SHERLOCK

Security Review For aave

Public Audit Contest Prepared For:	**Aave DAO**
Lead Security Expert:	**hyh**
Date Audited:	**January 13 - January 22, 2025**
Final Commit:	**21c3014**
Code developed by:	**BGD Labs**

Introduction

Aave is a DeFi liquidity protocol for users to supply and borrow assets. This contest focuses on its v3.3 version, optimising different components of the system like liquidations or bad debt management

Scope

Repository: bgd-labs/aave-v3-origin

Audited Commit: ad00e17d3f733f22bd006fd3b4adcfa91467811d

Final Commit: 21c30148d1484ddec57f5d223f530179b103cae6

Files:

- src/contracts/helpers/AaveProtocolDataProvider.sol
- src/contracts/helpers/WrappedTokenGatewayV3.sol
- src/contracts/protocol/libraries/configuration/ReserveConfiguration.sol
- src/contracts/protocol/libraries/logic/BorrowLogic.sol
- src/contracts/protocol/libraries/logic/BridgeLogic.sol
- src/contracts/protocol/libraries/logic/ConfiguratorLogic.sol
- src/contracts/protocol/libraries/logic/LiquidationLogic.sol
- src/contracts/protocol/libraries/logic/ReserveLogic.sol
- src/contracts/protocol/libraries/logic/ValidationLogic.sol
- src/contracts/protocol/libraries/types/DataTypes.sol
- src/contracts/protocol/pool/Pool.sol

Final Commit Hash

21c30148d1484ddec57f5d223f530179b103cae6

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	0	2

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Security experts who found valid issues

bughuntoor hyh pkqs90

Issue L-1: Deficit is ignored as if it wasn't ever borrowed in IRM supply-demand logic, so interest rates will be shifted downwards over time

Source: https://github.com/sherlock-audit/2025-01-aave-v3-3-judging/issues/202

Summary

`deficit` part of the borrowed funds are now ignored in IRM logic as total debt figure supplied there does not include deficit. This twists any supply-demand logic IRM have / will have as it should't depend on the quality of the debt, but on the what was supplied to the pool and what was loaned from it only. Now deficit is treated as if it was never loaned, which is not correct.

IRM interest rate logic is now twisted proportionally to the realized deficit value compared to the remaining active debt. This doesn't depend on the IRM logic itself as deficit is ignored on ReserveLogic call level.

Root Cause

`deficit` is not the same as `unbacked` in a way that there was no underlying supply for `unbacked` yet (first goes `mintUnbacked`, then `backUnbacked`), while there was supply for `deficit` (funds were supplied, loaned, then their accrual stopped when the debt was marked as bad, then there will be a coverage supply in `eliminateReserveDeficit()`). I.e. for interest rate logic `were never supplied` and `were supplied, were lost and to be compensated` are not the same situations.

Essentially this is ReserveLogic issue, since any IRM logic has no chance to treat deficit correctly since it's not given to it as a parameter. Unlike `supplyUsageRatio` situation mixing Bridge initiated `unbacked` with `deficit` makes little sense for `borrowUsageRatio`, i.e. estimation of supply-demand situation of the pool, so it can't be calculated in any IRM logic.

In order words, `totalDebt` in IRM logic should include deficit, but can't as it's not provided by ReserveLogic's `updateInterestRatesAndVirtualBalance()`:

ReserveLogic.sol#L173-L177

```
    (uint256 nextLiquidityRate, uint256 nextVariableRate) =
    ↪  IReserveInterestRateStrategy(
      reserve.interestRateStrategyAddress
    ).calculateInterestRates(
        DataTypes.CalculateInterestRatesParams({
>>        unbacked: reserve.unbacked + reserve.deficit,
```

DefaultReserveInterestRateStrategyV2.sol#L141-L152

```
    if (params.totalDebt != 0) {
      vars.availableLiquidity =
        params.virtualUnderlyingBalance +
        params.liquidityAdded -
        params.liquidityTaken;

      vars.availableLiquidityPlusDebt = vars.availableLiquidity + params.totalDebt;
>>    vars.borrowUsageRatio =
↪   params.totalDebt.rayDiv(vars.availableLiquidityPlusDebt);
      vars.supplyUsageRatio = params.totalDebt.rayDiv(
        vars.availableLiquidityPlusDebt + params.unbacked
      );
    } else {
```

For the protocol accounting viewpoint deficit is still borrowed from the reserve since the borrowing was done for it, while repayment (in any form) wasn't. I.e. it is being `borrowed` similarly to the healthy debt, the difference is that deficit is expected to be repaid by Umbrella (written off), while healthy debt is expected to be repaid by the borrower. Until the repayment is done both types represent active debt, i.e. what was ever borrowed:

```
    totalDebt
    uint256
>>  The total borrowed from the reserve
```

That is, with the introduction of the deficit the inclusion of deficit to unbacked done as Certora#M-01 mitigation doesn't look to be enough as, keeping the same variables as in issue description, `totDEBT` is no longer `the amount of money that was borrowed` since deficit was borrowed and wasn't yet repaid, so is still borrowed, just being marked for a write off. I.e. for pool it's still a debt, but frozen and not yield bearing.

It cannot be deemed as if it is written off already because usage ratios and IRM logic is based on the balances of funds, and the corresponding supply side balance is present until aToken burn, i.e. supply reduction, on `executeEliminateDeficit()`:

LiquidationLogic.sol#L135-L158

```
>>    IAToken(reserveCache.aTokenAddress).burn(
        msg.sender,
        reserveCache.aTokenAddress,
        balanceWriteOff,
        reserveCache.nextLiquidityIndex
      );
    } else {
      // This is a special case to allow mintable assets (ex. GHO), which by
      ↪  definition cannot be supplied
      // and thus do not use virtual underlying balances.
      // In that case, the procedure is 1) sending the underlying asset to the
      ↪  aToken and
```

```
      // 2) trigger the handleRepayment() for the aToken to dispose of those assets
      IERC20(params.asset).safeTransferFrom(
        msg.sender,
        reserveCache.aTokenAddress,
        balanceWriteOff
      );
      // it is assumed that handleRepayment does not touch the variable debt balance
>>    IAToken(reserveCache.aTokenAddress).handleRepayment(
        msg.sender,
        // In the context of GHO it's only relevant that the address has no debt.
        // Passing the pool is fitting as it's handling the repayment on behalf of
        ↪  the protocol.
        address(this),
        balanceWriteOff
      );
```

GhoAToken.sol#L160-L173

```
  /// @inheritdoc IAToken
  function handleRepayment(
    address user,
    address onBehalfOf,
    uint256 amount
  ) external virtual override onlyPool {
    uint256 balanceFromInterest =
    ↪  _ghoVariableDebtToken.getBalanceFromInterest(onBehalfOf);
    if (amount <= balanceFromInterest) {
      _ghoVariableDebtToken.decreaseBalanceFromInterest(onBehalfOf, amount);
    } else {
      _ghoVariableDebtToken.decreaseBalanceFromInterest(onBehalfOf,
      ↪  balanceFromInterest);
>>    IGhoToken(_underlyingAsset).burn(amount - balanceFromInterest);
    }
  }
```

Internal Pre-conditions

Material deficit was formed for a debt reserve compared to the active debt.

External Pre-conditions

None, it's internal accounting.

Attack Path

The impact will be accrued automatically along with deficit formation.

Impact

IRM logic (any version of it) is artificially shifted towards low usage situation and lower interest rates, despite user funds might being fully/almost fully utilized.

PoC

As an example, let's suppose there is `900 USDC` of deficit, `100 USDC` of the healthy debt and `100 USDC` of the available liquidity in the pool, no unbacked.

It will be:

- `borrowUsageRatio = 100 / 200  = 0.500` in 3.3 and `1000 / 1100 = 0.909` in 3.2 (1)
- `supplyUsageRatio = 100 / 1100 = 0.091` in 3.3 and `1000 / 1100 = 0.909` in 3.2 (2)

DefaultReserveInterestRateStrategyV2.sol#L141-L152

```
    if (params.totalDebt != 0) {
      vars.availableLiquidity =
        params.virtualUnderlyingBalance +
        params.liquidityAdded -
        params.liquidityTaken;

      vars.availableLiquidityPlusDebt = vars.availableLiquidity + params.totalDebt;
>>    vars.borrowUsageRatio =
↪   params.totalDebt.rayDiv(vars.availableLiquidityPlusDebt);
>>    vars.supplyUsageRatio = params.totalDebt.rayDiv(
        vars.availableLiquidityPlusDebt + params.unbacked
      );
    } else {
```

`supplyUsageRatio` is a coefficient between yield generating debt, paying `currentVariableBorrowRate`, and yield requiring liquidity, receiving `currentLiquidityRate`:

DefaultReserveInterestRateStrategyV2.sol#L171-L174

```
>>  vars.currentLiquidityRate = vars
>>    .currentVariableBorrowRate
>>    .rayMul(vars.supplyUsageRatio)
      .percentMul(PercentageMath.PERCENTAGE_FACTOR - params.reserveFactor);
```

This way 3.3 version of (2), `supplyUsageRatio = 100 / 1100 = 0.091`, looks correct as it is `100` units generate yield, while `1100` units expecting it.

On the other hand, `borrowUsageRatio` should push interest rate higher when the demand is high vs `optimalUsageRatio` and vice versa. So, 3.3 version of (1), `borrowUsageRatio = 100 / 200  = 0.500` is not a correct representation of the supply-demand situation in the pool: liquidity supply is `1100`, this is what was supplied and accrued so far, while the loan

demand is `1000`, this is what was taken out as debt and accrued as its interest, i.e. the 3.2 version of (1), `1000 / 1100 = 0.909`, is valid instead.

In other words it is high demand, very low coverage situation: almost all the tokens supplied are utilized by loans (`1000` of `1100` are taken), which have low coverage for the supply (`100` loans generate interest for `1100` supply, because of the bad debt, represented by deficit). By `borrowUsageRatio` computed it is low demand, very low coverage instead, which doesn't represent the real situation and misalign LP incentives. That is, the deficit can't be ignored for borrow demand calculation as demand-supply is a function of loan origination, which happens before debt quality is realized. When debt is issued the demand is set, if supply is fixed for the sake of the example, and the demand doesn't change if that debt turned out to be bad, as the loan is originated already and that is not replayed when loan is §written off.

Current situation can be equivalent to shifting `optimalUsageRatio` and can even surpass that.

Rate dynamics depend on `excessBorrowUsageRatio = (borrowUsageRatio - optimalUsageRatio) / (1 - optimalUsageRatio) = 1 - (1 - borrowUsageRatio) / (1 - optimalUsageRatio)` when `borrowUsageRatio > optimalUsageRatio`, and on `borrowUsageRatio / optimalUsageRatio` when `borrowUsageRatio <= optimalUsageRatio`.

Continuing the same example (1), `borrowUsageRatio = 100 / 200 = 0.500` now and `1000 / 1100 = 0.909` before ignoring the deficit, if `optimalUsageRatio = 0.8` it's `borrowUsageRatio > optimalUsageRatio` and `excessBorrowUsageRatio = (1 - 0.909) / (1 - 0.8) = 0.45`. No `optimalUsageRatio` can make it equivalent when `borrowUsageRatio == 0.5` since even when `optimalUsageRatio == 0` it's `excessBorrowUsageRatio = (1 - 0.5) / (1 - 0) = 0.5 > 0.45`.

Mitigation

Consider including the deficit into both sides of the `borrowUsageRatio` fraction, e.g.:

DefaultReserveInterestRateStrategyV2.sol#L141-L154

```
    if (params.totalDebt != 0) {
      vars.availableLiquidity =
        params.virtualUnderlyingBalance +
        params.liquidityAdded -
        params.liquidityTaken;

      vars.availableLiquidityPlusDebt = vars.availableLiquidity + params.totalDebt;
-     vars.borrowUsageRatio =
↪   params.totalDebt.rayDiv(vars.availableLiquidityPlusDebt);
+     vars.borrowUsageRatio = (params.totalDebt +
↪   params.deficit).rayDiv(vars.availableLiquidityPlusDebt + params.deficit);
      vars.supplyUsageRatio = params.totalDebt.rayDiv(
-       vars.availableLiquidityPlusDebt + params.unbacked
+       vars.availableLiquidityPlusDebt + params.unbacked + params.deficit
      );
```

```
    } else {
      return (0, vars.currentVariableBorrowRate);
    }
```

As a side effect there will no longer be a jump in `borrowUsageRatio` when deficit is increased on bad debt liquidations. Also new bad debt deficit will be treated similarly to the bad debt positions existing pre 3.3 release.

Since it requires adding a `deficit` variable there is no need to include it in `unbacked`:

ReserveLogic.sol#L173-L186

```
    (uint256 nextLiquidityRate, uint256 nextVariableRate) =
    ↪  IReserveInterestRateStrategy(
      reserve.interestRateStrategyAddress
    ).calculateInterestRates(
        DataTypes.CalculateInterestRatesParams({
-         unbacked: reserve.unbacked + reserve.deficit,
+         unbacked: reserve.unbacked,
+         deficit: reserve.deficit,
          liquidityAdded: liquidityAdded,
          liquidityTaken: liquidityTaken,
          totalDebt: totalVariableDebt,
          reserveFactor: reserveCache.reserveFactor,
          reserve: reserveAddress,
          usingVirtualBalance:
          ↪  reserveCache.reserveConfiguration.getIsVirtualAccActive(),
          virtualUnderlyingBalance: reserve.virtualUnderlyingBalance
        })
      );
```

DataTypes.sol#L311-L320

```
  struct CalculateInterestRatesParams {
    uint256 unbacked;
+   uint256 deficit;
    uint256 liquidityAdded;
    uint256 liquidityTaken;
    uint256 totalDebt;
    uint256 reserveFactor;
    address reserve;
    bool usingVirtualBalance;
    uint256 virtualUnderlyingBalance;
  }
```

Protocol Team's Response

The current approach models the possible withdrawals of the system.

It is assumed that the umbrella will always be able to cover the debt that the seized aTokens will not be withdrawn.
Therefore, the modeling of the IR is sound.
We'll reconsider the suggested approach in a future upgrade.

Issue L-2: Liquidator can avoid resolving bad debt with dust supply/transfer while seizing all the borrower's collateral

Source: https://github.com/sherlock-audit/2025-01-aave-v3-3-judging/issues/203

The protocol has acknowledged this issue.

Summary

Whenever a position with one collateral and many debt reserves is up for liquidation with claiming all the collateral, its debt reserves will be deemed bad debt and cleaned up during `executeLiquidationCall()`, which will bear a significant cost for a liquidator.

To avoid that the liquidator can create an additional collateral for the borrower. For example, can send a dust amount of non-isolated mode aToken not used by them yet, enabling it as a collateral via `executeFinalizeTransfer()`. This can be used by liquidators routinely as profit enhancement and leaves all the bad debt intact with deficit not formed.

Root Cause

When `vars.totalCollateralInBaseCurrency > vars.collateralToLiquidateInBaseCurrency` in `executeLiquidationCall()` it is `hasNoCollateralLeft == false` and bad debt clean-up is avoided. For a bad debt bearing position with many debt reserves, liquidators will do that as long as this is profitable, which can frequently be the case on L1.

Internal Pre-conditions

Borrower being liquidated has one collateral, with other supplies being not used as collaterals, and a number of debts. This can be quite common since 3.3 spreading over many debt reserves comes as a natural remedy from 50% rule change from being debt reserve wise to be the whole position wise.

External Pre-conditions

Gas price is high on liquidations so paying for dust aToken transfer or supply on behalf is cheaper than covering gas costs of the deficit formation for all debt reserves of the borrower.

Attack Path

The goal is to trigger `setUsingAsCollateral(id, true)` with some additional action that doesn't require borrower participation. This can be supply with `onBehalfOf = borrower` on straightforward aToken transfer, which is cheaper:

`executeFinalizeTransfer()`, SupplyLogic.sol#L216-L230:

```
      if (params.balanceToBefore == 0) {
        DataTypes.UserConfigurationMap storage toConfig = usersConfig[params.to];
        if (
          ValidationLogic.validateAutomaticUseAsCollateral(
            reservesData,
            reservesList,
            toConfig,
            reserve.configuration,
            reserve.aTokenAddress
          )
        ) {
>>        toConfig.setUsingAsCollateral(reserveId, true);
          emit ReserveUsedAsCollateralEnabled(params.asset, params.to);
        }
      }
```

Impact

Since allowing the 50% of all the total debt to be liquidated at once is a extra payoff for liquidators at the expense of the borrowers in order to provide bad debt clearing and deficit formation, the failure to do so is a direct loss for the borrowers from the 3.3 release. I.e. in the described circumstances nothing changes bad debt vice, it's still unrealized and require manual DAO intervention (repaying on behalf), but borrowers now lose more LB to liquidators.

PoC

According to the contest snapshot it's about `145k` for aToken transfer with enabling the collateral. Supply looks to be more expensive, `176k`.

AToken.transfer.json#L2

```
"full amount; receiver: ->enableCollateral": "144881",
```

It's about 100k per an additional asset bad debt clean-up. Whenever a bad debt bearing borrower have more than 2 debt reserves with one of them capable to take all the collateral it's profitable to transfer aToken and avoid the cleanup (`100*k > 145 if k > 1`, where `k` is number of additional debt reserves).

Mitigation

Since user can always enable the collateral manually one way to control for the issue is to require that automatic use happens on non-dust amount addition only, e.g.:

ValidationLogic.sol#L621-L641

```
  function validateAutomaticUseAsCollateral(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    DataTypes.UserConfigurationMap storage userConfig,
    DataTypes.ReserveConfigurationMap memory reserveConfig,
+   uint256 amountInBaseCurrency,
    address aTokenAddress
  ) internal view returns (bool) {
    if (reserveConfig.getDebtCeiling() != 0) {
      // ensures only the ISOLATED_COLLATERAL_SUPPLIER_ROLE can enable collateral as
      ↪  side-effect of an action
      IPoolAddressesProvider addressesProvider = IncentivizedERC20(aTokenAddress)
        .POOL()
        .ADDRESSES_PROVIDER();
      if (
        !IAccessControl(addressesProvider.getACLManager()).hasRole(
          ISOLATED_COLLATERAL_SUPPLIER_ROLE,
          msg.sender
        )
      ) return false;
-   }
+   } else {
+     // ensures that amount that triggered the action is not below minimum
+     if (amountInBaseCurrency < LiquidationLogic.MIN_LEFTOVER_BASE) return false;
+   }
    return validateUseAsCollateral(reservesData, reservesList, userConfig,
    ↪  reserveConfig);
  }
```

All the uses of `validateAutomaticUseAsCollateral` will need to supply the base currency equivalent amount for the action, e.g.:

SupplyLogic.sol#L215-L230

```
      if (params.balanceToBefore == 0) {
        DataTypes.UserConfigurationMap storage toConfig = usersConfig[params.to];
+       uint256 assetPrice =
↪   IPriceOracleGetter(params.oracle).getAssetPrice(params.asset);
+       uint256 assetUnit = 10 ** reserve.configuration.getDecimals();
+       uint256 amountInBaseCurrency = (params.amount * assetPrice) / assetUnit;
        if (
          ValidationLogic.validateAutomaticUseAsCollateral(
            reservesData,
```

```
            reservesList,
            toConfig,
            reserve.configuration,
+           amountInBaseCurrency,
            reserve.aTokenAddress
          )
        ) {
          toConfig.setUsingAsCollateral(reserveId, true);
          emit ReserveUsedAsCollateralEnabled(params.asset, params.to);
        }
      }
```

Protocol Team's Response

As stated on the docs, the system is designed as a "best effort" approach.
Currently, due to the automation of collateral enabling, there are certain constraints that don't allow a perfect solution in all scenarios.
Dependent on gas price, chain bonus it might be possible to optimize liquidations by not resolving deficit.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.