind)` (This equality basically says that: `(x/ind + z) * ind == x + z*ind`, where for the division we use `rayDiv()` and for the multiplication we use `rayMul()`.) c. `∀x ∀ind ∀z.rayMul(rayDiv(x,ind)-z),ind) == x-rayMul(z,ind)`. ↩ ↩2