CANTINA

Aave v3.1 Competition

Competition

June 2, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Loss of funds (global losses <10% or losses to only a subset of users, but still unacceptable) or core protocol requirements fail to function as intended by the specifications.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Aave is a decentralized non-custodial liquidity protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an overcollateralized (perpetually) or undercollateralized (one-block liquidity) fashion.

From May 10th to May 20th Cantina hosted a competition based on Aave v3.1 Competition. The participants identified a total of **21** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 14

3 Findings

3.1 Medium Risk

3.1.1 `_pendingltv[asset]` will be set to 0 if 2 consecutive calls to `setreservefreeze(asset, true)` are executed

Submitted by r0bert, also found by StErMi, krikolkk, zigtur, lukaprini, Chad0 and 99Crits

Severity: Medium Risk

Context: PoolConfigurator.sol#L228

Description: The function `setReserveFreeze()` was modified in this 3.1 update to set LTV to 0 on freeze, or revert to previous value in case of unfreezing:

```
function setReserveFreeze(
  address asset,
  bool freeze
) external override onlyRiskOrPoolOrEmergencyAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(asset);
  currentConfig.setFrozen(freeze);

  if (freeze) {
    _pendingLtv[asset] = currentConfig.getLtv();
    _isPendingLtvSet[asset] = true;
    currentConfig.setLtv(0);

    emit PendingLtvChanged(asset, currentConfig.getLtv());
  } else if (_isPendingLtvSet[asset]) {
    uint256 ltv = _pendingLtv[asset];
    currentConfig.setLtv(ltv);

    delete _pendingLtv[asset];
    delete _isPendingLtvSet[asset];

    emit PendingLtvRemoved(asset);
  }
```

With the current implementation, if 2 consecutive calls to `setReserveFreeze(asset, true)` are executed, the second one will reset the `_pendingLtv[asset]` to 0. This is enterely possible as the function contains the `onlyRiskOrPoolOrEmergencyAdmins` modifier:

```
function _onlyRiskOrPoolOrEmergencyAdmins() internal view {
  IACLManager aclManager = IACLManager(_addressesProvider.getACLManager());
  require(
    aclManager.isRiskAdmin(msg.sender) ||
      aclManager.isPoolAdmin(msg.sender) ||
      aclManager.isEmergencyAdmin(msg.sender),
    Errors.CALLER_NOT_RISK_OR_POOL_OR_EMERGENCY_ADMIN
  );
}
```

Let's imagine the following scenario:

1. USDC asset experiences significant price fluctuations.
2. The risk admin decides to freeze the asset by calling `setReserveFreeze(USDC, true)`. `_pendingLtv[USDC]` is set to 7700.
3. The emergency admin has had the same thought and also calls `setReserveFreeze(USDC, true)`. `_pendingLtv[USDC]` is set to `currentConfig.getLtv()` which, since the asset is already frozen, is 0.
4. USDC price fluctuations has stopped and the risk admin decides to call `setReserveFreeze(USDC, false)`. USDC LTV is set to 0 as the `_pendingLtv[USDC]` mapping was previously overwritten.

Temporary, as the LTV of USDC is 0, the users can still deposit the asset into the Aave protocol and earn interest on it, but they cannot use it as collateral for borrowing purposes.

Impact: Medium as users can not use the unfrozen asset as collateral for borrowing purposes temporary, until this is noticed by an admin.

Likelihood: Medium, as there are multiple risk admins, pool admins & emergency admins that could call this function.

Recommendation: It is recommended to update the `setReserveFreeze()` function as shown below:

```
function setReserveFreeze(
  address asset,
  bool freeze
) external override onlyRiskOrPoolOrEmergencyAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(asset);
  currentConfig.setFrozen(freeze);

  if (freeze && (currentConfig.getLtv() != 0)) { // <---------
    _pendingLtv[asset] = currentConfig.getLtv();
    _isPendingLtvSet[asset] = true;
    currentConfig.setLtv(0);

    emit PendingLtvChanged(asset, currentConfig.getLtv());
  } else if (_isPendingLtvSet[asset]) {
    uint256 ltv = _pendingLtv[asset];
    currentConfig.setLtv(ltv);

    delete _pendingLtv[asset];
    delete _isPendingLtvSet[asset];

    emit PendingLtvRemoved(asset);
  }
```

3.2 Low Risk

3.2.1 If the asset is frozen, `collateralconfigurationchanged` event will use the wrong `ltv` value

Submitted by StErMi

Severity: Low Risk

Context: PoolConfigurator.sol#L181

Description: If the asset is frozen (`currentConfig.getFrozen() == true`) the config's LTV parameter is not changed. The `ltv` value used in `CollateralConfigurationChanged` is the wrong one in this case, and the one from the config (unchanged) should be used.

Recommendation: If `currentConfig.getFrozen() == true`, `CollateralConfigurationChanged` should use the `LTV` from the config and not the one passed as input.

A better solution would be to update the `CollateralConfigurationChanged` event and pass both the LTV, the one that the admin would like to set but can't because the asset is frozen, and the one that instead will be used by the asset while the asset is in the frozen state.

3.2.2 `setreservefreeze` pass the wrong ltv value to the `pendingltvchanged` event

Submitted by StErMi

Severity: Low Risk

Context: PoolConfigurator.sol#L232

Description: When `setReserveFreeze` is executed and the admin freezes the asset (`freeze == true`) the current LTV is saved into `_pendingLtv[asset]` and the `LTV` of the config is set to `0`. When `PendingLtvChanged` is emitted, the `ltv` parameter of the event is initialized with `currentConfig.getLtv()` that is no longer the "old" LTV but `0`.

Recommendation: Store `currentConfig.getLtv()` into a local variable before updating it to `0` and use such variable to initialize `_pendingLtv[asset]` and pass it to `PendingLtvChanged`

3.2.3 `liquidationgraceperiod` cannot be disabled anymore once configured

Submitted by StErMi

Severity: Low Risk

Context: PoolConfigurator.sol#L264

Description: Once the `liquidationGracePeriod` has been set for the asset, the admin won't be able to disable it completely. Let's assume that `setReservePause(asset, false, 4 hours)` has been called. This means that the asset cannot be liquidated or seized for 4 hours (included). For any security/market reason, the admin needs to reset it to 0 and allow immediately to be liquidated or be seized.

The admin won't be able to disable it completely

1) Calling `setReservePause(asset, false, 0)` will do nothing, leaving the grace period unchanged.
2) Calling `setReservePause(asset, false, 1)` will set the grace period to `block.timestamp + 1` seconds but liquidating or seizing the `asset` will be prevented up to `block.timestamp + 1` **included**, this can't be counted as a complete shutdown of the grace period given that for at least 2 seconds liquidations/seizing cannot be done.

Recommendation: Aave should consider adding a `disableLiquidationGracePeriod` that will set the liquidation grace period in the past:

```
  /// @inheritdoc IPoolConfigurator
  function disableLiquidationGracePeriod(
    address asset,
  ) public override onlyEmergencyOrPoolAdmin {
    // TODO: add requirement that pool is not disabled
    // TODO: add requirement that grace period is indeed enabled

    // set the liquidation grace period in the pass to fully disable it
    uint40 until = uint40(block.timestamp) - 1;
    _pool.setLiquidationGracePeriod(asset, until);

    // TODO: consider emitting a specific event for the grace period disable event. having a timestamp in the
↪   past
    // could be considered confusing
    emit LiquidationGracePeriodChanged(asset, until);
  }
```

3.2.4 Insufficient check in `validateflashloan` still allows to borrow/flashloan more assets than the atoken's `totalsupply`

Submitted by krikolkk, also found by nmirchev8, Bauchibred and r0bert

Severity: Low Risk

Context: ValidationLogic.sol#L473-L474

Description: The function `validateFlashloan` in `ValidationLogic` is used to validate the parameters of flashloans executed by a user. One of the new checks introduced in AAVE 3.1. adds an extra soft protection on borrowing actions, expecting the transaction to revert if the amount borrowed exceeds the total supply of the respective aToken. Although this check will always hold when executing `flashLoanSimple` or `borrow`, it will not have in the case of `flashLoan`.

Since `executeFlashLoan` calls `validateFlashloan`, which calls `validateFlashloanSimple`, where the amount is being checked against the total supply, in a loop, a user can circumvent this check by executing a flashloan where `assets[i] = assets[i+1]`, with `amounts[i] <= totalSupply()` and `amounts[i+1] <= totalSupply()`, but `amounts[i] + amounts[i+1] > totalSupply`. Since `flashLoan` can also be used to borrow, which is done in a loop, it is also possible to borrow more than the total supply, circumventing the introduced check.

Note that the `virtualUnderlyingBalance` of the `asset` must be greater than or equal to `amounts[i] + amounts[i+1]`, and the total supply of the respective aToken must be greater than or equal to `amounts[i] + amounts[i+1]` as well.

Proof of concept: For this test, we will first do a little tweak in the `MockFlashLoanReceiver` contract to handle allowances better:

```
  function executeOperation(
      address[] memory assets,
      uint256[] memory amounts,
      uint256[] memory premiums,
      address, // initiator
      bytes memory // params
    ) public override returns (bool) {
      if (_failExecution) {
        emit ExecutedWithFail(assets, amounts, premiums);
        return !_simulateEOA;
      }

      for (uint256 i = 0; i < assets.length; i++) {
        //mint to this contract the specific amount
        MintableERC20 token = MintableERC20(assets[i]);

        //check the contract has the specified balance
        require(
          amounts[i] <= IERC20(assets[i]).balanceOf(address(this)),
          'Invalid balance for the contract'
        );

        uint256 amountToReturn = (_amountToApprove != 0)
          ? _amountToApprove
          : amounts[i] + premiums[i];
        //execution does not fail - mint tokens and return them to the _destination

        token.mint(premiums[i]);

+       uint allowance = IERC20(assets[i]).allowance(address(this), address(POOL));
+
+       IERC20(assets[i]).approve(address(POOL), allowance + amountToReturn);
-       IERC20(assets[i]).approve(address(POOL), amountToReturn);
      }
```

After that, we can add this test to the Pool.FlashLoans.t.sol test file:

```
function test_flashloan_borrow_moreThanTotalSupply() public {
  vm.prank(carol);
  contracts.poolProxy.withdraw(tokenList.wbtc, 20e8, carol);

  vm.prank(alice);
  contracts.poolProxy.supply(tokenList.wbtc, 0.5e8, alice, 0);

  // first we will make sure the virtual underlying balance and underlying balance are greater than the total
↪  supply
  // so bob will first flashloan 0.5 wbtc, while 0.05 wbtc will go to `accruedToTreasury`

  vm.startPrank(poolAdmin);
  contracts.poolConfiguratorProxy.updateFlashloanPremiumTotal(10_00);
  contracts.poolConfiguratorProxy.updateFlashloanPremiumToProtocol(100_00);
  TestnetERC20(tokenList.wbtc).transferOwnership(address(mockFlashSimpleReceiver));
  vm.stopPrank();

  bytes memory emptyParams;

  vm.prank(bob);
  contracts.poolProxy.flashLoanSimple(
    address(mockFlashSimpleReceiver),
    tokenList.wbtc,
    0.5e8,
    emptyParams,
    0
  );

  // now we will do the flashloan that is supposed to revert
  address[] memory assets = new address[](2);
  uint256[] memory amounts = new uint256[](2);
  uint256[] memory modes = new uint256[](2);

  assets[0] = tokenList.wbtc;
  assets[1] = tokenList.wbtc;
  amounts[0] = 0.5e8;
  amounts[1] = 0.05e8;
```

```
  address aTokenAddress = contracts.poolProxy.getReserveData(tokenList.wbtc).aTokenAddress;
  // make sure we are indeed flashloaing more than the total supply
  assertGt(amounts[0] + amounts[1], IERC20(aTokenAddress).totalSupply());

  // just to showcase the vulnerability we will set the fees to 0
  vm.startPrank(poolAdmin);
  contracts.poolConfiguratorProxy.updateFlashloanPremiumTotal(0);
  contracts.poolConfiguratorProxy.updateFlashloanPremiumToProtocol(0);
  vm.stopPrank();

  vm.prank(address(mockFlashSimpleReceiver));
  TestnetERC20(tokenList.wbtc).transferOwnership(address(mockFlashReceiver));

  vm.prank(alice);
  contracts.poolProxy.flashLoan(
    address(mockFlashReceiver),
    assets,
    amounts,
    modes,
    alice,
    emptyParams,
    0
  );
}
```

The test showcases that it is still possible to perform the action that the v3.1 upgrade should have fixed. Note that with this function it is also possible to borrow the funds flashloaned.

Recommendation: Consider not allowing non-unique assets to be flashloaned, or, in the case of non-unique assets in the flashloan `assets` parameter, compare the sum of such assets against the total supply instead of each amount separately.

3.2.5 Setting emode category params can potentially set emode `ltv` to a lower value than the pending `ltv` of a frozen asset

Submitted by krikolkk, also found by zigtur

Severity: Low Risk

Context: PoolConfigurator.sol#L379

Description: The function `setEModeCateogry` in `PoolConfigurator` updates the EMode params for a specific EMode category. One requirement for these new parameters is that the EMode `ltv` value is greater than the current `ltv` of all the assets within the category. However, the category can contain frozen assets, and, in this case, either purposely or mistakenly set the EMode `ltv` to a value lower than the current `ltv` (which is cached) of the frozen asset within the category.

Proof of concept:

```
function setEModeCategory(
  uint8 categoryId,
  uint16 ltv,
  uint16 liquidationThreshold,
  uint16 liquidationBonus,
  address oracle,
  string calldata label
) external override onlyRiskOrPoolAdmins {
  // ...

  address[] memory reserves = _pool.getReservesList();
  for (uint256 i = 0; i < reserves.length; i++) {
    DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(reserves[i]);
    if (categoryId == currentConfig.getEModeCategory()) {
      require(ltv > currentConfig.getLtv(), Errors.INVALID_EMODE_CATEGORY_PARAMS); // <- current ltv will be 0
↪  on frozen asset, after unfreeze it can be higher than EMode ltv
      require(
        liquidationThreshold > currentConfig.getLiquidationThreshold(),
        Errors.INVALID_EMODE_CATEGORY_PARAMS
      );
    }
  }

  // ...
}
```

Recommendation: Consider adding a check for frozen assets:

```
} else if (\_isPendingLtvSet[asset]) {
uint256 ltv = \_pendingLtv[asset];
```

```
  function setEModeCategory(
    uint8 categoryId,
    uint16 ltv,
    uint16 liquidationThreshold,
    uint16 liquidationBonus,
    address oracle,
    string calldata label
  ) external override onlyRiskOrPoolAdmins {
    require(ltv != 0, Errors.INVALID_EMODE_CATEGORY_PARAMS);
    require(liquidationThreshold != 0, Errors.INVALID_EMODE_CATEGORY_PARAMS);

    // validation of the parameters: the LTV can
    // only be lower or equal than the liquidation threshold
    // (otherwise a loan against the asset would cause instantaneous liquidation)
    require(ltv <= liquidationThreshold, Errors.INVALID_EMODE_CATEGORY_PARAMS);
    require(
      liquidationBonus > PercentageMath.PERCENTAGE_FACTOR,
      Errors.INVALID_EMODE_CATEGORY_PARAMS
    );

    // if threshold * bonus is less than PERCENTAGE_FACTOR, it's guaranteed that at the moment
    // a loan is taken there is enough collateral available to cover the liquidation bonus
    require(
      uint256(liquidationThreshold).percentMul(liquidationBonus) <=
        PercentageMath.PERCENTAGE_FACTOR,
      Errors.INVALID_EMODE_CATEGORY_PARAMS
    );

    address[] memory reserves = _pool.getReservesList();
    for (uint256 i = 0; i < reserves.length; i++) {
      DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(reserves[i]);
      if (categoryId == currentConfig.getEModeCategory()) {
+       uint256 currentLtv = _isPendingLtvSet[reserves[i]] ? _pendingLtv[reserves[i]] : currentConfig.getLtv();
-       require(ltv > currentConfig.getLtv(), Errors.INVALID_EMODE_CATEGORY_PARAMS);
+       require(ltv > currentLtv, Errors.INVALID_EMODE_CATEGORY_PARAMS);
        require(
          liquidationThreshold > currentConfig.getLiquidationThreshold(),
          Errors.INVALID_EMODE_CATEGORY_PARAMS
        );
      }
    }

    _pool.configureEModeCategory(
      categoryId,
```

```
      DataTypes.EModeCategory({
        ltv: ltv,
        liquidationThreshold: liquidationThreshold,
        liquidationBonus: liquidationBonus,
        priceSource: oracle,
        label: label
      })
    );
    emit EModeCategoryAdded(categoryId, ltv, liquidationThreshold, liquidationBonus, oracle, label);
  }
```

3.2.6 State changing function doesn't emit events in `executesetliquidationgraceperiod()`

Submitted by ladboy233

Severity: Low Risk

Context: PoolLogic.sol#L132-L138

Description: State changing function `executeSetLiquidationGracePeriod()` is used to set the `liquidationGracePeriodUntil` to the new timestamp. As the function changes the state, it should emit events for parameters change:

```
  function executeSetLiquidationGracePeriod(
      mapping(address => DataTypes.ReserveData) storage reservesData,
      address asset,
      uint40 until
      ) external {
          reservesData[asset].liquidationGracePeriodUntil = until;
}
```

Impact: Not adding an event will obfuscate off-chain monitoring when changing system parameters.

Recommendation: Ensure these endpoints emit events as some off-chain agents might be monitoring the protocol for these events.

3.3 Informational

3.3.1 `swaptovariable` natspec is incorrect

Submitted by StErMi

Severity: Informational

Context: IPool.sol#L384

Description: The `swapToVariable` function is described as "Permission-less movement of stable positions to variable". From the official docs:

> allowing any address to swap any stable rate user to variable, without changing anything else in the position.

The natspec documentation instead says that the function allows the owner of the debt position to swap **his own** debt from stable to variable.

Recommendation: Update the natspec to document the real behavior and scope of the function.

3.3.2 `executeswapborrowratemode` miss the natspec documentation of the new `user` parameter

Submitted by StErMi

Severity: Informational

Context: BorrowLogic.sol#L317

Description: `executeSwapBorrowRateMode` miss the natspec documentation of the new `user` parameter

Recommendation: Add the natspec docs for the new `user` parameter

3.3.3 `setreservefreeze` should trigger the `collateralconfigurationchanged` event if `ltv` is changed in the config

Submitted by StErMi

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When `setReserveFreeze` is executed, based on `freeze` and `_isPendingLtvSet[asset]` the `ltv` value of the config could be changed.

When `ltv` is changed, usually the `PoolConfigurator` triggers the `CollateralConfigurationChanged` event.

This is not done inside the `setReserveFreeze` function

Recommendation: If `setReserveFreeze` changes the pool's LTV it should trigger the `CollateralConfigurationChanged` event to reflect the change and be coherent with `configureReserveAsCollateral`

3.3.4 `virtual_acc_active` name is not coherent with the other masks defined in `reserveconfiguration`

Submitted by StErMi

Severity: Informational

Context: ReserveConfiguration.sol#L32

Description: `VIRTUAL_ACC_ACTIVE` should be renamed as `VIRTUAL_ACC_ACTIVE_MASK` to be coherent with the other masks.

Recommendation: Rename `VIRTUAL_ACC_ACTIVE` to `VIRTUAL_ACC_ACTIVE_MASK`.

3.3.5 `poolrevisionfourinitialize.initialize` is not initializing the new "pending ltv" logic for already frozen assets

Submitted by StErMi

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When the upgrade script is executed, the `PoolRevisionFourInitialize.initialize` logic is applied. Inside such logic, the new data structure and logic used for frozen address is not applied and `_pendingLtv` and `_isPendingLtvSet` will remain not-initialized for already frozen assets.

This is not a security concern because after the upgrade, unfreezing assets will not do any harm given that the `else if (_isPendingLtvSet[asset])` branch will be skipped because `_isPendingLtvSet` is not initialized.

It's still a good practice to apply the intended behavior during the upgrade for the unfrozen assets to be in a state that is consistent with the new logic.

Recommendation: During the upgrade phase, Aave should loop all the assets and initialize the frozen logic to all the already-frozen assets. If the asset is frozen, the following changes should be applied:

```
_pendingLtv[asset] = currentConfig.getLtv();
_isPendingLtvSet[asset] = true;
currentConfiguration.setLtv(0);
```

3.3.6 Past or current grace period can be setup : conflicting the natspec of new reservedata struct

Submitted by 0xumarkhatab

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `Pool.sol - setLiquidationGracePeriod`:

```
/// @inheritdoc IPool
function setLiquidationGracePeriod(
  address asset,
  uint40 until
) external virtual override onlyPoolConfigurator {
  require(_reserves[asset].id != 0 || _reservesList[0] == asset, Errors.ASSET_NOT_LISTED);
  PoolLogic.executeSetLiquidationGracePeriod(_reserves, asset, until);
}
```

allows setting the graceperiod pram to either in future, present or past due to no validation on it. And its natspec states this also:

```
/// @inheritdoc IPool
/**
* @dev To enable a liquidation grace period, a timestamp in the future should be set,
*      To disable a liquidation grace period, any timestamp in the past works, like 0
```

However if we look at the Natspec of newly updated `ReserveData` struct:

```
struct ReserveData {
   //snip

    //timestamp in the future, until when liquidations are not allowed on the reserve
    uint40 liquidationGracePeriodUntil;

//snip

}
```

The Natspec of the argument states that the timestamp needs to be in the future always. Disregarding any external validation beyond Pool's method for setting grace period, this is an issue where bounds need to be ensured if they are ensured outside of the `Pool.sol`, the Natspec should be updated to reflect that only future grace periods are accepted.

Recommendation: Correct Natspec or implement bounds on `until` param to only be in future, i.e. until `> block.timestamp`.

3.3.7 Decimals mismatching during reserve initialization will break calculations

Submitted by zigtur

Severity: Informational

Context: ConfiguratorLogic.sol#L108, ConfiguratorLogic.sol#L63, ConfiguratorLogic.sol#L77, ConfiguratorLogic.sol#L91, PoolConfigurator.sol#L84-L87

Description: The v3.1 update adds a minimum decimals for listed assets. During a reserve initialization, a newly implemented check ensure that the asset's decimals is greater than or equal to 6.

However, the reserve is initialized with an `underlyingAssetDecimals` parameter that is not checked. This parameter is user controlled, so it can be different than the real asset's decimals.

Proof of concept: The `PoolConfigurator.initReserves` function ensures that the asset's decimals is greater than 5:

```
function initReserves(
  ConfiguratorInputTypes.InitReserveInput[] calldata input
) external override onlyAssetListingOrPoolAdmins {
  IPool cachedPool = _pool;
  for (uint256 i = 0; i < input.length; i++) {
    require(IERC20Detailed(input[i].underlyingAsset).decimals() > 5, Errors.INVALID_DECIMALS); // @POC: ensure
↪  that decimals is greater than 5

    ConfiguratorLogic.executeInitReserve(cachedPool, input[i]); // @POC: Use input parameters to initialize
↪  the reserve
    emit ReserveInterestRateDataChanged(
      input[i].underlyingAsset,
      input[i].interestRateStrategyAddress,
      input[i].interestRateData
    );
  }
}
```

However, the `ConfiguratorLogic.executeInitReserve` will initialize the reserve with a user-controlled parameter named `underlyingAssetDecimals` instead of the asset's decimals:

```
function executeInitReserve(
  IPool pool,
  ConfiguratorInputTypes.InitReserveInput calldata input
) external {
  address aTokenProxyAddress = _initTokenWithProxy(
    input.aTokenImpl,
    abi.encodeWithSelector(
      IInitializableAToken.initialize.selector,
      pool,
      input.treasury,
      input.underlyingAsset,
      input.incentivesController,
      input.underlyingAssetDecimals, // @POC: user-controlled parameter can mismatch asset.decimals
      input.aTokenName,
      input.aTokenSymbol,
      input.params
    )
  );
```

As this user-controlled value is not checked against `asset.decimals`, there can be a mismatch in decimals. This will lead to over or under estimations in future calculations due to this decimals inconsistency.

Recommendation: Consider adding a check to ensure that the user input `underlyingAssetDecimals` matches the expected `asset.decimals`. The following patch implements this check:

```
diff --git a/src/core/contracts/protocol/pool/PoolConfigurator.sol
↪  b/src/core/contracts/protocol/pool/PoolConfigurator.sol
index e023dc6..9e870d5 100644
--- a/src/core/contracts/protocol/pool/PoolConfigurator.sol
+++ b/src/core/contracts/protocol/pool/PoolConfigurator.sol
@@ -82,7 +82,8 @@ abstract contract PoolConfigurator is VersionedInitializable, IPoolConfigurator
   ) external override onlyAssetListingOrPoolAdmins {
     IPool cachedPool = _pool;
     for (uint256 i = 0; i < input.length; i++) {
-      require(IERC20Detailed(input[i].underlyingAsset).decimals() > 5, Errors.INVALID_DECIMALS);
+      uint8 decimals = IERC20Detailed(input[i].underlyingAsset).decimals();
+      require(decimals > 5 && decimals == input[i].underlyingAssetDecimals, Errors.INVALID_DECIMALS);

       ConfiguratorLogic.executeInitReserve(cachedPool, input[i]);
       emit ReserveInterestRateDataChanged(
```

Note: The patch can be applied through `git apply`.

3.3.8 Consider documenting explicitly that virtual accounting is a requirement for "normal" reserves

Submitted by JCN

Severity: Informational

Context: DefaultInterestRateStrategyV2.sol#L122-L124

Description: The documentation regarding Aave's new features currently state that the implementation of the virtual accounting feature includes an `optional virtualUnderlyingBalance field for assets listed on Aave` and that `"Special" assets like GHO (minted instead of supplied) are to be configured without virtual accounting`. However, it is not explicitly mentioned (in the documentation and natspec) that enabling virtual accounting is actually a **requirement** for "normal" reserves, i.e. reserves which support supplying.

As shown in the context above, any reserve configured with virtual accounting `disabled` will *not* have a liquidity rate and the borrow rate will be capped at the `baseVariableBorrowRate`. For "normal" reserves, this would translate to a loss of interest for suppliers and the treasury (if a reserve factor is set) and break the economic incentives of the interest rate model.

Recommendation: Aave should consider explicitly documenting (in documentation and natspec) that virtual accounting is a **requirement** for "*normal*" reserves, and outline the implications of "normal" reserves opting out of using the virtual accounting feature.

3.3.9 Unnecessary storage read during `pool::borrow` calls

Submitted by JCN

Severity: Informational

Context: Pool.sol#L233

Description: When a borrow occurs, the `_maxStableRateBorrowSizePercent` storage variable is read and used to populate the `maxStableRateBorrowSizePercent` field of the `ExecuteBorrowParams` memory struct used during the `executeBorrow` call. This value is then passed into the `maxStableLoanPercent` field of the `ValidateBorrowParams` memory struct used during validation. The value is then ultimately used on L299 of ValidationLogic::validateBorrow when a user is attempting to borrow in stable rate mode.

However, since Aave is deprecating stable rate mode, all reserves have disabled stable rate borrowing and thus any attempts to borrow in stable rate mode will revert on line 286. As a result, line 299 is now unreachable code and therefore the value of the `_maxStableRateBorrowSizePercent` storage variable will never be utilized during borrow calls.

Recommendation: Consider setting the `maxStableRateBorrowSizePercent` field to 0 instead of performing an unnecessary storage read.

3.3.10 Consider optimizing `poolconfigurator::_checknosuppliers` internal function

Submitted by JCN

Severity: Informational

Context: PoolConfigurator.sol#L575

Description: When the `_checkNoSuppliers` internal function is invoked it performs a call to the `Pool::getReserveData` function to grab the reserve state data from the Pool's storage. However, this function will perform unnecessary memory operations in this context since it will now copy the reserve state to a new `ReserveDataLegacy` memory struct. In this internal function call the only reserve state utilized is the `accruedToTreasury` storage value, which is present in both the `ReserveData` and `ReserveDataLegacy` struct. Therefore, the `Pool::getReserveDataExtended` function can instead be invoked in order to avoid unnecessary memory operations and directly retrieve a single copy of the reserve data via the `ReserveData` struct.

Recommendation: Consider invoking the `Pool::getReserveDataExtended` function instead of `Pool::getReserveData` to improve gas efficiency.

3.3.11 The function `swaptovariable()` is missing in `l2pool`

Submitted by ladboy233

Severity: Informational

Context: L2Pool.sol#L83-L89

Description: The function `swapToVariable()` allows a borrower to swap his debt between stable and variable mode.

According to the Aave docs:

> Implementation-wise, this adds a swapToVariable() function in the Aave pool, allowing any address to swap any stable rate user to variable, without changing anything else in the position.

While the contract `Pool` implements the function swapToVariable() but the `L2Pool` contract doesn't implement the function `swapToVariable()` to allow any address to swap any stable rate user to variable, without changing anything else in the position.

Impact:

> @notice Calldata optimized extension of the Pool contract allowing users to pass compact calldata representation
>
> - to reduce transaction costs on rollups.

But a user cannot reduce the transaction costs on rollups by calling `swapToVariableRate`.

Recommendation: Add the `swapToVariable()` function `L2Pool`.

3.3.12 Consider optimizing the `poolconfigurator::setreserveinterestrate...` functions

Submitted by JCN

Severity: Informational

Context: PoolConfigurator.sol#L469, PoolConfigurator.sol#L479

Description: Both functions mentioned above invoke the `Pool::getReserveData` function in order to retrieve a copy of the reserve state data from the Pool's storage.

However, the `getReserveData` function will now perform extra memory operations in order to copy the reserve state into a new `ReserveDataLegacy` struct. This is unnecessary in the context of the two functions shown above since the only reserve value utilized in these function calls is the `reserveInterestRateStrategyAddress`, which is present in both the `ReserveData` and `ReserveDataLegacy` structs.

Therefore, the `Pool::getReserveDataExtended` function can instead be invoked in order to directly retrieve a single copy of the reserve data from the Pool's storage via a `ReserveData` struct.

Recommendation: Consider using the `Pool::getReserveDataExtended` function instead of `Poo::getReserveData` to improve gas efficiency.

3.3.13 `executedropreserve()` doesn't clean interest rates from `defaultreserveinterestratestrategyv2` upon dropping a reserve

Submitted by alix40

Severity: Informational

Context: PoolLogic.sol#L146-L155

Description: After the introduction of the statefull `DefaultReserveInterestRateStrategyV2` the interest rate parameters for each reserve is saved in the local mapping `_interestRateData`. When a new Reserve is added to the Pool, those params are set in the `_interestRateData` mapping but when a Reserve is dropped or removed from the Protocol those params will stay in the mapping and are never cleaned:

```
function executeDropReserve(
  mapping(address => DataTypes.ReserveData) storage reservesData,
  mapping(uint256 => address) storage reservesList,
  address asset
) external {
  DataTypes.ReserveData storage reserve = reservesData[asset];
  ValidationLogic.validateDropReserve(reservesList, reserve, asset);
  reservesList[reservesData[asset].id] = address(0);
  delete reservesData[asset];
}
```

This is inconsistent and wouldn't clean the mapping upon the removal of a token.

Recommendation: We recommend cleaning up the rates params for the dropped param from the `DefaultReserveInterestRateStrategyV2`.

does not include the `user` parameter which is beneficial for code quality and review.

```
function executeSwapBorrowRateMode(
  DataTypes.ReserveData storage reserve,
  DataTypes.UserConfigurationMap storage userConfig,
  address asset,
  address user, // @audit add Natspec for user
  DataTypes.InterestRateMode interestRateMode
) external {
  DataTypes.ReserveCache memory reserveCache = reserve.cache();

  reserve.updateState(reserveCache);

  // ...
}
```

Recommendation: Add Natspec for `user`:

```
+ * @param user The user of the asset being repaid
```

3.3.14 Undocumented function parameters

Submitted by 99Crits, also found by Mansa11

Severity: Informational

Context: BorrowLogic.sol#L308

Description: There are multiple functions in the codebase where parameters got added, but their meaning was not documented in the function docs:

- `BorrowLogic.executeSwapBorrowRateMode` added `address user`.
- `ValidationLogic.validateSupply` added `DataTypes.ReserveData storage reserve` and `address onBehalfOf`.
- `ValidationLogic.validateFlashloanSimple` added `uint256 amount`.