yAudit VMEX Review

Review Resources:

A pull request of the feature, and two notion documents explaining the use case and context of the feature were provided.

Auditors:

- Jackson
- HHK

Table of Contents

- Review Summary
- Scope
- Code Evaluation Matrix
- Findings Explanation
- Critical Findings
- High Findings
  - 1. High - `accruedPerToken` Will always be 0 as soon as `totalSupply` > `received`
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
  - 2. High - `_checkNoLiquidity()` doesn't check for staked tokens
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
- Medium Findings
  - 1. Medium - `aToken.getStakedAmount()` return the total of all aTokens with the

same underlying token instead of current aToken
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
- Low Findings
  - 1. Low - Missing security checks on adding stakingReward for an aToken
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
  - 2. Low - Approving `type(uint).max` is not recommended
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
  - 3. Low - No check if previous liquidity when adding a stakingReward
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
- Gas Savings Findings
  - 1. Gas - redundant `stakingExists()` checks
    - Technical Details
    - Impact
    - Recommendation
    - Developer Response
- Informational Findings
  - 1. Informational - Can't add a stakingReward for an aToken that already has

liquidity
        - Technical Details
        - Impact
        - Recommendation
        - Developer Response
    - 2. Informational - Can’t remove or update a stakingReward
        - Technical Details
        - Impact
        - Recommendation
        - Developer Response
        - Yaudit Response
        - Developer Response
- Final remarks

Review Summary

VMEX External Reward Distributor

In VMEX’s own words

> It is common for the protocols issuing “higher-order” assets like vault and pool LP tokens to distribute incentives to participants (LP holders) either through native token emissions or in other denominations. Since some tranches will support lending and collateralization of these asset types, and our protocol accomplishes this by escrowing deposited assets in the lending contract, we’d like a way to avoid opportunity costs for users between depositing their LP on VMEX and earning rewards from the LP protocol. This is similar to a vault strategy, but only used with risk-free incentives contracts that at most purely hold staked LP and don’t interact with any other protocol features.

The contracts of the VMEX External Reward Distributor were reviewed over 7 days. The code review was performed by 2 auditors between May 22, 2023 and May 29, 2023. The pull request was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit fee62be0a579c6da93c296e1f5f3b4569b364251 for the `vmex` repo in the `cyclomancer/external-rewards` branch.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- incentives/
  - ExternalRewardDistributor.sol
  - IncentivesController.sol
- lendingpool/
  - DefaultReserveInterestRateStrategy.sol
  - LendingPoolCollateralManager.sol
- libraries/logic/
  - ValidationLogic.sol
- libraries/types/
  - DistributionTypes.sol
- tokenization/
  - AToken.sol
  - IncentivizedERC20.sol

After the findings were presented to the VMEX team, fixes were made and included in the same PR.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, VMEX and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The appropriate functions are access controlled with the appropriate actors.
Mathematics	Medium	One of the findings in the report is related to a division rounding error.
Complexity	Good	See the Final Remarks section. The auditors felt as though the same technical outcome could have been achieved with a less complex solution.
Libraries	Good	The libraries used in the pull request are appropriate and used correctly.
Decentralization	Low	Similar to the first VMEX report produced by yAudit, there are trusted actors in the system who have a high degree of power.
Code stability	Medium	There were a few minor commits added to the PR after the audit commenced.
Documentation	Good	The documentation supplied was insightful and the comments in the pull request provide useful context.
Monitoring	Average	The core functions of the ExternalRewardDistributor emit events.
Testing and verification	Average	Similar to the first VMEX report produced by yAudit. Increased code coverage is suggested and would have caught some of the issues.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None

High Findings

1. High - `accruedPerToken` Will always be 0 as soon as `totalSupply` > `received`

Current reward accounting uses a division with no precision factor. This makes the result being 0 as soon as the divisor is bigger than the dividend.

Because `totalSupply` is the divisor, if it is bigger than `received` then the result will be 0.

In the case of rewards being claim on every action it is very likely that rewards will always be less than the amount staked.

Technical Details

In ExternalRewardDistributor.sol the function `harvestAndUpdate()` doesn't use any precision factor during the calculation ending up being 0 as soon as `totalSupply` > `received`.

ExternalRewardDistributor:44

```
if (totalSupply > 0) {
        uint256 accruedPerToken = received / totalSupply; //@audit this need a
precision or it will be 0 if totalSupply > received
        rewardData.cumulativeRewardPerToken += accruedPerToken;
        emit Harvested(underlying, accruedPerToken);
}
```

Impact

- Reward will not accumulate properly and will not be claimable by users.

Recommendation

Add a precision factor that will multiply `received` and then divide later when accumulating total rewards.

ExternalRewardDistributor:44

```
if (totalSupply > 0) {
-       uint256 accruedPerToken = received / totalSupply;
+       uint256 accruedPerToken = received * 1e24 / totalSupply;
        rewardData.cumulativeRewardPerToken += accruedPerToken;
        emit Harvested(underlying, accruedPerToken);
}

...

if (rewardData.users[user].lastUpdateRewardPerToken <
rewardData.cumulativeRewardPerToken) {
      uint256 diff =
        rewardData.cumulativeRewardPerToken -
rewardData.users[user].lastUpdateRewardPerToken;
-     rewardData.users[user].rewardBalance += diff *
rewardData.users[user].assetBalance;
+     rewardData.users[user].rewardBalance += diff *
rewardData.users[user].assetBalance / 1e24;
      rewardData.users[user].lastUpdateRewardPerToken =
rewardData.cumulativeRewardPerToken;

      emit UserUpdated(user, msg.sender, underlying,
rewardData.users[user].rewardBalance);
}
```

Developer Response

Fixed in this commit: eb120d8ddd56c0923372af8351cc644fcfa291c9.

2. High - `_checkNoLiquidity()` doesn’t check for staked tokens

The function `_checkNoLiquidity()` is supposed to check for no liquidity on a reserve or else revert.

It achieves it by checking the underlying balance of aToken and `liquidityRate`, both need to be zero to pass the check.

But since the addition of `ExternalRewardDistributor` the underlying balance of aToken can be zero without meaning that the reserve has no liquidity as the underlying tokens might be staked on the stakingReward contract associated with it.

Technical Details

In LendingPoolConfigurator.sol the `_checkNoLiquidity()` doesn’t check the balance of aToken properly.

This function is used to make sure there is no liquidity for a given reserve.

It is being used in `setCollateralEnabledOnReserve()` and `deactivateReserve()`.

LendingPoolConfigurator.sol:508

```
    function _checkNoLiquidity(address asset, uint64 trancheId) internal view {
        DataTypes.ReserveData memory reserveData = pool.getReserveData(
            asset,
            trancheId
        );

        uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(
            reserveData.aTokenAddress
        ); //@audit should also check staked token in external rewarder

        require(
            availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
            Errors.LPC_RESERVE_LIQUIDITY_NOT_0
        );
    }
```

Impact

- Global admin can call `deactivateReserve()` and turn off a reserve even if there is some liquidity in the reserve, making user’s balance stuck in the protocol.
- A tranche admin can call `setCollateralEnabledOnReserve()` and turn off a collateral currently being used by users, reducing their health factor, resulting in liquidation risk, potential defaults on their loans and protocol bad debt. *This is especially an issue as Tranche admins are not supposed to be trusted.*

Recommendation

Like it has been added in other part of the code, consider adding `aToken.getStakedAmount()` to the balance to get the full amount of underlying tokens.

```
function _checkNoLiquidity(address asset, uint64 trancheId) internal view {
        DataTypes.ReserveData memory reserveData = pool.getReserveData(
            asset,
            trancheId
        );

        uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(
            reserveData.aTokenAddress
        );
+       availableLiquidity =
availableLiquidity.add(IAToken(reserveData.aTokenAddress).getStakedAmount())

        require(
            availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
            Errors.LPC_RESERVE_LIQUIDITY_NOT_0
        );
    }
```

Developer Response

Fixed in this commit: 6e502572ac54d83f539aebb1dd1db1288a9a41bb.

Medium Findings

1. Medium - `aToken.getStakedAmount()` return the total of all aTokens with the same underlying token instead of current aToken

`aToken.getStakedAmount()` is used multiple times in the lendingPool to calculate the amount of underlying token being staked in a StakingReward contract by an aToken.

But currently the value returned by the IncentiveController contract returns the total of underlying token being staked and not the one staked by the aToken executing the call.

If multiple tranches accept yUSDC, resulting in multiple aToken having yUSDC as underlying token, then the amount returned by calling `aToken.getStakedAmount()` will be the total of all tranches and not the tranche that the aToken belongs to.

Technical Details

In AToken.sol the function `aToken.getStakedAmount()` calls the IncentiveController function `totalStaked()` which return the total staked for a given underlying token.

AToken.sol:524

```
function getStakedAmount() external view override returns (uint256) {
        IIncentivesController ic = _getIncentivesController();
        if (address(ic) == address(0)) {
            return 0;
        }

        return ic.totalStaked();
}
```

IncentiveController.sol:227

```
function totalStaked() external view override returns (uint256) {
    return _totalStaked();
}
```

The `_totalStaked()`function is inherited from ExternalRewardDistributor.

ExternalRewardDistributor.sol:182

```
function _totalStaked() internal view returns (uint256) {
    if (!stakingExists(msg.sender)) {
      return 0;
    }
    return stakingData[aTokenMap[msg.sender]].staking.balanceOf(address(this));
}
```

Impact

Multiple elements in the lendingPool logic use `aToken.getStakedAmount()` to check the total underlying of a given aToken. This includes:

- `getReserveData()` in AaveProtocolDataProvider.sol which would return the wrong `availableLiquidity`.
- `calculateInterestRates()` in DefaultInterestRateStrategy.sol which doesn't have much impact as long as aToken with stakingRewards are not borrowable.

- `liquidationCall()` in LendingPoolCollateralManager.sol:187 which could cause the call to not revert with the proper error.
- Was suggested to be added to `_checkNoLiquidity()` in LendingPoolConfigurator.sol which could return false even when there do be no liquidity.

On top of that if a third party wanted to check the underlying value of an aToken there would be no way to properly check onchain how many tokens are being staked.

Recommendation

Add a counter for underlyings being staked by each aToken and return it in `getStakedAmount()`.

One easy way would be to add a second `aTokenMap` in the form of an `address` to `uint256` mapping and update it on deposit and withdraw.

ExternalRewardDistributor.sol

```
contract ExternalRewardDistributor is IExternalRewardsDistributor {

  mapping(address => StakingReward) internal stakingData; // incentivized underlying asset => reward info
  mapping(address => address) internal aTokenMap; //  aToken => underlying, authorized callers
+ mapping(address => uint256) public aTokenMapStakedAmount; //  aToken => underlying amount staked, authorized callers

...

function onDeposit(address user, uint256 amount) internal {
  ...
+ aTokenMapStakedAmount[msg.sender] += amount;
}

function onWithdraw(address user, uint256 amount) internal {
  ...
+ aTokenMapStakedAmount[msg.sender] -= amount;
}
```

Then `aToken.getStakedAmount()` could call the getter of this mapping with `address(this)` to get the exact staked amount of itself.

```
function getStakedAmount() external view override returns (uint256) {
  IIncentivesController ic = _getIncentivesController();
  if (address(ic) == address(0)) {
    return 0;
  }

- return ic.totalStaked();
+ return ic.aTokenMapStakedAmount(address(this));
}
```

Developer Response

Fixed in this commit: 457bcfa66ae6b36ff5426a718e3806849d226fd4.

Low Findings

1. Low - Missing security checks on adding stakingReward for an aToken

The VMEX team affirmed that only collateral can be staked on a stakingReward.

Adding a stakingReward can be done only by the manager of the contract which will likely be the team or DAO but it is crucial that aToken added are not borrowable.

If they become borrowable, a borrowing transactions will fail because `validateBorrow()` will revert as `balanceOf(reserve.aTokenAddress)` will be 0 and `releaseUnderlying()` will not be able to send underlying tokens as they are being staked.

But an attacker could send some underlyings to the aToken to make these two functions not revert on borrowing and then borrow which would start creating interest and increase the liquidity index of the reserve.

This will impact the ExternalRewardDistributor that uses scaled balance for its accounting, making new deposits not stake the full amount and withdraw from older deposits revert as the full amount will not be unstaked making some of the users funds stuck in the protocol.

Technical Details

ATokens are supposed to be 1:1 with their underlying asset, to achieve this and allow holders to receive interest rate, the AAVE/VMEX protocol uses a liquidity index and a scaled balance.

When minting and burning tokens the underlying amount is divided by the liquidity index becoming the `scaledBalance`.

Currently the ExternalRewardDistributor.sol uses `scaledBalance` for its accounting which is not an issue as liquidity index starts at 1 thus it's the same as the underlying amount of token.

But in the case of liquidity index increasing, the `scaledBalance` will become less than the underlying amount.

AToken.sol:178

```
function mint(
        address user,
        uint256 amount,
        uint256 index
    ) external override onlyLendingPool returns (bool) {
        approveIncentivesController();

        uint256 previousBalance = super.balanceOf(user);
        uint256 amountScaled = amount.rayDiv(index);
        require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);
        _mint(user, amountScaled); //@audit amount passed to the IncentiveController will be the scaledBalance

        emit Transfer(address(0), user, amount);
        emit Mint(user, amount, index);
        return previousBalance == 0;
}
```

IncentivizedErc20.sol:245

```
function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        uint256 oldTotalSupply = _totalSupply;
        _totalSupply = oldTotalSupply.add(amount);

        uint256 oldAccountBalance = _balances[account];
        _balances[account] = oldAccountBalance.add(amount);

        if (address(_getIncentivesController()) != address(0)) {
            _getIncentivesController().handleAction( //@audit balances passed to the
IncentiveController will be the scaledBalances
                account,
                oldTotalSupply,
                oldAccountBalance,
                _balances[account],
                DistributionTypes.Action.DEPOSIT
            );
        }
}
```

A good explanation of `scaledBalance` vs `balance`can be found on the AAVE documentation: https://docs.aave.com/developers/v/2.0/the-core-protocol/atokens#scaledbalanceof

Impact

- Some funds of the protocol will be stuck in the staking if manager makes a mistake and add a stakingReward for a borrowable token or if turns on borrowing for a collateral that has a stakingReward.

Recommendation

- Add an extra check in `ExternalRewardDistributor.addStakingReward()` to make sure aToken is not borrowable.

```
function addStakingReward(
    address aToken,
    address staking,
    address reward
  ) public onlyManager {
    require(aTokenMap[aToken] == address(0), 'Already registered');

    address underlying = IAToken(aToken).UNDERLYING_ASSET_ADDRESS();

+   DataTypes.ReserveConfigurationMap memory reserveConf =
IAToken(aToken)._pool().getConfiguration(address(aToken),
IAToken(aToken)._tranche());
+   address assetMappings = IAToken(aToken)._addressesProvider().getAssetMappings();
+   require(!ReserveConfiguration.getBorrowingEnabled(reserveConf, underlying,
IAssetMappings(assetMappings)), "aToken is borrowable");

    if (address(stakingData[underlying].reward) == address(0)) {
        require(staking != address(0) && reward != address(0), 'No zero address');
        stakingData[underlying].staking = IStakingRewards(staking);
        IERC20(underlying).approve(staking, type(uint).max);
        stakingData[underlying].reward = IERC20(reward);
    }
    aTokenMap[aToken] = underlying;

    emit RewardConfigured(aToken, underlying, reward, staking);
}
```

- Add an extra check in `LendingPoolConfigurator.setBorrowingOnReserve()` to make sure aToken doesn’t have a stakingReward enabled.

```
function setBorrowingOnReserve(address[] calldata asset, uint64 trancheId, bool[]
calldata borrowingEnabled) external onlyTrancheAdmin(trancheId) {
  for(uint i = 0; i<asset.length;i++){
    require(!borrowingEnabled[i] || assetMappings.getAssetBorrowable(asset[i]),
Errors.LPC_NOT_APPROVED_BORROWABLE);
+   DataTypes.ReserveData memory reserveData = pool.getReserveData(asset[i],
trancheId);

+   if (borrowingEnabled[i]) {
+
require(!IExternalRewardsDistributor(addressesProvider.getIncentivesController()).sta
kingExists(reserveData.aTokenAddress), "Staking exist");
+   }

+    DataTypes.ReserveConfigurationMap memory currentConfig =
reserveData.configuration;
-    DataTypes.ReserveConfigurationMap memory currentConfig =
pool.getConfiguration(asset[i], trancheId);

     currentConfig.setBorrowingEnabled(borrowingEnabled[i]);

     pool.setConfiguration(asset[i], trancheId, currentConfig.data);

     emit BorrowingSetOnReserve(asset[i], trancheId, borrowingEnabled[i]);
  }
}
```

Developer Response

Acknowledged.

2. Low - Approving `type(uint).max` is not recommended

AToken and ExternalRewardDistributor both approve `type(uint).max` over a given contract, one over the incentive controller and one over a staking reward contract.

Infinite approval are not recommended because the contracts approved could become obsolete and changed for the incentive controller and will likely be the case for staking reward contracts in future version of the ExternalRewardDistributor that will add a function to remove a staking reward.

Technical Details

In the ExternalRewardDistributor.sol the function `addStakingReward()` approves the max amount once.

ExternalRewardDistributor.sol:72

```
if (address(stakingData[underlying].reward) == address(0)) {
  require(staking != address(0) && reward != address(0), 'No zero address');
  stakingData[underlying].staking = IStakingRewards(staking);
  IERC20(underlying).approve(staking, type(uint).max); //@audit if allowance reach 0
DOS will happen + in future version if staking is removed/change we still have
infinite approval over it
```

In AToken.sol, the function `approveIncentivesController()` approves the max amount and then approve again but only if `allowance = 0`.

AToken.sol:76

```
function approveIncentivesController() internal {
  address incentivesController = address(_getIncentivesController());
  if (incentivesController != address(0) &&
    IERC20(_underlyingAsset).allowance(address(this), incentivesController) == 0) {
//@audit only approve again if allowance = 0
    IERC20(_underlyingAsset).approve(incentivesController, type(uint).max);
  }
}
```

Impact

- If contracts approved are changed, the allowance would still hold to these obsolete contracts.

Recommendation

Because this contract was described as not needed to be gas optimised and will be used on Optimism, consider using safeApprove and only approve what’s needed on every mint and staking.

Consider also moving `approveIncentivesController()` after amounts get check in each of the 3 mint function to save gas and approve only when needed.

AToken.sol:76

```
- function approveIncentivesController() internal {
-   address incentivesController = address(_getIncentivesController());
-   if (incentivesController != address(0) && IERC20(_underlyingAsset).allowance(address(this), incentivesController) == 0) {
-     IERC20(_underlyingAsset).approve(incentivesController, type(uint).max);
-   }
- }

+ function approveIncentivesController(uint256 amount) internal {
+   IIncentivesController incentivesController = address(_getIncentivesController());
+   if (incentivesController != address(0) && incentivesController.stakingExists(address(this)) {
+     IERC20(_underlyingAsset).safeApprove(incentivesController, 0); //make sure to reset to 0 in case previous approve was not fully consumed
+     IERC20(_underlyingAsset).safeApprove(incentivesController, amount);
+   }
+ }

...

function mint(address user, uint256 amount, uint256 index) external override onlyLendingPool returns (bool) {
- approveIncentivesController();

  uint256 previousBalance = super.balanceOf(user);
  uint256 amountScaled = amount.rayDiv(index);
  require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);
+ approveIncentivesController(amountScaled);
  _mint(user, amountScaled);

...

function mintToTreasury(uint256 amount, uint256 index)
- approveIncentivesController();

  if (amount == 0) {
    return;
  }
+ approveIncentivesController(amount.rayDiv(index));

...

function mintToVMEXTreasury(uint256 amount, uint256 index)
- approveIncentivesController();

  if (amount == 0) {
    return;
  }
+ approveIncentivesController(amount.rayDiv(index));
```

ExternalRewardDistributor.sol:72

```
if (address(stakingData[underlying].reward) == address(0)) {
  require(staking != address(0) && reward != address(0), 'No zero address');
  stakingData[underlying].staking = IStakingRewards(staking);
- IERC20(underlying).approve(staking, type(uint).max);
```

ExternalRewardDistributor.sol:105

```
  IERC20(underlying).transferFrom(msg.sender, address(this), amount);
```

```
+ IStakingRewards staking = rewardData.staking;
+ IERC20(underlying).safeApprove(address(staking), 0); //make sure to reset to 0 in case previous approve was not fully consummed
+ IERC20(underlying).safeApprove(address(staking), amount);
- rewardData.staking.stake(amount);
+ staking.stake(amount);
  rewardData.users[user].assetBalance += amount;
```

Developer Response

Acknowledged.

3. Low - No check if previous liquidity when adding a stakingReward

It has been mentioned in an informational finding that a stakingReward contract cannot be set for an aToken that already has liquidity or an underflow will happen on withdraw and cause loss of funds for users.

Adding a stakingReward can be done only by the manager of the contract which will likely be the team or DAO but it is crucial that aToken added don't have any prior liquidity.

There is currently no check in `addStakingReward()` to make sure it is the case.

Technical Details

In ExternalRewardDistributor.sol the function `onWithdraw()` subtracts `amount` from user's balance and the function `onDeposit()` adds it to user's balance.

Because the user balance start at 0, if the lendingPool was already accepting the underlying token before stakingRewards are turned on then the lendingPool's balance of a user could be above 0 thus when withdrawing from the pool the `amount` passed by the lendingPool to the aToken's `burn()` function could be higher than ExternalRewardDistributor's balance making it revert as an underflow will happen.

In ExternalRewardDistributor.sol:122

```
function onWithdraw(
    address user,
    uint256 amount
```

```
  ) internal {
    if (!stakingExists(msg.sender)) {
      return;
    }
    address underlying = aTokenMap[msg.sender];
    harvestAndUpdate(user, underlying);
    StakingReward storage rewardData = stakingData[underlying];
    rewardData.staking.withdraw(amount); //@follow-up will revert if amount is 0
    IERC20(underlying).transfer(msg.sender, amount);
    rewardData.users[user].assetBalance -= amount; //@audit if a stakingReward is
turned on after user has deposited, an underflow can happen because his assetBalance
is less than what's deposited on vmex
}
```

Impact

- If the manager makes a mistake and add a stakingReward for an aToken that already have liquidity, withdraws will fail and user’s funds will be stuck in the protocol.

Recommendation

Add a check in the function `addStakingReward()` to verify that the aToken has no liquidity.

```
function addStakingReward(
    address aToken,
    address staking,
    address reward
  ) public onlyManager {
    require(aTokenMap[aToken] == address(0), 'Already registered');
+   require(IAToken(aToken).totalSupply() == 0, 'AToken has liquidity');

    address underlying = IAToken(aToken).UNDERLYING_ASSET_ADDRESS();
```

Developer Response

Fixed in this commit: 867a88b272e3e5c8fa3abbe4f1f3cb280f4bfe12.

Gas Savings Findings

1. Gas - redundant `stakingExists()` checks

`stakingExists()` is called but not needed in `onDeposit()`, `onWithdraw()` and `onTransfer()` as `handleAction()` from Incentive Controller already makes the check.

Technical Details

`onDeposit()`, `onWithdraw()` and `onTransfer()` are called only by `handleAction()` from Incentive Controller if the aToken exist in the mapping.

IncentiveController.sol:73

```
if (aTokenMap[msg.sender] != address(0)) {
```

Which correspond to the same thing as `stakingExists()` that is called later on in each function.

```
function stakingExists(address aToken) internal view returns (bool) {
    return aTokenMap[aToken].underlying != address(0);
}
```

Impact

Use extra gas in `onDeposit()`, `onWithdraw()` and `onTransfer()`.

Recommendation

Remove `stakingExists()` check from `onDeposit()`, `onWithdraw()` and `onTransfer()`.

For better readability, update the mapping check in `handleAction()` with a `stakingExists()` call.

Developer Response

Fixed in this commit: f449540d903e583d8aff0eef1cb4b3dba7e76717.

Informational Findings

1. Informational - Can't add a stakingReward for an aToken that already

has liquidity

A stakingReward can be added only for a token that was not being used by any tranches before otherwise when withdrawing from the lendingPool an underflow will happen as the user's balance on the ExternalRewardDistributor might differ from lendingPool's balance.

This is an issue as in the future OP rewards could be turned on for an asset after it has been added to VMEX loosing the opportuniy of rewards while using as collateral/lending which is what this contract was supposed to achieve.

Technical Details

In ExternalRewardDistributor.sol the function onWithdraw() substracts amount from user's balance and the function onDeposit() adds it to user's balance.

Because the user balance start at 0, if the lendingPool was already accepting the underlying token before stakingRewards are turned on then the lendingPool's balance of a user could be above 0 thus when withdrawing from the pool the amount passed by the lendingPool to the aToken's burn() function could be higher than ExternalRewardDistributor's balance making it revert as an underflow will happen.

In ExternalRewardDistributor.sol:122

```
function onWithdraw(
    address user,
    uint256 amount
  ) internal {
    if (!stakingExists(msg.sender)) {
      return;
    }
    address underlying = aTokenMap[msg.sender];
    harvestAndUpdate(user, underlying);
    StakingReward storage rewardData = stakingData[underlying];

    rewardData.staking.withdraw(amount); //@follow-up will revert if amount is 0
    IERC20(underlying).transfer(msg.sender, amount);
    rewardData.users[user].assetBalance -= amount; //@audit if a stakingReward is
turned on after user has deposited, an underflow can happen because his assetBalance
is less than what's deposited on vmex
}
```

Impact

- Can’t add a stakingReward for an aToken that already have liquidity.

Recommendation

Update the design of the contract to make this feature possible.

Developer Response

Acknowledged, won’t fix in current version of the protocol.

2. Informational - Can’t remove or update a stakingReward

There is no function to change or remove a stakingReward, if a stakingReward doesn’t provide rewards anymore but a new contract does it will be impossible to migrate to it and if the rewards just stop then it will be using extra gas by staking and unstaking for nothing.

Technical Details

Feature missing.

Impact

- Extra gas will be used for nothing when rewards stop.
- Some reward opportunities might be missed if can’t move to a new stakingReward contract.

Recommendation

Update the design of the contract to make this feature possible.

Developer Response

Fixed in this commit: 457bcfa66ae6b36ff5426a718e3806849d226fd4.

Yaudit Response

Multiple findings were found in the new features added by this fix:

- `stakingExists()` should be updated to check for `rewardData.rewardEnded` so `_totalStaked()` returns 0 when the stakingReward contract has been removed.

Currently this results in `aToken.getStaked()` returning the amount initially staked even if not staked anymore.

- `updateStakingContract()` doesn’t approve the balance on the new stakingContract before calling `stake()`, this will result in the transaction failing. This would have been catched once test files are updated.
- In `updateStakingContract()` a comment explain why `ERC20.balanceOf()` is used instead of `staking.balanceOf()` but in the case of some of the underlying to be lost, the accounting will be broken as the total of users balances will be more than the total supply. It will result in some users not able to withdraw because not enough underlying and share of rewards distributed per user being higher than what’s available.
- A precision factor will need to be added to `exitStakingContract()`, it should be the same as the one from `harvestAndUpdate()`.
- `totalSupply`need to be stored before calling `exit()` in the function `exitStakingContract()` or the reward will not be taken into account as `totalSupply` will be 0.
- `getUserDataByAToken()` doesn’t take into account if `rewardData.rewardEnded` and will return the initial staked balance of user.

Developer Response

Fixed in multiple commits: dc4c8aaf18b237c2b4cce0004f17cb0a3766c0c5, db1911f6c32e0ebab7bc8512b76025462c149189, f659c89b7b1d94d61a31bc0b5ecf7d575dbfe048, ed25908418bde756b913eec4e2490a6b463737e7, 6c7939e94c504e07bf89703df8e18bf29feac584.

And aknowledged: https://github.com/VMEX-finance/vmex/pull/145#issuecomment-1574215992.

Final remarks

Both auditors agreed that the solution was inflexible and potentially overly complex. Suggestions were made such as:

- Add removal function for stakingReward

- Introduce a migrate function to send all underlyings from aToken to a new stakingReward
- Update the internal balance usage to allow adding a stakingReward even when collateral already used by a tranche

VMEX made changes to support the suggested fixes, and these fixes were reviewed.

Because the solution introduces new risks not present in the original codebase the project is forked from, it is recommended that more heavy testing is done along with a progressive and cautious deployment into production.