Aave v3.5 Report

Prepared for: Aave DAO

Code produced by: BGD Labs

Report prepared by: Emanuele Ricci (StErMi), Independent Security Researcher

A time-boxed security review of the **Aave v3.5** protocol was done by **StErMi**, with a focus on the security aspects of the application's smart contracts implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About Aave v3.5

3.5 is an upgrade of the Aave v3 protocol, focusing on refining mathematical precision across the system.

Among other improvements, it reduces unpredictability in rounding by being more explicit on the rounding direction depending on the protocol action type.

Initially planned as part of v3.4, it was split into a separate release to simplify its development and review.

About StErMi

StErMi, is an independent smart contract security researcher. He serves as a Lead Security Researcher at Spearbit and has identified multiple bugs in the wild on Immunefi and on protocol's bounty programs like the Aave Bug Bounty.

Do you want to connect with him?

- stermi.xyz website
- @StErMi on Twitter

Summary & Scope

1. Review of snapshot 5177902bd8472174e841c38982ab4d0a183771a2 (Dated Jun 9, 2025)
2. Review of snapshot c3e88e161b44de14f7cbbfd86749ffb8bb392260 (Dated Jun 26, 2025)
3. Review of snapshot f76f00773016f35dc1a134f090afc5923c6bcbe1 (Dated Jul 17, 2025)

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

Findings Summary

ID	Title	Severity	Status
[M-01]	When liquidation with `receiveAToken=false` liquidate and seize the same token, rates will be calculated using the	Medium	Fixed

ID	Title	Severity	Status
	outdated (bigger) total debt		
[L-01]	`scaledDownLiquidationProtocolFee` calculation should round UP to avoid possible underflow reverts	Low	Fixed
[L-02]	functions that rely on `accruedToTreasury == 0` could be broken because `Pool.mintToTreasury` reverts for `InvalidMintAmount`	Low	Fixed
[L-03]	`IsolationModeLogic` accounted debt against collateral is not accurate	Low	Ack
[L-04]	`LiquidationLogic` does not round up the debt in base currency	Low	Fixed
[L-05]	`ValidationLogic.validateSupply` should use the "raw" supplied amount when the supply cap is validated	Low	Ack
[L-06]	`ValidationLogic.validateBorrow` should use the original `amount` (specified in the `executeBorrow`) and not recalculating it	Low	Ack
[L-07]	Repay debt with `AToken` could cost more than repaying with underlying	Low	Ack
[I-01]	Consider minting treasury shares (as fee) directly instead of simply accounting them and minting later on	Info	Ack
[I-02]	Typos or missing documentation in natspec	Info	Fixed
[I-03]	The new `executeRepay` logic allows `repayWithATokens` to specify an arbitrary `onBehalfOf` user	Info	Ack
[I-04]	Validation of the HF when repaying debt with `AToken` can be avoided when conditions are met	Info	Fixed
[I-05]	General informational issues	Info	Fixed
[I-06]	`SupplyLogic.executeFinalizeTransfer` can turn off the `using-as-collateral` flag before executing `validateHFAndLtvzero` to waste less gas	Info	Fixed
[I-07]	Consider merging `validateHealthFactor` and `validateHFAndLtv` from `ValidationLogic` for a cleaner solution	Info	Ack
[I-08]	`MathUtils.mulDivCeil` should revert when `c` is equal to `0`	Info	Fixed
[I-09]	Early return check in `AToken.mintToTreasury` can now be removed	Info	Ack
[I-10]	Consider aligning the early returns performed in the `balanceOf` and `totalSupply` functions of both the `AToken` and `VariableDebtToken` contracts	Info	Fixed

ID	Title	Severity	Status
[I-11]	Consider aligning the "spend debt allowance" logic from `VariableDebtToken.mint` to the one of `AToken.transferFrom`	Info	Ack
[I-12]	`IncentivizedERC20._burnScaled` events could be more accurate	Info	Ack
[I-13]	`AToken._transfer` is using the wrong amount when the `Transfer` event is emitted	Info	Ack
[I-14]	Consider aligning the `AToken.transferFrom` logic to the OpenZeppelin ERC20 implementation	Info	Ack
[I-15]	`ValidationLogic.validateHFAndLtv` should validate the borrower's `LTV` to be above zero to avoid a division by zero error	Info	Fixed
[I-16]	`Pool.getUserAccountData` could overestimate the borrowable amount	Info	Fixed

[M-01] When liquidation with `receiveAToken=false` liquidate and seize the same token, rates will be calculated using the outdated (bigger) total debt

Context

- LiquidationLogic.sol#L522-L529
- LiquidationLogic.sol#L426-L432

Description

During the liquidation process, when the debt is eliminated, the liquidation flow will call `_burnDebtTokens` that will

1. burn the requested amount of token calling `VToken.burn`
2. update `debtReserveCache.nextScaledVariableDebt` with one of the values returned by `VToken.burn`
3. update the `debtReserve` liquidity and borrow rates (+ virtual underlying balance) by calling `debtReserve.updateInterestRatesAndVirtualBalance` that will internally use `reserveCache.nextScaledVariableDebt` to calculate the non-scaled amount of debt to be then passed to the `IReserveInterestRateStrategy.calculateInterestRates` function

LiquidationLogic.sol#L522-L529

```
  function _burnDebtTokens(
    DataTypes.ReserveCache memory debtReserveCache,
    [...] other inputs
  ) internal {
    bool noMoreDebt = true;
    if (borrowerReserveDebt != 0) {
      // As vDebt.burn rounds down, we ensure an equivalent of <= amount
debt is burned.
>>>       (noMoreDebt, debtReserveCache.nextScaledVariableDebt) =
IVariableDebtToken(
        debtReserveCache.variableDebtTokenAddress
      ).burn(
          borrower,
          hasNoCollateralLeft ? borrowerReserveDebt : actualDebtToLiquidate,
          debtReserveCache.nextVariableBorrowIndex
        );
    }

    // [...] other code

    debtReserve.updateInterestRatesAndVirtualBalance(
>>>       debtReserveCache,
      debtAsset,
      actualDebtToLiquidate,
      0,
      interestRateStrategyAddress
    );
  }
```

ReserveLogic.sol#L128-L173

```
  function updateInterestRatesAndVirtualBalance(
    DataTypes.ReserveData storage reserve,
    DataTypes.ReserveCache memory reserveCache,
    [...] other inputs
  ) internal {
>>>     uint256 totalVariableDebt =
reserveCache.nextScaledVariableDebt.rayMulCeil(
>>>       reserveCache.nextVariableBorrowIndex
>>>     );

    (uint256 nextLiquidityRate, uint256 nextVariableRate) =
IReserveInterestRateStrategy(
```

```
      interestRateStrategyAddress
    ).calculateInterestRates(
        DataTypes.CalculateInterestRatesParams({
          unbacked: reserve.deficit,
          liquidityAdded: liquidityAdded,
          liquidityTaken: liquidityTaken,
>>>         totalDebt: totalVariableDebt,
          reserveFactor: reserveCache.reserveFactor,
          reserve: reserveAddress,
          usingVirtualBalance: true,
          virtualUnderlyingBalance: reserve.virtualUnderlyingBalance
        })
      );

>>>   reserve.currentLiquidityRate = nextLiquidityRate.toUint128();
>>>   reserve.currentVariableBorrowRate = nextVariableRate.toUint128();

    // [...] other code
  }
```

After eliminating the debt, updating `debtReserveCache.nextScaledVariableDebt` and updating `reserve.currentLiquidityRate` and `reserve.currentVariableBorrowRate` (in the `ReserveLogic` part) the liquidation process will execute the `_burnCollateralATokens` if the `liquidator` has asked to seize the underlying instead of the `AToken` of the collateral seized.

The AAVE v3.4 version of `_burnCollateralATokens` would have "forcefully" fetched the `collateralReserve` cache and called `collateralReserve.updateState(collateralReserveCache)`. The AAVE v3.5 is different, and in normal circumstances this missing logic would have not been a problem, but it will in this specific case where the `collateral` seized is the same as the `debt` liquidated.

Because `_burnDebtTokens` has not updated the `nextScaledVariableDebt` attribute of the `collateralReserveCache` memory variable (when debt and collateral tokens are the same), when `_burnCollateralATokens` will call `collateralReserve.updateInterestRatesAndVirtualBalance` the `vars.collateralReserveCache.nextScaledVariableDebt` will contain the "old" scaled debt value before the burning of the liquidated debt.

The `IReserveInterestRateStrategy.calculateInterestRates` will use the wrong `totalDebt` value that will lead to the wrong calculation of the new liquidity and debt rates.

The rates calculated will be higher than they should be and so the indexes, updated by the next `reserve.updateState` will be higher (how much depending on the elapsed time). Suppliers

will earn more interest than deserved, and borrowers will pay more interest than they should.

Recommendations

There are multiple places where `vars.collateralReserveCache.nextScaledVariableDebt` could be updated. One possible suggestion would be to update it in the `else` branch of the condition `if (params.receiveAToken)` only when the debt and collateral token are the same

```
if (params.receiveAToken) {
  _liquidateATokens(reservesData, reservesList, usersConfig,
collateralReserve, params, vars);
} else {
+  if( params.collateralAsset == params.debtAsset ) {
+    vars.collateralReserveCache.nextScaledVariableDebt =
vars.debtReserveCache.nextScaledVariableDebt;
+  }
  _burnCollateralATokens(collateralReserve, params, vars);
}
```

StErMi: the recommendations have been implemented by the commit `1c055988991dc16241c737f30ada0089b2c0fe68`

[L-01] `scaledDownLiquidationProtocolFee` calculation should round UP to avoid possible underflow reverts

Context

- LiquidationLogic.sol#L370-L372

Description

During the liquidation process, the logic will take care to check and recalculates the amount of fees to be transferred to the AAVE Treasury if such amount would be greater than the remaining borrower collateral balance (part of the seized collateral has been already transferred to the liquidator).

```
  // Transfer fee to treasury if it is non-zero
  if (vars.liquidationProtocolFeeAmount != 0) {
    uint256 scaledDownLiquidationProtocolFee =
vars.liquidationProtocolFeeAmount.rayDivFloor(
      vars.collateralReserveCache.nextLiquidityIndex
```

```
    );
    uint256 scaledDownBorrowerBalance =
IAToken(vars.collateralReserveCache.aTokenAddress)
      .scaledBalanceOf(params.borrower);
    // To avoid trying to send more aTokens than available on balance, due
to 1 wei imprecision
    if (scaledDownLiquidationProtocolFee > scaledDownBorrowerBalance) {
      vars.liquidationProtocolFeeAmount =
scaledDownBorrowerBalance.rayMulFloor(
        vars.collateralReserveCache.nextLiquidityIndex
      );
    }

IAToken(vars.collateralReserveCache.aTokenAddress).transferOnLiquidation(
      params.borrower,

IAToken(vars.collateralReserveCache.aTokenAddress).RESERVE_TREASURY_ADDRESS(
),
      vars.liquidationProtocolFeeAmount,
      vars.collateralReserveCache.nextLiquidityIndex
    );
  }
```

The new logic of the `AToken.transfer` will **always** round up the amount of shares that corresponds to the non-scaled amount of assets to be transferred from a user to another. The same rounding direction should also be used when `scaledDownLiquidationProtocolFee` is calculated to avoid underestimating that value, skipping the `liquidationProtocolFeeAmount` re-calculation and then reverting when the `AToken.transferOnLiquidation` logic will try to transfer too many shares from the borrowers balance.

Recommendations

BGD should change the rounding direction used for the calculation of `scaledDownLiquidationProtocolFee`

```
-uint256 scaledDownLiquidationProtocolFee =
vars.liquidationProtocolFeeAmount.rayDivFloor(
+uint256 scaledDownLiquidationProtocolFee =
vars.liquidationProtocolFeeAmount.rayDivCeil(
  vars.collateralReserveCache.nextLiquidityIndex
);
```

I also suggest adding a dev comment that explains that `rayDivCeil` has been used because during the `AToken.transfer` (executed by `AToken.transferOnLiquidation` internally) the

amount of shares transferred to the `receiver` will be calculated following the rounding **UP** direction.

StErMi: the recommendations have been implemented in the commit `b6567d452fee9514683cae11205c94b616c944ed` which has been later on replaced by the commit `070cd23d949d828ad78d098ef481c2049f5eabb6`

[L-02] functions that rely on `accruedToTreasury == 0` could be broken because `Pool.mintToTreasury` reverts for `InvalidMintAmount`

Context

- ValidationLogic.sol#L508
- PoolConfigurator.sol#L514

Description

The `PoolConfigurator.configureReserveAsCollateral`, `PoolConfigurator.setReserveActive`, `PoolConfigurator.setDebtCeiling` and `PoolConfigurator.dropReserve` are functions that require (on top of other requirements) that there are currently no more suppliers for the target reserve.

The no supplier requirement is expressed as `IAToken(reservesData[reserve].aTokenAddress).totalSupply() == 0 && reservesData[reserve].accruedToTreasury == 0`.

The `reserve.accruedToTreasury` value represents the amount of shares that have been accounted to the treasury but have not been minted yet and so, are not accounted yet into the `aToken.totalSupply()`.

To do so, reset `reserve.accruedToTreasury`, withdraw those underlying and proceed with the execution of one of the above cited functions the `Pool.mintToTreasury` function must be executed.

This function will calculate the amount of assets corresponding to the `accruedToTreasury` shares and call `aToken.mintToTreasury(amount, inex)` that will again convert back such amount to shares to be accounted to the treasury.

It's possible that the number of shares in `accruedToTreasury` could round down to zero (given the rounding **down**) during such process and the minting operation in executed by the `AToken` contract will revert with the `InvalidMintAmount` error.

It's possible to reproduce the above scenario when all the "normal suppliers" (common users) have already withdrawn everything from the pool and the last shares to be minted are indeed the one accounted in `accruedToTreasury`

Given that `mintToTreasury` reverts, AAVE will not be able to execute crucial functions like `PoolConfigurator.configureReserveAsCollateral`, `PoolConfigurator.setReserveActive`, `PoolConfigurator.setDebtCeiling` and `PoolConfigurator.dropReserve`.

PoC

```
function testCannotMintToTreasuryForRoundingDown() public {
  address s = makeAddr('s');
  address b = makeAddr('b');

  _setupUser(s);
  _setupUser(b);

  vm.startPrank(s);
  contracts.poolProxy.supply(tokenList.usdx, 1_000e6, s, 0);
  vm.stopPrank();


  vm.startPrank(b);
  contracts.poolProxy.supply(tokenList.weth, 1e18, b, 0);
  contracts.poolProxy.borrow(tokenList.usdx, 100e6, 2, 0, b);
  vm.stopPrank();

  vm.warp(vm.getBlockTimestamp() + 9 minutes);


  vm.startPrank(b);
  contracts.poolProxy.repay(tokenList.usdx, 100000015, 2, b);
  vm.stopPrank();


  vm.startPrank(s);
  uint256 sb = IAToken(address(aUSDX)).balanceOf(s);
  contracts.poolProxy.withdraw(tokenList.usdx, sb, s);
  vm.stopPrank();

  // there are no more AToken supply
  // there are still debt to be repaid
  // the accrued to treasury value is > 0
  assertEq(IAToken(address(aUSDX)).scaledBalanceOf(s), 0);
```

```
    assertEq(IAToken(address(aUSDX)).scaledTotalSupply(), 0);
    assertGt(IAToken(address(vUSDX)).balanceOf(b), 0);

assertGt(contracts.poolProxy.getReserveData(tokenList.usdx).accruedToTreasur
y, 0);

    // reverts because `accruedToTreasury > 0`
    // (AToken scaledTotalSupply is correctly 0 see above asserts)
    vm.prank(poolAdmin);

vm.expectRevert(abi.encodeWithSelector(Errors.ReserveLiquidityNotZero.select
or));
    contracts.poolConfiguratorProxy.setReserveActive(tokenList.usdx, false);

    // `ScaledBalanceTokenBase._mintScaled` will revert because of rounding
down to zero
    address[] memory assets = new address[](1);
    assets[0] = tokenList.usdx;

vm.expectRevert(abi.encodeWithSelector(Errors.InvalidMintAmount.selector));
    contracts.poolProxy.mintToTreasury(assets);
  }
```

Recommendations

One possible option for BGD could be to have an "authed" function that will force-reset `reserve.accruedToTreasury` (only in case of a revert caused by a round down error) without minting any shares. Those shares will be lost, but all the crucial functions cited above will be executable.

StErMi: The commit `070cd23d949d828ad78d098ef481c2049f5eabb6` has addressed the issue. The `PoolLogic.executeMintToTreasury`, `AToken.mintToTreasury`, `AToken.mint` and `ScaledBalanceTokenBase._mintScaled` has been rewritten to be able to directly use shares without double conversion.

The new logic allows `PoolLogic` to directly mint `reserve.accruedToTreasury` shares to the treasury.

[L-03] `IsolationModeLogic` accounted debt against collateral is not accurate

Context

- BorrowLogic.sol#L96
- BorrowLogic.sol#L205
- LiquidationLogic.sol#L360

Description

With the introduction of the new rounding direction for debt mint and burn and with the heavy usage of the `scaledAmount` (to increase precision), the accounting logic of the debt made in isolation mode against a collateral is now not as accurate as it should be.

1. User borrow operation

When a user borrows `amount` of underlying, AAVE will in reality account for `amount.getVTokenMintScaledAmount(index).getVTokenBalance(index)` of debt, which could be `1 wei` more (depending on the rounding result) of the initial `amount` of underlying borrowed by the user.

`IsolationModeLogic.increaseIsolatedDebtIfIsolated` is instead updating the isolated debt with `params.amount` which could be `1 wei` less.

2. User repay operation

When a user repays `amount` of underlying, AAVE will in reality remove `amount.getVTokenBurnScaledAmount(index).getVTokenBalance(index)` of debt, which could be `1 wei` less (depending on the rounding result) of the initial `amount` of underlying repaid as debt by the user.

`IsolationModeLogic.reduceIsolatedDebtIfIsolated` is instead updating the isolated debt with `paybackAmount` which could be `1 more` less.

3. User liquidate borrower's debt

This case is the same as the repay one. `IsolationModeLogic.updateIsolatedDebt` is instead updating the isolated debt with `vars.actualDebtToLiquidate` which could be `1 more` less than the actual debt liquidated by the liquidator.

Recommendations

BGD should consider updating the isolated debt accounting with the actual debt being borrowed or repaid against the collateral in isolation mode.

BGD: Ack. The tracking of ceiling is very inaccurate by design (e.g. not tracking debt accrual).

[L-04] `LiquidationLogic` does not round up the debt in base currency

Context

- LiquidationLogic.sol#L246-L248
- LiquidationLogic.sol#L305-L308

Description

In AAVE 3.5 the rounding direction of the calculation of the debt in the base currency (USD) performed by `GenericLogic._getUserDebtInBaseCurrency` has been changed from rounding down to rounding up.

This change in the rounding direction has not been applied to the debt in USD calculated in the `LiquidationLogic` that could result in `1 wei` (of USD) less than they should be and affect the rest of the liquidation logic that will rely on those values.

```
vars.borrowerReserveDebtInBaseCurrency =
      (vars.borrowerReserveDebt * vars.debtAssetPrice) /
      vars.debtAssetUnit;
```

If underestimated (rounding down instead of up), could avoid the liquidator entering the code block that would recalculate the max debt liquidable `maxLiquidatableDebt` following the Liquidation rules.

```
bool isDebtMoreThanLeftoverThreshold = ((vars.borrowerReserveDebt -
  vars.actualDebtToLiquidate) * vars.debtAssetPrice) /
  vars.debtAssetUnit >=
  MIN_LEFTOVER_BASE;
```

If underestimated (rounding down instead of up), the liquidation logic could revert with the error `MustNotLeaveDust` even if no dust position would be left (given the liquidation logic).

Recommendations

BGD should update the calculation of the above two debt values in the base currency by rounding down the operation following the same rounding rules of `GenericLogic._getUserDebtInBaseCurrency`

StErMi The recommendations have been implemented in the commit `61bb283e99dd4647dbf5083079ee381219cc4ec0`

[L-05] `ValidationLogic.validateSupply` should use the "raw" supplied amount when the supply cap is validated

Context

- ValidationLogic.sol#L82-L88

Description

The new `validateSupply` logic has changed how the supplied `amount` is used when the new total supply (including the new amount) is validated against the `supplyCap`.

The old logic was using the "raw" `amount` value, adding it to the non-scaled value of the `scaledSupply` of the reserve. The new logic is instead using the `scaledAmount` (which could round down by `1 wei`).

The supply cap is used to protect the pool from being exposed above such cap. The real number of tokens that will be supplied (and transferred) to the reserve is given by `amount` and not by `amount.getATokenMintScaledAmount(index).getATokenBalance(index)` which could round down by `1 wei`.

Recommendations

BGD should revert the current logic to the previous one to check the `supplyCap` against the actual amount of underlying being sent by the user to the reserve.

```
require(
  supplyCap == 0 ||
    (
      (IAToken(reserveCache.aTokenAddress).scaledTotalSupply() +
-        scaledAmount +

uint256(reserve.accruedToTreasury)).getATokenBalance(reserveCache.nextLiquid
ityIndex)
-    ) <=
+    ) + amount <=
    supplyCap * (10 ** reserveCache.reserveConfiguration.getDecimals()),
```

```
    Errors.SupplyCapExceeded()
  );
```

BGD: Ack. The check is there to limit `aToken` exposure, therefore we think that considering the `aToken` amount is more appropriate.

[L-06] `ValidationLogic.validateBorrow` should use the original `amount` (specified in the `executeBorrow`) and not recalculating it

Context

- ValidationLogic.sol#L145
- ValidationLogic.sol#L156-L159

Description

The `validateBorrow` logic re-calculate the "original" amount requested by the user during the borrow operation as `vars.amount = params.amountScaled.getVTokenBalance(params.reserveCache.nextVariableBorrowIndex);`. This value does not represent the amount of underlying borrowed by the user, but the amount of debt that corresponds to the debt shares minted by the operation.

With the new rounding direction, `vars.amount` could be `1 wei` more than the original `amount` requested by the user.
The `IERC20(params.reserveCache.aTokenAddress).totalSupply() >= vars.amount` validation should validate that the liquidity of the reserve has enough tokens to satisfy the debt requested by the user, and for that reason, it should use the original `amount` value and not the recalculated one.

Note: I have already stated that this check could be removed, given that the borrow logic would anyway revert for an underflow error if the borrower request to pull from the reserve an amount of liquidity greater than the one that is currently available and accounted in the `reserve.virtualUnderlyingBalance` (see ReserveLogic.sol#L163-L165)

Recommendations

BGD should consider using the original `amount` value specified by the user during the borrow (or flash borrow) request instead of the one re-calculated internally by `validateBorrow`

BGD: The validation using aToken `totalSupply()` was introduced as redundant protection in the v3.1 release, and in the current release, we explicitly don't want to address this type of potentially redundant properties/validations, unless mandatory or creating major issues, which is not the case.

[L-07] Repay debt with `AToken` could cost more than repaying with underlying

Context

- BorrowLogic.sol#L215

Description

AAVE allow the user to repay a debt position directly with underlying token or with `AToken` that will be burned to burn (and repay) the corresponding debt specified by the `params.amount` in the `BorrowLogic.executeRepay` function call.

With the new rounding logic implemented with AAVE v3.5, the amount of `AToken` burned to repay the corresponding amount `params.amount` of debt will always be rounded **UP**.

This could lead the borrower to pay `1 wei` more of `AToken` compared to repaying the debt with raw underlying.
This difference could specifically become expensive when the underlying token is a low-decimal-high-value token like BTC

PoC

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.0;

import 'forge-std/Test.sol';
import {IDefaultInterestRateStrategyV2} from
'../../../src/contracts/interfaces/IDefaultInterestRateStrategyV2.sol';
import {IAaveOracle} from
'../../../src/contracts/interfaces/IAaveOracle.sol';
import {IPoolAddressesProvider} from
'../../../src/contracts/interfaces/IPoolAddressesProvider.sol';
import {IAToken} from '../../../src/contracts/interfaces/IAToken.sol';
import {UserConfiguration} from
'../../../src/contracts/protocol/libraries/configuration/UserConfiguration.s
ol';
import {ReserveLogic} from
```

```
'../../../src/contracts/protocol/libraries/logic/ReserveLogic.sol';
import {ReserveConfiguration} from
'../../../src/contracts/protocol/libraries/configuration/ReserveConfiguratio
n.sol';
import {PriceOracleSentinel} from
'../../../src/contracts/misc/PriceOracleSentinel.sol';
import {SequencerOracle, ISequencerOracle} from
'../../../src/contracts/mocks/oracle/SequencerOracle.sol';
import {DataTypes} from
'../../../src/contracts/protocol/libraries/types/DataTypes.sol';
import {PercentageMath} from
'../../../src/contracts/protocol/libraries/math/PercentageMath.sol';
import {WadRayMath} from
'../../../src/contracts/protocol/libraries/math/WadRayMath.sol';
import {TestnetProcedures} from '../../utils/TestnetProcedures.sol';
import {LiquidationDataProvider} from
'../../../src/contracts/helpers/LiquidationDataProvider.sol';
import {TestnetERC20} from '../../../src/contracts/mocks/testnet-
helpers/TestnetERC20.sol';

contract SPoolTest is TestnetProcedures {
  using stdStorage for StdStorage;

  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;
  using PercentageMath for uint256;
  using WadRayMath for uint256;
  using ReserveLogic for DataTypes.ReserveCache;
  using ReserveLogic for DataTypes.ReserveData;


  PriceOracleSentinel internal priceOracleSentinel;
  SequencerOracle internal sequencerOracleMock;
  LiquidationDataProvider internal liquidationDataProvider;

  function setUp() public virtual {
    initTestEnvironment();

    sequencerOracleMock = new SequencerOracle(poolAdmin);
    priceOracleSentinel = new PriceOracleSentinel(
      IPoolAddressesProvider(report.poolAddressesProvider),
      ISequencerOracle(address(sequencerOracleMock)),
      1 days
    );
    liquidationDataProvider = new LiquidationDataProvider(
      address(contracts.poolProxy),
```

```
      address(contracts.poolAddressesProvider)
    );

    vm.prank(poolAdmin);
    sequencerOracleMock.setAnswer(false, 0);

    // oracle
    _setOraclePrice(tokenList.usdx, 100015000);
    _setOraclePrice(tokenList.weth, 246216320000);
    _setOraclePrice(tokenList.wbtc, 10708564022899);

    // reserve factor

    // IRS
    vm.startPrank(poolAdmin);
    contracts.poolConfiguratorProxy.setReserveInterestRateData(
      tokenList.usdx,
      abi.encode(IDefaultInterestRateStrategyV2.InterestRateData({
        optimalUsageRatio: 9200,
        baseVariableBorrowRate: 0,
        variableRateSlope1: 550,
        variableRateSlope2: 2250
      }))
    );
    contracts.poolConfiguratorProxy.setReserveFactor(tokenList.wbtc, 1000);


    contracts.poolConfiguratorProxy.setReserveInterestRateData(
      tokenList.weth,
      abi.encode(IDefaultInterestRateStrategyV2.InterestRateData({
        optimalUsageRatio: 9000,
        baseVariableBorrowRate: 0,
        variableRateSlope1: 270,
        variableRateSlope2: 8000
      }))
    );
    contracts.poolConfiguratorProxy.setReserveFactor(tokenList.weth, 1500);

    contracts.poolConfiguratorProxy.setReserveInterestRateData(
      tokenList.wbtc,
      abi.encode(IDefaultInterestRateStrategyV2.InterestRateData({
        optimalUsageRatio: 8000,
        baseVariableBorrowRate: 0,
        variableRateSlope1: 400,
        variableRateSlope2: 30000
      }))
```

```
    );
    contracts.poolConfiguratorProxy.setReserveFactor(tokenList.wbtc, 5000);


    vm.stopPrank();
  }

  function testRepayDiff() public {
    (address atoken, , address vtoken) =
contracts.protocolDataProvider.getReserveTokensAddresses(tokenList.usdx);
    address s1 = makeAddr('s1');
    address b1 = makeAddr('b1');

    _approveUser(s1);
    _approveUser(b1);

    _mintToken(s1, tokenList.usdx, 1_000e6);
    _mintToken(b1, tokenList.weth, 1e18);
    _mintToken(b1, tokenList.usdx, 10e6);

    // supply
    vm.prank(s1);
    contracts.poolProxy.supply(tokenList.usdx, 1_000e6, s1, 0);
    vm.prank(b1);
    contracts.poolProxy.supply(tokenList.weth, 1e18, b1, 0);

    // borrow
    vm.prank(b1);
    contracts.poolProxy.borrow(tokenList.usdx, 100e6, 2, 0, b1);

    // 100 days pass by
    vm.warp(vm.getBlockTimestamp() + 100 days);

    uint256 debtToRepay = 10e6;
    uint256 debtBefore = IAToken(vtoken).balanceOf(b1);
    uint256 snapshotId = vm.snapshot();

    // borrower repay 10 USD of debt with 10 USD
    vm.prank(b1);
    contracts.poolProxy.repay(tokenList.usdx, debtToRepay, 2, b1);

    uint256 debtAfter1 = IAToken(vtoken).balanceOf(b1);

    vm.revertTo(snapshotId);

    _mintToken(b1, tokenList.usdx, 100e6);
```

```
    vm.prank(b1);
    contracts.poolProxy.supply(tokenList.usdx, 100e6, b1, 0);

    uint256 collBefore2 = IAToken(atoken).balanceOf(b1);

    vm.prank(b1);
    contracts.poolProxy.repayWithATokens(tokenList.usdx, debtToRepay, 2);

    uint256 collAfter2 = IAToken(atoken).balanceOf(b1);
    uint256 debtAfter2 = IAToken(vtoken).balanceOf(b1);

    uint256 repayCostUnderlying = debtToRepay;
    uint256 repayCostWithAToken = collBefore2-collAfter2;

    assertGt(repayCostWithAToken, repayCostUnderlying);
    assertEq(debtBefore-debtAfter1, debtBefore-debtAfter2);
  }

  function _setOraclePrice(address asset, uint256 newPrice) internal {
    stdstore
      .target(IAaveOracle(report.aaveOracle).getSourceOfAsset(asset))
      .sig('_latestAnswer()')
      .checked_write(newPrice);
  }

  function _approveUser(address user) public {
    vm.startPrank(user);
    TestnetERC20(tokenList.usdx).approve(address(contracts.poolProxy),
UINT256_MAX);
    TestnetERC20(tokenList.wbtc).approve(address(contracts.poolProxy),
UINT256_MAX);
    TestnetERC20(tokenList.weth).approve(address(contracts.poolProxy),
UINT256_MAX);
    vm.stopPrank();
  }

  function _mintToken(address user, address token, uint256 amount) public {
    vm.startPrank(poolAdmin);
    if (token == address(usdx)) usdx.mint(user, amount);
    if (token == address(wbtc)) wbtc.mint(user, amount);
    if (token == tokenList.weth) deal(address(tokenList.weth), user,
amount);
    vm.stopPrank();
  }
}
```

Recommendations

BGD should document this behavior and disclose it to the end user.
BGD should consider removing the "pay with `AToken`" feature if it's not widely used and the loss of the user is considered higher compared to the possible side effects of deprecating such a function (integrators could revert, ...)

BGD: Dropping the `repayWithAToken` behavior will be analyzed for a future release.

[I-01] Consider minting treasury shares (as fee) directly instead of simply accounting them and minting later on

Context

- FlashLoanLogic.sol#L226-L229
- ReserveLogic.sol#L181-L203
- PoolLogic.sol#L104-L134
- ... other places inside the codebase where `reserve.accruedToTreasury` is read to estimate the real `AToken` total supply

Description

In the current implementation of AAVE when `AToken` fees need to be minted to the AAVE Treasury (from liquidation or accrued debt interest), those fees are not "directly" and immediately minted to the treasury account but are rather "accounted" and then later on minted as via the permissionless function `Pool.mintToTreasury`

This choice has the advantage of lowering the gas cost that the user (the liquidator for the liquidation process or the user for the first block's operation that triggers the "full flow" of the `reserve.updateState()`) but comes with side effects:

- when the `AToken.totalSupply` needs to be fetched, the `reserve.accruedToTreasury` needs also to be considered
- the code is more complex and more prone to possible errors (you always need to remember to account for the non-minted-yet `accruedToTreasury`)
- the minting operation `Pool.mintToTreasury` could be timed and gamed by attackers/griefiers, making the value of the not-yet-minted shares less predictable (until they are actually minted)
- the process of accounting the shares as "assets" and then minting shares it is made by

multiple conversion `assets <> shares` which will always introduce rounding imprecisions

Recommendations

BGD should consider replacing the current logic that of assigning fees to the Treasury by "directly" minting `AToken` shares to the treasury instead of accounting them (as assets) and only later on transforming the accounted virtual assets into real `AToken` fees.

BGD: Ack, won't do now, due to high gas implications on all operations.

[I-02] Typos or missing documentation in natspec

Description

- ~~ScaledBalanceTokenBase.sol#L116-L117: consider explaining with a dev comment why the `WadRayMath.reverseRounding` function is used instead of the "natural" rounding direction passed to the `_burnScaled` function.~~
- ~~FlashLoanLogic.sol#L226-L229: Add the dev comment "// Replicate aToken.balanceOf (round down), to always underestimate the fee." like it has been done already in `ReserveLogic._accrueToTreasury`~~
- ~~Aave-v3.5-properties.md?plain=1#L17: consider using the `liquidationProtocolFeeAmount` variable name which stores the protocol fee taken from the liquidation bonus~~
- ~~Aave-v3.5-properties.md?plain=1#L7: consider updating the "Rounding" sub-section of the "Properties" section of the `Aave-v3.5-properties.md` file with the following changes:~~
  - ~~add the properties for the `AToken.totalSupply` (explicitly rounding DOWN)~~
  - ~~add the properties for the `VToken.totalSupply` (explicitly rounding UP)~~

Recommendations

BGD should fix all the issues listed in the above section

StErMi:

The first and second points have been addressed by the commit `070cd23d949d828ad78d098ef481c2049f5eabb6`. The new `TokenMath` logic has been used where needed, providing a clearer and more self-explanatory logic to calculate the conversion between assets and shares (non-scaled and scaled amounts).

The third and fourth points have been addressed by the commit `13effed3e903fac664b53344d0f52587e00407a7`.

[I-03] The new `executeRepay` logic allows `repayWithATokens` to specify an arbitrary `onBehalfOf` user

Context

- Pool.sol#L321-L324
- BorrowLogic.sol#L200-L208

Description

The new `BorrowLogic.executeRepay` will enforce that the `params.user` (`msg.sender`) is **always** healthy after using his **own** `AToken` to repay the debt of the `onBehalfOf` user.

This could enable the possibility for the `Pool.repayWithATokens` function to allow the caller to specify an arbitrary `onBehalfOf` user that could be different from the currently enforced user, which is `msg.sender`.

Recommendations

BGD could consider enabling this new user case that would allow the caller to repay someone else's debt by using the caller's `AToken`.

BGD: We consider the existing repayWithAToken feature already weird. Why would a user ever have same collateral & debt? Economically, it should never make sense. Therefore, we don't want to introduce new functionality that does not make sense.

[I-04] Validation of the HF when repaying debt with `AToken` can be avoided when conditions are met

Context

- BorrowLogic.sol#L200-L208

Description

When the user executes a **repay** debt operation using their `AToken` instead of `underlying` the new logic will validate that, after the operation, the user is still healthy.

Like for a **withdrawal** operation, the health check should be performed only when specific conditions are met:

- the `AToken` was indeed a collateral. By burning collateral, your health check can decrease even if you have repaid some debt.
- the user is still borrowing some tokens. If the user is not borrowing anymore, his health factor will be infinite

Recommendations

BGD should consider executing the call to `ValidationLogic.validateHealthFactor(...)` only when `isCollateral == true && onBehalfOfConfig.isBorrowingAny() == true)`

⚠️ Note that the above change will also **change** how the `executeRepay` with `useATokens == true` behaves. With the existing logic, the transaction would have reverted if the user had used a non-collateral `AToken` to reduce the debt (and still being unhealthy). With the suggested changes, the user would be able to reduce the debt but still being unhealthy after the operation.

StErMi: the recommendations have been implemented in the commit `3031d6eb533123694debd388bc0453bba26dee97`

[I-05] General informational issues

Description

Natspec typos, errors or improvements

- ~~Aave-v3.5-features.md?plain=1#L91: consider documenting the fact that the values used in the `Transfer` and `BalanceTransfer` events, emitted by the `transfer`, `transferFrom`, `mint` and `burn` functions of the `AToken` and `VToken` contracts have been modified, and the value might be different (because of rounding)~~
- ~~Aave-v3.5-properties.md?plain=1#L19: consider specifying that the logic of `GeneralLogic._getUserBalanceInBaseCurrency` has not been changed, it was already rounding down the operation.~~
- ~~BorrowLogic.sol#L76: consider removing the dev comment relative to the rounding direction~~
- ~~GenericLogic.sol#L156 + GenericLogic.sol#L159: the percentage has 2 decimals and not 4. Replace "4 decimals" with "2 decimals" in both the comments~~
- ~~GenericLogic.sol#L223: the rounding comment can be removed~~
- ~~LiquidationLogic.sol#L66-L74: The natspec documentation of the `executeEliminateDeficit` function misses the `@return` parameter introduced with the code change.~~
- ~~LiquidationLogic.sol#L110: the rounding direction comment can be removed~~
- ~~ValidationLogic.sol#L180: the dev comment relative to the rounding direction can be removed~~

- [x] ValidationLogic.sol#L458-L459: the `oracle` and `userEModeCategory` order of the input parameter in the `validateHFAndLtvzero` function's natspec have been inverted compared to the function's signature order.
- [x] VariableDebtToken.sol#L28: the `__unusedGap` should be declared as `uint256` and not `uint`
- [x] VariableDebtToken.sol#L34: consider using a "stronger" suffix for the `__deprecated_ghoUserState` variable name. This store slot must **never** be used now or in the future. There have been cases in the past where a slot declared as deprecated has been resetted and re-used, and it must be clear that it should not happen for this specific one.
- [x] VariableDebtToken.sol#L97: consider re-writing the dev comment relative to the "Max(availableAllowance, (amount, correctAmount))" part to make it more clear and easier to read and understand.
- [x] AToken.sol#L212: consider re-writing the dev comment relative to the "Max(availableAllowance, (amount, correctAmount))" part to make it more clear and easier to read and understand.

Small refactor and code improvements

- [x] IncentivizedERC20.sol#L212-L213: wrap the conditional branch code block in `{}` to align the code with the already adopted code style

Improved Documentation

- [x] ValidationLogic.sol#L445: `userDebtInBaseCurrency` is the sum of all the user's debt converted in USD. Each debt is now converted rounding it **UP**, while before it was always rounded **DOWN**. Given that this is the sum of multiple rounded UP values we could accumulate multiple weis and the `userDebtInBaseCurrency` could be greater than before leading to reverts in `executeBorrow` compared to the previous logic. This behavior should be documented in the v3.5 specification documents.

Recommendations

BGD should fix all the issues listed in the above section

StErMi: The recommendations have been implemented in the commit `4cc61ba49a25643fa9227fb3c0100a9302ac1145`

[I-06] `SupplyLogic.executeFinalizeTransfer` can turn off the `using-as-collateral` flag before

executing `validateHFAndLtvzero` to waste less gas

Context

- SupplyLogic.sol#L210-L224

Description

The `executeFinalizeTransfer` function can move the execution of

```
if (params.scaledBalanceFromBefore == params.scaledAmount) {
  fromConfig.setUsingAsCollateral(reserveId, params.asset, params.from,
false);
}
```

before the `if (fromConfig.isBorrowingAny()) {` branch. With this small code change, the `GenericLogic.calculateUserAccountData` function, invoked internally by `ValidationLogic.validateHFAndLtvzero`, will skip the code executed within the branch `if (vars.liquidationThreshold != 0 && isEnabledAsCollateral) {` for the asset and save some gas.

In this very specific case, the asset `params.asset` has been fully moved from the `params.from` user to the `params.to` user, and it should not be considered a collateral anymore.

Recommendations

BGD should consider performing the following change to `executeFinalizeTransfer` to save gas

```
+if (params.scaledBalanceFromBefore == params.scaledAmount) {
+  fromConfig.setUsingAsCollateral(reserveId, params.asset, params.from,
false);
+}
if (fromConfig.isBorrowingAny()) {
  ValidationLogic.validateHFAndLtvzero(
    reservesData,
    reservesList,
    eModeCategories,
    usersConfig[params.from],
    params.asset,
```

```
      params.from,
      params.oracle,
      params.fromEModeCategory
    );
  }
-if (params.scaledBalanceFromBefore == params.scaledAmount) {
-  fromConfig.setUsingAsCollateral(reserveId, params.asset, params.from,
false);
-}
```

StErMi: The recommendations have been implemented in the commit `64fb8e5d45a48b02edae23d58e54c65c9cb02ab7`

[I-07] Consider merging `validateHealthFactor` and `validateHFAndLtv` from `ValidationLogic` for a cleaner solution

Context

- ValidationLogic.sol#L359-L397
- ValidationLogic.sol#L399-L448

Description

The only difference between the functions `validateHealthFactor` and `validateHFAndLtv` is that the first one performs an additional check relative to the `userCollateralInBaseCurrency`. The rest of the code is a 1:1 copy that could be refactored and merged into a single function that accepts a new `bool` parameter to perform or not (given the context) the `userCollateralInBaseCurrency >= userDebtInBaseCurrency.percentDivCeil(currentLtv)` check.

Recommendations

BGD should consider merging the `validateHealthFactor` and `validateHFAndLtv` functions to reduce the code size of the contract and make it more clear. The new function must accept a new `bool` input that will trigger or not the additional `userCollateralInBaseCurrency >= userDebtInBaseCurrency.percentDivCeil(currentLtv)` check introduced in the current code of `validateHealthFactor`.

BGD: ack. We perceive the current solution as less intrusive, which we see as preferable given the number of changes introduced in v3.4 and v3.5. We'll keep the suggestion in mind for further upgrades.

[I-08] `MathUtils.mulDivCeil` should revert when `c` is equal to `0`

Context

- MathUtils.sol#L108

Description

The current implementation of the new function `mulDivCeil` in the `MathUtils` contract does not revert when the `c` parameter used in the assembly division `div(product, c)` is equal to zero.

Note that the severity of the issue has been set to Informational only because in the current codebase this function is invoked always with `c` as a positive number (`RAY` or `WAD`). The fix is still anyway needed, considering that this function could be used with an arbitrary `c` value in the future.

Recommendations

BGD should revert the execution of `mulDivCeil` when the input parameter `c` is equal to `0` .

StErMi: The recommendations have been implemented in the commit `05b185f98b2c8cc7036ebc6761f762d9b8678b9a`

[I-09] Early return check in `AToken.mintToTreasury` can now be removed

Context

- AToken.sol#L104-L106

Description

The current implementation of `AToken.mintToTreasury` will early return if the number of shares to be minted to the treasury is equal to zero. This check was needed, otherwise the `_mintScaled` function would have internally reverted.

The new logic introduced with AAVE v3.5 is now "directly" minting the `reserve.accruedToTreasury` shares without the need to convert them to assets and then converting them back to shares, with the possible side effects of rounding down the shares in the final process.

Given this change and the fact that will only call `mintToTreasury` when `accruedToTreasury != 0` (see PoolLogic.sol#L123-L125), the early return in `mintToTreasury` can now be removed.

Recommendations

BGD should consider removing the early return given the new logic, the check already performed by `executeMintToTreasury` and the fact that `AToken.mintToTreasury` is only called by `executeMintToTreasury`.

Note: this is a small gas saving improvement performed on a function that won't be called as often as other core functions, it's perfectly fine for the team to acknowledge the issue and keep that in mind for future improvements.

BGD: ack. Not removing the check was a conscious decision. While currently we agree that it's unnecessary, we don't see the value given outweighing the potential pitfalls down the road.

[I-10] Consider aligning the early returns performed in the `balanceOf` and `totalSupply` functions of both the `AToken` and `VariableDebtToken` contracts

Context

- AToken.sol#L134-L139
- AToken.sol#L142-L150
- VariableDebtToken.sol#L68-L76
- VariableDebtToken.sol#L134-L137

Description

The current implementation of `VariableDebtToken.balanceOf` and `AToken.totalSupply` will early return if the scaled balance/supply are equal to zero.

The same behavior has not been implemented for `VariableDebtToken.totalSupply` and `AToken.balanceOf`.

Recommendations

BGD should consider aligning all the above functions with the same behavior, adding or removing the early return from all of them.

StErMi: The recommendations have been implemented in the commit `06df5aae7444d6e59f15f09655b8de1a010aa9f5`

[I-11] Consider aligning the "spend debt allowance" logic from `VariableDebtToken.mint` to the one of `AToken.transferFrom`

Context

- VariableDebtToken.sol#L87-L90

Description

Even if the purpose is different, the logic applied to both the `VariableDebtToken` and `AToken` spending allowance is the same.
BGD should consider aligning the code of those function to match and follow the same logic.
One possible solution could be to align the `VariableDebtToken` one and make the following changes

`VariableDebtToken.mint` changes:

```
function mint(
  address user,
  address onBehalfOf,
  uint256 amount,
  uint256 scaledAmount,
  uint256 index
) external virtual override onlyPool returns (uint256) {
  if (user != onBehalfOf) {
-    uint256 borrowAllowance = _borrowAllowances[onBehalfOf][user];
-    if (borrowAllowance < amount) {
-      revert InsufficientBorrowAllowance(user, borrowAllowance, amount);
-    }
    // [other code]
-    uint256 scaledUp = scaledAmount.getVTokenBalance(index);
    _decreaseBorrowAllowance(
      onBehalfOf,
      user,
-      borrowAllowance >= scaledUp ? scaledUp : borrowAllowance
+      scaledAmount.getVTokenBalance(index);
    );
  }
```

```
  // [other code]
}
```

`DebtTokenBase._decreaseBorrowAllowance` changes:

```
/**
 * @notice Decreases the borrow allowance of a user on the specific debt
token.
 * @param delegator The address delegating the borrowing power
 * @param delegatee The address receiving the delegated borrowing power
 * @param amount The amount to subtract from the current allowance
+ * @param correctedAmount The maximum amount being consumed from the
allowance
 */
-function _decreaseBorrowAllowance(address delegator, address delegatee,
uint256 amount) internal {
+function _decreaseBorrowAllowance(address delegator, address delegatee,
uint256 amount, uint256 correctedAmount) internal {
  -uint256 newAllowance = _borrowAllowances[delegator][delegatee] - amount;
  -_borrowAllowances[delegator][delegatee] = newAllowance;
  -emit BorrowAllowanceDelegated(delegator, delegatee, _underlyingAsset,
newAllowance);
+  uint256 currentAllowance = _borrowAllowances[delegator][delegatee];
+  if (currentAllowance < amount) {
+    revert InsufficientBorrowAllowance(user, borrowAllowance, amount);
+  }
+  uint256 consumption = currentAllowance >= correctedAmount ?
correctedAmount : currentAllowance;
+  uint256 newAllowance = currentAllowance - consumption;
+  _borrowAllowances[delegator][delegatee] = newAllowance;
+  emit BorrowAllowanceDelegated(delegator, delegatee, _underlyingAsset,
newAllowance);
}
```

Recommendations

BGD should consider implementing the above changes to align the logic, code and behavior of the two tokens. The alternative options would be to align the `AToken.transferFrom` and `IncentivizedERC20._spendAllowance` to the variable debt token existing one.

BGD: ack. Will be re-considered in a future upgrade after measuring the exact gas impact.

[I-12] `IncentivizedERC20._burnScaled` events could be more accurate

Context

- ScaledBalanceTokenBase.sol#L123-L131

Description

When the `_mintScaled` logic is executed, some of the minted tokens comes from the interest accrual (if there's any interest) and some of them that comes from the "active" mint operation, but both of them can correctly be aggregated in the same `Transfer` event.

The `_burnScaled` is instead different and more complicated. Let's assume that there's interest to be minted (to make things easier).
In this case, we have some tokens that will be minted and some tokens that will burned, depending on the amounts.
The current logic is simplifying things, aggregating the amounts and just computing the difference and emit "one type" of event.

In this case, the aggregation of the amounts leading to a single event could lead to problems when external tools are using the emitted events to track balance transfers to re-build the operations.

Recommendations

BGD should consider re-implementing the part of `_burnScaled` logic that emit the `Transfer` and `Mint` / `Burn` event to be more accurate emitting multiple events depending on the quantity of tokens that have been minted (for the interest accrual) and burned because of the "root" operation.

BGD: ack. We don't see the added value as the balanceIncrease is already emitted in the `Mint` & `Burn` events.

[I-13] `AToken._transfer` is using the wrong amount when the `Transfer` event is emitted

Context

- AToken.sol#L291

Description

With the AAVE v3.5 update, BGD have applied a specific rounding direction logic to be applied to `mint`, `burn` and `transfer` operations depending on the token type (`AToken`,

`VariableDebtToken`).

When the `AToken` transfer operation is executed, the actual `amount` transferred from a user to another is calculated as `amount.getATokenTransferScaledAmount(index).getATokenBalance(index)` which could result (depending on the rounding) to be `1 wei` more compared to the `amount` specified by the user when the `transfer` (or `transferFrom`) operation is executed.

To be accurate, the final `Transfer` event emission should use `amount.getATokenTransferScaledAmount(index).getATokenBalance(index)` and not `amount` as the input parameter.

Recommendations

BGD should perform the following changes in the `AToken._transfer` logic

```
-emit Transfer(sender, recipient, amount);
+emit Transfer(sender, recipient, scaledAmount.getATokenBalance(index));
```

BGD: With a rebasable token, the concept of transfer itself is very abstract, and on transfer of `amount` we would expect `amount` on the event.

[I-14] Consider aligning the `AToken.transferFrom` logic to the OpenZeppelin ERC20 implementation

Context

- AToken.sol#L200-L214
- https://github.com/bgd-labs/aave-v3-origin-stermi/blob/c3e88e161b44de14f7cbbfd86749ffb8bb392260/src/contracts/protocol/tokenization/base/IncentivizedERC20.sol#L205-L216

Description

BGD should consider aligning the behavior of the `AToken` spending allowance to the one implemented by the OpenZeppelin ERC20.

1. if `currentAllowance == type(uint256).max` the logic should skip the allowance check, the decrease in the allowance and the emission of the `Approval` event
2. The `Approval` event should not emitted when the allowance is decreased by the `transferFrom`. In the OZ implementation, the event is **only** emitted when the allowance is "actively" modified by the owner.

Recommendations

BGD should consider aligning the `AToken` allowance spending to the one implemented by the OpenZeppelin ERC20 implementation.

BGD: Aligning the implementation with OZ would break the event emission.
While we think it could/should be considered, we think it should be done in a separate upgrade.

[I-15] `ValidationLogic.validateHFAndLtv` should validate the borrower's `LTV` to be above zero to avoid a division by zero error

Context

- ValidationLogic.sol#L219
- ValidationLogic.sol#L411-L448

Description

AAVE v3.5 has removed part of the early validations performed in `ValidationLogic.validateBorrow` and moved part of them in new `ValidationLogic.validateHFAndLtv` function.

This new function is only validating the `healthFactor` and if the collateral provided by the user is enough to sustain the new debt given the user's LTV (from the collateral provided).

Unlike the previous logic, the current one does not validate that the `currentLtv` value is non-zero and could end up reverting with a "native" error that is less clear compared to the `LtvValidationFailed` custom error previously used.

Recommendations

BGD should at least add the `require(vars.currentLtv != 0, Errors.LtvValidationFailed());` validation against the user's LTV.
BGD should also consider adding the `require(vars.userCollateralInBaseCurrency != 0, Errors.CollateralBalanceIsZero());` validation that was providing an additional precision on the revert's reason.

Both these requirements should be added **before** the `healthFactor >= HEALTH_FACTOR_LIQUIDATION_THRESHOLD` validation.

StErMi: BGD has reintroduced the `currentLtv != 0` requirement for the `validateHFAndLtv` function in the commit `1381a7377a6436d6bf9005e06a2c3de023488a1e` .

BGD will not reintroduce the `userCollateralInBaseCurrency` check. The following statement has been provided as an explanation:

> Ack.
> Removing the error was a conscious decision, as it is in line with the overall approach to not optimizing for the zero cases.

Note that the `userCollateralInBaseCurrency` check was a nice-to-have suggestion that would have provided more in-depth detail on the revert error; it was not a security requirement.

[I-16] `Pool.getUserAccountData` could overestimate the borrowable amount

Context

- GenericLogic.sol#L196
- PoolLogic.sol#L221-L225

Description

The `GenericLogic.calculateAvailableBorrows` should be updated to reflect the rounding direction changes made in `ValidationLogic.validateHFAndLtv` when the `userCollateralInBaseCurrency >= userDebtInBaseCurrency.percentDivCeil(currentLtv)` check is evaluated.

The current implementation of `GenericLogic.calculateAvailableBorrows` could overestimate by `1 wei` the amount borrowable by the user, resulting in a revert once the user executes the real operation on the pool.

Recommendations

BGD should update the `GenericLogic.calculateAvailableBorrows` logic with the following code change

```
-uint256 availableBorrowsInBaseCurrency =
totalCollateralInBaseCurrency.percentMul(ltv);
+uint256 availableBorrowsInBaseCurrency =
totalCollateralInBaseCurrency.percentMulFloor(ltv);
```

StErMi: The reccomendations have been implemented in the commit `00cec2283ba71bbb2330c637b8dc5b019e00adcc`