CERTORA

Formal Verification Report of Aave's Stake Token Rev 3

Summary

This document describes the specification and verification of Aave's Staked Token (Aave Safety Module) v1.5 using the Certora Prover. The work was undertaken from 7th December 2022 to 10th March 2023. The latest commit reviewed and ran through the Certora Prover was 4cf8837 committed on the 1st of March.

The scope of this verification is Aave’s Stake token V3 code which includes the following contracts:

- AaveDistributionManager.sol
- StakedAaveV3.sol
- StakedTokenV2.sol
- StakedTokenV3.sol

The Certora Prover proved the implementation is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora Prover discovered bugs in the code listed in the two tables below. The first table contains two issues, that were also present in the life version of the contract. The second table is a list of all issues found only in the new version of the contract. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The next section formally defines the high-level specifications of Aave’s Stake token V3. All the rules are publicly available in the original repository.

List of Discovered Issues Present in Life Version

Severity: High

Issue:	Indefinite extension of unstake window
Rules Broken:	Property #17 - `cooldownCorrectness` failing here.
Description:	A user located within the unstake window of the staking module can perpetually extend his unstake period such that he will enjoy the rewards of staking (rewards for taking the risk of slashing) without bearing the risk of slashing. This act exploits the delay mechanism of `unstake()` to the user’s own benefit.
Concrete Example:	Step 1 - A user stakes some x tokens (AAVE or ABPT) to the module and, in the worse case, immediately starts his cooldown period. Step 2 - When the user gets into the unstake period, he can unstake and re-stake (or receive a transfer from another user) a certain portion of his balance. When he does so, the contract recalculates the new starting point of the cooldown period (line 488 in `stake()`) using a weighted average formula. This right shift of the starting point effectively extends the user’s remaining unstake window. To optimise this, the user can wait until the last block of his unstake window, then `unstake()` and `stake()` 1/7 of his balance to prolong the unstake period in precisely two days.
Mitigation/Fix:	Fixed on commit 2e9e5a0. A prove can be found here

Severity: Low

Issue:	Deserved rewards of users changed as a result of configureAssets
Rules Broken:	Property #20 - `collectedRewardsMonotonicallyIncrease` failing here.

Description:	When calling `configureAssets()`, the index is updated according to the new parameters passed by the emission manager, which is fully trusted. Since the index is being changed, all the rewards that users collected since their last update are now computed according to the new index. This can manipulate the reward amount that a user has already accrued.
Mitigation/Fix:	Fixed on commit 2e9e5a0. A prove can be found here.

List of Main Issues Discovered in the New Version

Severity: Medium

Issue:	First depositor attack
Rules Broken:	Property #10 - `returnFundsDecreaseExchangeRate` failing here.
Description:	Due to rounding errors, a user can make the pool owe him more money than he deserves and more than the pool holds now. In addition, the user will have the entire alleged sum as voting power for as long as he desires. This exploit can only be executed by the first depositor to an empty pool.
Mitigation/Fix:	Fixed on commit 2e9e5a0. A prove can be found here. The rule still fails and detects another problem described in issue "Exchange rate increased with no costs and other effects."

Severity: Medium

Issue:	Exchange rate can became zero and cause general DoS of the system
Rules Broken:	Property #14 - `exchangeRateNeverZero` failing here.
Description:	When the safety module is in a state where no shares are minted, i.e. `totalSupply = 0`, any user can call `returnFunds` with `amount != 0` to update the exchange rate to 0. Once the exchange rate drops to 0, the protocol is in a DoS state; `previewRedeem` will always revert (dividing by the current exchange rate), and `previewStake` will always return 0 (multiplying by 0). This will block the `Stake`, `Redeem`, `returnFunds` and `Slash` functions.
Mitigation/Fix:	Fixed by limiting `returnFunds` argument, `amount ≥ LOWER_BOUND` in commit 8336dc0. A prove can be found here.

Severity: Low

Issue:	Exchange rate increased by calling `returnFunds`
Rules Broken:	Property #10 - `returnFundsDecreaseExchangeRate` failing here.
Description:	When calling `returnFunds`, it is expected that the exchange rate will be diminished; however, by calling `returnFunds` with `amount = 0`, the exchange rate can be increased. This change in exchange rate effectively reduces the value of each share to a smaller value.
Mitigation/Fix:	Fixed by limiting `returnFunds` argument, `amount ≥ LOWER_BOUND` in commit 8336dc0. A prove can be found here.

Severity: Low

Issue:	Exchange Rate changed by calling `slash` with `amount = 0`.
Rules Broken:	Property #8 - `slashAndReturnFundsOfZeroDoesntChangeExchangeRate` failing here
Description:	When calling `Slash` with `amount = 0`, it is expected that the exchange rate will not be modified; however, due to round up in the `_getExchangeRate` method, the exchange rate can can be altered even by slashing 0 amount.
Mitigation/Fix:	Fixed on commit 4cf8837 by requiring the amount to be slashed non-zero. A prove can be found here.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, 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.

We hope that this information is useful but 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.

Overview of the AAVE Staked Token

The staked token is deployed on Ethereum, with the main utility of participating in the safety module. There are currently two proxy contracts which utilise the staking token - stkAAVE and stkABPT. The new version, the StakedTokenV3, adds enhanced mechanics for slashing in the case of shortfall events. Therefore a new exchange rate is introduced, which will reflect the ratio between stkToken and Token. Initially, this exchange rate will be initialised as 1e18, which reflects a 1:1 peg. In the case of StakedAaveV3, the upgrade adds a hook for managing GHO discounts.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- Our verification assumes that the staked and reward tokens are the same. We used this assumption since this is the system's current state (both are AAVE tokens).
- We assume the reward vault is a contract different from the staked token or any other safety module contract.
- We unroll loops. Violations that require executing a loop more than three times will not be detected.
- We excluded the permit function from our verification by assuming it operates in a valid non-deterministic function.

Notations

✔ By adding this checkmark to a property in the list below we indicate, that the latest commit we reviewed, satisfies this property.

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

✘ indicates the property was violated under one of the tested versions of the code.

indicates that verification of the property is timing out.

Our tool uses Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. This logical structure is reflected in the included formulae for many of the properties below. Note that p and q here can be thought of as analogous to require and assert in Solidity.

The syntax {p} (C1 ~ C2) {q} is a generalisation of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1 ~op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Verification of Example contract

Properties

1. integrityOfStaking ✔

A successful `stake()` function must move an amount of the staking token from the sender to the contract and must increase the sender's share balance accordingly.

```
{
    balanceStakeTokenDepositorBefore := stake_token.balanceOf(msg.sender),
    balanceStakeTokenVaultBefore := stake_token.balanceOf(currentContract),
    balanceBefore := balanceOf(onBehalfOf)
}
    stake(onBehalfOf, amount)
{
    balanceOf(onBehalfOf) = balanceBefore + amount * currentExchangeRate / EXCHANGE_RATE_FACTOR,
    stake_token.balanceOf(msg.sender) = balanceStakeTokenDepositorBefore - amount,
    stake_token.balanceOf(currentContract) = balanceStakeTokenVaultBefore + amount
}
```

3. previewStakeEquivalentStake ✔

Preview stake must return the same shares amount as actual staking.

```
{
    amountOfShares := previewStake(amount),
    receiverBalanceBefore := balanceOf(receiver)
}
    stake(receiver, amount)
{
    amountOfShares = previewStake(amount) - receiverBalanceBefore
}
```

4. noEntryUntilSlashingSettled ✖✔

Users must not be able to stake until slashing is settled (after the post-slashing period).

```
    stake@withrevert(msg.sender, amount)
{
    inPostSlashingPeriod ⇒ stake function reverted
}
```

5. integrityOfSlashing ✔

A successful slash function call must increase the recipient balance by the slashed amount, decrease the vault's balance by the same amount, and turn on the post-slashing period flag.

```
{
    recipientStakeTokenBalanceBefore := stake_token.balanceOf(recipient),
    vaultStakeTokenBalanceBefore := stake_token.balanceOf(currentContract)
}
    slash(recipient, amountToSlash)
{
    stake_token.balanceOf(recipient) = recipientStakeTokenBalanceBefore + amountToSlash,
    stake_token.balanceOf(currentContract) = vaultStakeTokenBalanceBefore - amountToSlash,
    inPostSlashingPeriod = True
}
```

6. noSlashingMoreThanMax ✔

Slashing must not exceed the available slashing amount.

```
{
    vaultBalanceBefore := stake_token.balanceOf(currentContract),
    maxSlashable := vaultBalanceBefore * MaxSlashablePercentage / PERCENTAGE_FACTOR
}
    slash(recipient, amount)
{
    vaultBalanceBefore - stake_token.balanceOf(currentContract) = maxSlashable
}
```

7. slashingIncreaseExchangeRate ✖✔

Slashing must increase the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    slash(args)
{
    getExchangeRate() ≥ ExchangeRateBefore
}
```

8. slashAndReturnFundsOfZeroDoesntChangeExchangeRate ✖✔

Slashing 0 and returning funds of 0 must not affect the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    slash(recipient, 0) || returnFunds(0)
{
    getExchangeRate() = ExchangeRateBefore
}
```

9. integrityOfReturnFunds ✔

A successful `returnFunds()` function call must decrease the sender balance by the returned amount and increase the vault's balance by the same amount.

```
{
    senderStakeTokenBalanceBefore := stake_token.balanceOf(msg.sender),
    vaultStakeTokenBalanceBefore := stake_token.balanceOf(currentContract)
}
    returnFunds(amount)
{
    stake_token.balanceOf(msg.sender) = recipientStakeTokenBalanceBefore - amount,
    stake_token.balanceOf(currentContract) = vaultStakeTokenBalanceBefore + amount
}
```

10. returnFundsDecreaseExchangeRate ✖✔

Returning funds must monotonically decrease the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    returnFunds(args)
{
    getExchangeRate() ≤ ExchangeRateBefore
}
```

11. integrityOfRedeem ✔

A successful redeem function must increase the staked token balance of the recipient by the redeemed amount and must decrease the staked token balance of the contract by the same amount. In addition, the sender's shares balance must decrease by the amount it wanted to redeem.

```
{
    balanceStakeTokenToBefore := stake_token.balanceOf(to),
    balanceStakeTokenVaultBefore := stake_token.balanceOf(currentContract),
    balanceBefore := balanceOf(msg.sender)
}
    redeem(to, amount)
```

```
{
    if (amount > balanceBefore) {
        amountToRedeem := balanceBefore * EXCHANGE_RATE_FACTOR / getExchangeRate();
    } else {
        amountToRedeem := amount * EXCHANGE_RATE_FACTOR / getExchangeRate();
    }

    stake_token.balanceOf(to) = balanceStakeTokenToBefore + amountToRedeem,
    stake_token.balanceOf(currentContract) = balanceStakeTokenVaultBefore - amountToRedeem,
    amount > balanceBefore ⇒ balanceOf(msg.sender) = 0,
    amount ≤ balanceBefore ⇒ balanceOf(msg.sender) = balanceBefore - amount
}
```

12. noRedeemOutOfUnstakeWindow ✔

A successful redeem function call must mean that the user's timestamp is within the unstake window or the safety module is in post-slashing period.

```
{
    cooldown := stakersCooldowns(msg.sender)
}
    redeem(to, amount)
{
    (inPostSlashingPeriod = true) ||
    (block.timestamp > cooldown + getCooldownSeconds() &&
    UNSTAKE_WINDOW ≥ block.timestamp - (cooldown + getCooldownSeconds()))
}
```

13. previewRedeemEquivalentRedeem ✔

`previewRedeem()` must return the same underlying amount as calling redeem.

```
{
    totalUnderlying := previewRedeem(amount),
    receiverBalanceBefore := stake_token.balanceOf(receiver)
}
    redeem(receiver, amount)
{
    totalUnderlying = stake_token.balanceOf(receiver) - receiverBalanceBefore
}
```

14. exchangeRateNeverZero ✖ -- deprecated and deleted

ExchangeRate must never be zero.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    <invoke any method f>
{
    getExchangeRate() ≠ 0
}
```

15. airdropNotMutualized ✔

Transferring tokens to the contract must not change the exchange rate.

```
{
    exchangeRateBefore := getExchangeRate()
}
    stake_token.transfer(currentContract, amount)
{
    getExchangeRate() = exchangeRateBefore
}
```

16. noStakingPostSlashingPeriod ✔

No user should be able to stake while in the post-slashing period.

```
    stake(onBehalfOf, amount)
{
    inPostSlashingPeriod = true ⇒ function reverts
}
```

17. cooldownCorrectness ✖ -- Rule replaced by cooldownDataCorrectness and cooldownAmountNotGreaterThanBalance.

stakersCooldowns must function correctly.

```
{
    windowBefore := stakersCooldowns(msg.sender) + getCooldownSeconds() + UNSTAKE_WINDOW() - block.timestamp
}
    <invoke any method f>
{
    windowAfter := stakersCooldowns(msg.sender) + getCooldownSeconds + UNSTAKE_WINDOW() - block.timestamp,

    (stakersCooldowns(msg.sender) + getCooldownSeconds()) ≤ block.timestamp ⇒ windowBefore ≥ windowAfter,
    (stakersCooldowns(msg.sender) + getCooldownSeconds()) > block.timestamp ⇒ windowBefore ≥ 0
}
```

18. rewardsGetterEquivalentClaim ✔

Rewards getter must return an amount equal to the max rewards the user deserves (if the user were to withdraw max).

```
{
    deservedRewards := getTotalRewardsBalance(from),
    receiverBalanceBefore := reward_token.balanceOf(receiver)
}
    claimedAmount := claimRewardsOnBehalf(from, receiver, max_uint256)
{
    deservedRewards = claimedAmount,
    reward_token.balanceOf(receiver) = receiverBalanceBefore + claimedAmount
}
```

19. rewardsMonotonicallyIncrease ✖✔

Rewards must increase monotonically (except for claim methods).

```
{
    deservedRewardsBefore := getTotalRewardsBalance(user)
}
    <invoke any method f>
{
    deservedRewardsBefore < getTotalRewardsBalance(user) ⇒
        f = claimRewards(address, uint256) ||
        f = claimRewardsOnBehalf(address, address, uint256) ||
        f = claimRewardsAndStake(address, uint256) ||
        f = claimRewardsAndStakeOnBehalf(address, address, uint256) ||
        f = claimRewardsAndRedeem(address, uint256, uint256) ||
        f = claimRewardsAndRedeemOnBehalf(address, address, uint256, uint256)
}
```

20. collectedRewardsMonotonicallyIncrease ✖ removed

21. indexesMonotonicallyIncrease ✔

Global index and personal indexes must increase monotonically.

```
{
    globalIndexBefore := getAssetGlobalIndex(asset),
    personalIndexBefore := getUserPersonalIndex(asset, user)
}
    <invoke any method f>
{
    getAssetGlobalIndex(asset) ≥ globalIndexBefore,
    getUserPersonalIndex(asset, user) ≥ personalIndexBefore
}
```

22. PersonalIndexLessOrEqualGlobalIndex ✔

The personal index of a user on a specific asset must at most be equal to the global index of the same asset. The user's index is derived from the global index and, therefore, must not exceed it.

```
for all asset, user : getUserPersonalIndex(asset, user) ≤ getAssetGlobalIndex(asset)
```

23. totalSupplyGreaterThanUserBalance ✔

The total supply amount of shares must be greater than or equal to any user's share balance.

```
for all user : totalSupply() ≥  balanceOf(user)
```

24. allSharesAreBacked[1] ✗✔

Solvency rule - all shares must be backed. previewRedeem() of all shares must be less than or equal to the balance of the staked-token.

```
previewRedeem(totalSupply()) ≤ stake_token.balanceOf(currentContract)
```

25. totalSupplyDoesNotDropToZero ✔

For non-redeem functions (excluding `redeem`, `redeemOnBehalf`, `claimRewardsAndRedeem`, `claimRewardsAndRedeemOnBehalf`), once the `totalSupply` is positive, it can never become zero again.

```
{
    supplyBefore := totalSupply()
}
    <invoke any non-redeem method>
{
    supplyAfter := totalSupply()
    supplyBefore > 0 ⇒ supplyAfter > 0
}
```

26. rewardsIncreaseForNonClaimFunctions ✔

Rewards monotonically increasing for non claim functions (excluding `claimRewards`, `claimRewardsOnBehalf`, `claimRewardsAndStake`, `claimRewardsAndStakeOnBehalf`, `claimRewardsAndRedeem`, `claimRewardsAndRedeemOnBehalf`).

```
{
    deservedRewardsBefore := getTotalRewardsBalance(user)
}
    <invoke any non-claim method>
{
    deservedRewardsAfter := getTotalRewardsBalance(user)
    deservedRewardsAfter ≥ deservedRewardsBefore
}
```

27. balanceOfZero ✔

The balance of address zero is 0.

```
balanceOf(0) = 0
```

28. cooldownDataCorrectness ✔

When is cooldown amount of user non-zero, the cooldown had to be triggered, thus cooldown timestamp is non-zero.

```
cooldownAmount(user) > 0 => cooldownTimestamp(user) > 0
```

29. cooldownAmountNotGreaterThanBalance ✔

User cannot have greater cooldown amount than is their balance.

```
balanceOf(user) >= cooldownAmount(user)
```

1. Due to computational difficulties, we assumed that `_getRewards(uint256, uint256, uint256)` and `_getAssetIndex(uint256, uint256, uint128, uint256)` return non-deterministic values. This should not effect correctness of this invariant since the invariant is agnostic to the rewards accounting in the system. ↩