Aave v3.3 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 v3.3** 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 v3.3

Aave v3.3 is a new version of Aave V3, upgrading the protocol debt management and liquidations, and making Aave fully compatible with the upcoming Umbrella system.

Key changes include:

- **Bad Debt Management**: bad debt logging & burn functionality to handle unresolvable debts, creating compatibility with the upcoming Umbrella's automated bad debt coverage.
- **Liquidation Optimisations**: Refining the liquidations algorithm to reduce dust debt positions and make the procedure more economically appealing.
- **Misc optimisations**. On data reading and protocol configurations.

References:

- BGD. Aave v3.3 (feat Umbrella)
- BGD. Aave Safety Module - Umbrella

About StErMi

Emanuele, aka **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

***pre-review commit hash* - 07c1da7cebc30e1378fd12f0a9de50e6d0eb8e75**
***review commit hash* - 0c6fac8a1421f21bc62eaf26eb79bd05ee183ed8**

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
[PRE-REVIEW M-01]	`eliminateDeficit` does not update the reserve rates, leading to future miscalculated indexes and rates	Medium	Fixed
[PRE-REVIEW L-01]	Bad debt can be burned and deficit removed for non-active and paused reserves	Low	Fixed
[M-01]	Reserve `deficit` defined as `uint128` could fail to accrue new deficit	Medium	Acknowledged
[M-02]	Multiple Issues when `GHO` is part of the bad-debt liquidation process	Medium	Fixed
[L-01]	Liquidation logic allows the creation of "dust" collateral position even when full debt or collateral positions are not cleaned	Low	Fixed
[L-02]	Collateral position below `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD` do not trigger a 100% close factor like documented in the specification	Low	Fixed
[L-03]	`using-as-collateral` flag is not turned back to `true` when the borrower is self-liquidating with `receiveAToken = TRUE` and seize the whole collateral	Low	Fixed
[L-04]	`Umbrella` `using-as-collateral` flag is not updated in `ReserveLogic.executeEliminateDeficit`	Low	Fixed
[L-05]	Reserve's deficit could be too small to be eliminated	Low	Acknowledged
[I-01]	General informational issues	Low	Fixed + Acknowledged
[I-02]	Inconsistent behavior between liquidation and bad-debt-burning operations relative to the reserve state	Info	Fixed
[I-03]	Consider tracking the bad-debt-burned information that should be accounted in the isolated debt	Info	Acknowledged
[I-04]	Consider documenting the following self-liquidating scenarios	Info	Fixed
[I-05]	Suggestions and refactoring of the `ReserveLogic.executeEliminateDeficit` function	Info	Fixed

ID	Title	Severity	Status
[I-06]	`reserveCache.reserveLastUpdateTimestamp` is not synched with `reserve.lastUpdateTimestamp`	Info	Fixed

[PRE-REVIEW M-01] `eliminateDeficit` does not update the reserve rates, leading to future miscalculated indexes and rates

Context

- ReserveLogic.sol#L331-L391

Description

⚠️ **Note:** The following issue has been reported before the official start of the security review and has been already addressed in the snapshot `0c6fac8a1421f21bc62eaf26eb79bd05ee183ed8` used by the actual review.

When bad debt is removed from the protocol and the `deficit` of the reserve is increased, the `Umbrella` contract will try to eliminate such deficit by calling `Pool.eliminateReserveDeficit` that will execute `ReserveLogic.executeEliminateDeficit` to burn up to `deficit` of `AToken` amount (it works differently for `GHO`, but this issue is relative to the underlying related to `AToken` with virtual accounting).

The internal logic will correctly update the state of the reserve (to update indexes and perform internal accounting if needed) and burn the `AToken` leaving the corresponding `underlying` amount into the `AToken` contract itself.
After that it will decrease the deficit of the reserve based on the amount of `AToken` burned.
The problem in this case is that the logic is not updating the reserve liquidity and debt rates.
While it's true that no liquidity has been added or removed, the rates **must** be updated given that the indexes have been updated and the `reserve.lastUpdateTimestamp` has been synched with `block.timestamp`.

By not updating the rates, we could end up in one of the following scenarios:

Scenario 1) `reserveCache.reserveLastUpdateTimestamp == uint40(block.timestamp)`

Nothing happens here because indexes have been already calculated (in a previous operation) and rates won't change given that no liquidity has been added or removed.

Scenario 2) `reserveCache.reserveLastUpdateTimestamp != uint40(block.timestamp)`

Indexes are updated using the current rates, but the new rates are not updated to reflect the changes related to the new indexes.
Those operations that happen in the very same block won't be affected because they will rely on the already calculated indexes (`getNormalizedIncome` and `getNormalizedDebt` will return the pre-calculated indexes).

The problem rises for those operations that happen in the future, which will be based on "outdated" rates that have skipped an increase or decrease (depending on the `InterestRate` logic and the usage ratio value compared to the optimal one). This missed rates update influences the indexes (which are directly correlated) and everything related to them, causing harm to both the suppliers, borrowers and the protocol itself.

Recommendations

BGD must execute `reserve.updateInterestRatesAndVirtualBalance(reserveCache, params.asset, 0, 0);` to update the reserve rates inside the `ReserveLogic.executeEliminateDeficit` function.

StErMi: In the final snapshot `0c6fac8a1421f21bc62eaf26eb79bd05ee183ed8` used for the review, BGD has implemented the above suggestion.

[PRE-REVIEW L-01] Bad debt can be burned and deficit removed for non-active and paused reserves

Context

- ReserveLogic.sol#L331-L391
- LiquidationLogic.sol#L569-L600

Description

⚠️ **Note:** The following issue has been reported before the official start of the security review and has been already addressed in the snapshot `0c6fac8a1421f21bc62eaf26eb79bd05ee183ed8` used by the actual review.

When bad debt is removed during the liquidation of a borrower that has no more collateral or when the deficit is reduced, the current logic of the code does not check if the involved reserve is **not active** or **paused**.

Recommendations

BGD should consider aborting the operation if the reserve is in a **NOT ACTIVE** or **PAUSED** status.

StErMi: In the final snapshot `0c6fac8a1421f21bc62eaf26eb79bd05ee183ed8` used for the review, BGD has introduced a sanity check that skips burning bad debt or revert eliminating the deficit if the reserve is not active.

Note that both the operations can be still performed if the reserve is **paused**.

[M-01] Reserve `deficit` defined as `uint128` could fail to accrue new deficit

Context

- DataTypes.sol#L53-L55

Description

The reserve `deficit` attribute inside the `ReserveData` struct has replaced the deprecated attribute `currentStableBorrowRate`. Because BGD has decided not to use a new position at the end of the struct, they are forced to use the very same integer type of the old attribute, which is `uint128`.

While user's debt are stored as `uint128`, the `deficit` of a reserve could end up being, theoretically, up to the total supply of the token, which is `uint256`.

This means that there could be a scenario where the increase deficit operation, when bad-debt are burned, ends up reverting alongside the liquidation that has triggered it.

Recommendations

BGD should consider using a new storage slot for the `deficit` position and define it as a `uint256` variable.

[M-02] Multiple Issues when `GHO` is part of the bad-debt liquidation process

Context

- LiquidationLogic.sol#L326-L330
- LiquidationLogic.sol#L316
- ReserveLogic.sol#L368-L379

- GhoAToken.sol#L160-L173
- GhoToken.sol#L54

Description

👉 **Note:** further discussion and brainstorming can be found in the Discussion " `vGHO _ghoUserState[user].accumulatedDebtInterest` is not resetted during bad-debt liquidation" at the end of the report

`GHO` is a "special" token inside the AAVE ecosystem. Before explaining the multiple problems raised by this finding, it's important to understand some key aspects of how `GHO` works and how it's integrated within the AAVE ecosystem.

AAVE fees generated by borrowing `GHO`

It can be borrowed, but it cannot be supplied or flashloaned. Because it cannot be supplied, it also means that it cannot be seized as a collateral asset. For such reason, the `GHO` is configured with `reserveFactor = 0` and `liquidationProtocolFee`.

This means that the AAVE DAO will not receive fees like for a "normal" reserve but in a different way. Like for any borrowed token, the borrower will accrue interest over time. The **whole** interest accrued, that is tracked by `_ghoUserState[borrower].accumulatedDebtInterest` of the `GhoVariableDebtToken` contract, is the **fee** that the DAO will receive when the borrower's debt is repaid.

The debt repayment mechanism works like this. When the borrower repays the debt or the liquidator liquidates the borrower's debt, the system will execute the following actions:

- burn `vGHO` tokens
- transfer the `GHO` from the caller (repayer or liquidator) to the `aGHO` contract
- execute `aGHO.handleRepayment(user = caller, onBehalfOf = borrower, amount=debtRepaid)`

```
  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);
    }
  }
```

When that function is called, it will try first to reduce the debt interest accrued and tracked inside `_ghoUserState[borrower].accumulatedDebtInterest` and if something remain, is the amount of `GHO` that needs to be burned.

The part of `amount` of `GHO` that has not been burned (but decreased from the balance of interest of the borrower) is indeed the **fee** that the AAVE DAO will be later on transferred to the treasury when `aGHO.distributeFeesToTreasury()` is executed

`aGHO` as a `GHO` facilitator

`GHO` can be minted (and burned) only by entities called "facilitators". Each of them has an upper bound limit of `GHO` that can be burned, and the amount of `GHO` generated (minted) is tracked by the `_facilitators[facilitator].bucketLevel` state variable of the `GhoToken` contract.

To make things easy to understand

- when `GHO` is borrowed on AAVE, the `aGHO` call `GHO.mint` and `_facilitators[aGHO].bucketLevel` is increased by the borrowed amount
- when `GHO` debt is repaid during a `repay` or `liquidate` operation on AAVE
  - the repaid `GHO` amount is transferred from the caller to `aGHO`
  - `aGHO.handleRepayment` is executed
    - if the borrower has `GHO` as accrued interest, it will be decreased but **not burned**. That part of the `GHO` repaid is the AAVE DAO fee
    - the remaining amount is indeed the minted `GHO` (debt) that is burned

Theoretically, if all the minted `GHO` debt is repaid, the `_facilitators[aGHO].bucketLevel` should be equal to `0`.

Issues when `GHO` is part of the bad-debt liquidation

When a bad-debt liquidation happens, there are two possible scenarios:

1. Scenario 1: `GHO` is the "main" `debtToken` that is liquidated by the liquidator. In this case, part of the debt is repaid, and the rest is burned as bad debt and accounted as reserve's deficit

2. Scenario 2: `GHO` is a "secondary" bad debt token handled by the execution of the `_burnBadDebt(...)` function

Let's see the first one, the second will be just a possible sub-scenario of it. For the sake of the example, let's say we have the following configuration:

- borrower has borrowed `700 GHO`
- `vGHO.balanceOf(borrower)` returns `1200 GHO`
- `vGHO.getBalanceFromInterest(borrower)` returns `500 GHO`
  This means that:
- `1200 GHO` is the **total** debt of the borrower
- `500 GHO` of those `1200 GHO` of debt are the accrued interest amount, which is the `GHO` that should be received by the AAVE protocol as **fees**.
  With this configuration, the borrower is liquidated by a liquidator that can repay just `100 GHO`. The following actions are applied by the `LiquidationLogic.executeLiquidationCall`:

1. `vGHO.burn(borrower, 1200 GHO, nextVariableBorrowIndex)` will burn the whole borrower debt (bad debt liquidation)
2. `GHO.safeTransferFrom(liquidator, aGHO, 100 GHO)` will transfer `100 GHO` to the `aGHO` contract
3. `aGHO.handleRepayment(liquidator, borrower, 100 GHO)` is executed. The repaid amount is less than the borrower interest (`100 GHO < 500 GHO`), meaning that:
   - no `GHO` are burned
   - borrower `accumulatedDebtInterest` is decreased by `100 GHO`
   As a consequence :

- `vGHO.balance[borrower] = 0` (from 1200)
- `aGHO.balance[aGHO] = 100` (from 0)
- `vGHO._ghoUserState[borrower].accumulatedDebtInterest = 400` (from 500)
- `_reserves[GHO].deficit = 1100 GHO` (from 0)

`GHO` bad-debt handled by `_burnBadDebt`

As already mentioned, the second scenario, where `GHO` bad-debt liquidation is part of the `_burnBadDebt` loop, can be seen a sub-scenario of the one just described. In that case the whole `vGHO` debt is burned by `aGHO.handleRepayment` is **never** called and the **whole** debt is added into the `reserve.deficit` state variable. The whole `vGHO` accrued debt is never repaid to the AAVE DAO.

We can already see the very first problem. The AAVE DAO has "received" in the `aGHO` contract `100 GHO` (from the accrued debt) as fees, but the whole borrower debt has been burned in full.

1. When will the AAVE DAO receive the remaining `400 GHO` of fees that are still tracked by the `vGHO._ghoUserState[borrower].accumulatedDebtInterest`?
2. Is it a coherent state representation that the borrower has no more `vGHO` debt, but `vGHO._ghoUserState[borrower].accumulatedDebtInterest` has a non-zero value?
3. Would it be fair to say that during a bad-debt liquidation the accrued debt, which is indeed the fees that the AAVE should receive, should be written off?

For the first point, with the current implementation, the only way the DAO will be able to receive those `400 GHO` of fees from the borrower will be **if** the borrower, in the future, will **borrow again** an amount of `GHO` that is **greater** than the value stored in `vGHO._ghoUserState[borrower].accumulatedDebtInterest` and at some point in the future it repays it in full. We should also contemplate the problem that more debt interest will be accrued during time in the very same state variable, but I think that it's a problem to be contemplated depending on the solution chosen by BGD.

For the third point, it should be noted that when a "normal" underlying is liquidated, the `ReserveLogic.updateState(...)` (executed at the very beginning of the liquidation flow) would have already accounted for the debt interest that the DAO should receive as **fees** from the borrower (see the `_accrueToTreasury(...)` function). The mechanism is not really the same because the accounted shares are not minted during the process and the underlying is not withdrawn automatically, but at least they are correctly accounted. What you know for sure is that at some point, UMBRELLA will try to eliminate the deficit and allow the DAO to mint and withdraw their fees (that are part of the deficit).

Now that `GHO` bad debt has been removed, and the deficit has been accounted in `_reserves[GHO].deficit`, the `UMBRELLA` contracts needs to remove the `1100 GHO` deficit by calling `Pool.eliminateReserveDeficit` that will execute `ReserveLogic.executeEliminateDeficit`. The following actions will be executed:

- `GHO.safeTransferFrom(UMBRELLA, aGHO, 1100 GHO)` will transfer the `1100 GHO` deficit to the `aGHO` contract
- `aGHO.handleRepayment(UMBRELLA, POOL_ADDRESS, 1100 GHO)` will **burn** the whole amount of `GHO`, while, instead, `400` of those `GHO` should be left in the `aGHO` contract as part of the DAO fees

This happens because when `aGHO.handleRepayment(address user, address onBehalfOf, uint256 amount)` is executed

```
  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);
    }
  }
```

`balanceFromInterest` will be equal to **zero**, given that the `onBehalfOf == POOL_ADDRESS` and the `POOL` have **zero** debt and, for that reason, **zero** accrued interest. For that reason, the whole `amount` will be **burned** when `GHO.burn(1100 GHO)` is executed.

This behavior creates the following issues:

1. The AAVE DAO has not received `400 GHO` of **fees**
2. More `GHO` than it should have been burned. Only `700 GHO` should have been burned, `400 GHO` should have remained in the `aGHO` contract as DAO fees
3. Depending on the state of the `aGHO` facilitator `_facilitators[aGHO].bucketLevel` in the `GHO` contract, it's possible that the `executeEliminateDeficit` call will **REVERT**
4. Depending on the state of the `aGHO` facilitator `_facilitators[aGHO].bucketLevel` in the `GHO` contract, it's possible that normal `repay` and `liquidate` functions where the `GHO` token is the debt token repaid will **REVERT**

To prove the fourth point (the same logic can be applied for the third one) let's see this scenario:

1. At time `t0`: `alice` has borrowed `700 GHO` from AAVE. This means that the `aGHO` contract has triggered `GHO.mint(alice, 700 GHO)` when `aGHO.transferUnderlyingTo` is executed
2. At time `T0+X`: `bob` has borrowed `600 GHO` from AAVE. This means that the `aGHO` contract has triggered `GHO.mint(BOB, 600 GHO)` when `aGHO.transferUnderlyingTo` is executed
3. At time `T0+X`: `alice` debt has accrued to `1200 GHO`, `500` of those are accrued interest, meaning that the state variable `_ghoUserState[alice].accumulatedDebtInterest` in the `vGHO` contract is equal to `500`

Inside the `GHO` contract, we have the following state:
`_facilitators[aGHO].bucketLevel = 700 GHO + 600 GHO = 1300 GHO`. These are all the `GHO` minted by the `aGHO` contract, given the above borrowing operations.
Now the initial scenario of bad-debt removal is applied.

4. The liquidation process does not decrease the `_facilitators[aGHO].bucketLevel` level because the repaid `100 GHO` are **not burned**. They are just decreased from `alice` accumulated debt state variable
5. The `executeEliminateDeficit` will eliminate the `1100 GHO` deficit. The whole amount of `GHO` is burned (while only part should be burned) and the new `_facilitators[aGHO].bucketLevel` value will be equal to `200 GHO`
`bob` has still `600 GHO` of "pure" debt to be repaid, but only `200 GHO` can be burned before the `GHO.burn` function will revert because of an **underflow revert**.

The result is that:

- if `bob` tries to repay his whole debt, the `repay` operation will **revert**
- `bob` is forced to at most repay `200 GHO` of `GHO` debt (+ the accrued interest that is not burned)
- a liquidator is forced to repay at most `200 GHO` of `bob` debt (+ the accrued interest that is not burned) otherwise the liquidation will **revert**

Recommendations

Below, I have brainstormed and implemented three different possible solutions to the above problem, depending on the path that BGD and AAVE want to follow. Each of the solutions has pro and cons depending on what they want to achieve and should be evaluated carefully.

Bear in mind that this is pseudocode, not tested and should be further evaluated once BGD will implement their own version, they should be seen only as a possible example to start from.

Scenario 1: AAVE DAO receives the deserved fees (`vGHO` accrued interest)

In this case, the AAVE DAO fees are paid back once `UMBRELLA` eliminate the deficit via `Pool.eliminateReserveDeficit`

This is a pseudocode that relies on the implementation of custom code in the `GHO` / `vGHO` contracts
This snippet should be placed just **before** the `emit LiquidationCall`

```
if( params.debtAsset == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(params.user);

        // reset the user's `accumulatedDebtInterest` to zero
        // theoretically you could use `decreaseBalanceFromInterest` and
decrease it by `remainingFees`
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.resetBalanceFromInterest(params.user);

        // transfer it to the POOL/UMBRELLA
        // at the moment `ReserveLogic.executeEliminateDeficit`
        // will call `handleRepayment` on behalf of the POOL
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.increaseBalanceFromInterest(address(this));

        // TODO: emit an event if needed
}
```

Now we need to do the same inside the `_burnBadDebt` function implementation **after** that `_burnDebtTokens` has been executed

```
if( reserveCache.variableDebtTokenAddress == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(user);

        // reset the user's `accumulatedDebtInterest` to zero
        // theoretically you could use `decreaseBalanceFromInterest` and
decrease it by `remainingFees`
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.resetBalanceFromInterest(user);

        // transfer it to the POOL/UMBRELLA
        // at the moment `ReserveLogic.executeEliminateDeficit`
        // will call `handleRepayment` on behalf of the POOL
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.increaseBalanceFromInterest(address(this));
        // TODO: emit an event if needed
}
```

The `reserve.deficit` will contain both the `GHO` that should be correctly burned to recover from the bad-debt liquidation, but also the one that should go toward the AAVE DAO as fees. That amount will be equal to the value returned by `vGHO.getBalanceFromInterest(POOL_ADDRESS)`

When `ReserveLogic.executeEliminateDeficit` is executed, the `IAToken(reserveCache.aTokenAddress).handleRepayment(msg.sender, address(this), balanceWriteOff)` will:

- decrease the POOL `accumulatedDebtInterest`
- burn any remaining `GHO` token

The cons of this solution are:

1. you need to implement two new authed functions on `vGHO` that should be callable by `POOL` (or `aGHO`)
2. you "transfer" the user's `accumulatedDebtInterest` amount to the `POOL` (or UMBRELLA) to later on decrease it when Umbrella tries to eliminate the deficit. It can be seen wrong because `accumulatedDebtInterest` represents the accrued debt interest of the user and the POOL (or UMBRELLA) cannot (and should not) have debt and accrue it. It's also incorrect that `accumulatedDebtInterest > 0` when there's no debt in the user account.

Scenario 2: AAVE DAO won't receive fees

In this case, the remaining borrower `accumulatedDebtInterest` should be removed, and the deficit discounted by it, given that that value represents indeed the AAVE DAO fees that have not been received.

The first snipped should be changed like this

```
if( params.debtAsset == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(params.user);

        // reset the user's `accumulatedDebtInterest` to zero
        // theoretically you could use `decreaseBalanceFromInterest` and
decrease it by `remainingFees`
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.resetBalanceFromInterest(params.user);

        // discount the fees from the reserve deficit
        // AAVE DAO will not receive them
        debtReserve.deficit -= remainingFees.toUint128();

        // TODO: emit an event if needed
}
```

The second snipped should be changed like this:

```
if( reserveCache.variableDebtTokenAddress == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(user);

        // reset the user's `accumulatedDebtInterest` to zero
        // theoretically you could use `decreaseBalanceFromInterest` and
decrease it by `remainingFees`
        // NOTE: this function must be implemented and should be callable
only by
        // the pool itself
        vGHO.resetBalanceFromInterest(user);


        // discount the fees from the reserve deficit
        // AAVE DAO will not receive them
        currentReserve.deficit -= remainingFees.toUint128();

        // TODO: emit an event if needed
}
```

👉 **Note:** the above code should be refactored and re-ordered to emit the `DeficitCreated` event with the deficit already discounted.

The cons of this solution are:

1. you need to implement two new authed functions on `vGHO` that should be callable by `POOL` (or `aGHO`)
2. AAVE does not receive any fees

Scenario 2: AAVE DAO will receive fees if and when the borrower re-borrow `GHO` again

In this case, we won't repay AAVE fees when the deficit is eliminated by UMBRELLA, but we will wait that the borrower will re-borrow again `GHO` and repay it in time. Obviously, it's totally possible that it won't happen or that the borrower won't borrow enough to repay the already accrued interest.

The first snipped should be changed like this

```
if( params.debtAsset == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(params.user);

        // we DO NOT reset the user's `accumulatedDebtInterest`
        // user will repay those fees in time if and when it will borrow GHO
again

        // discount the fees from the reserve deficit
        // AAVE DAO will not receive them
        debtReserve.deficit -= remainingFees.toUint128();

        // TODO: emit an event if needed
}
```

The second snipped should be changed like this:

```
if( reserveCache.variableDebtTokenAddress == address(vGHO) ) {
        // fetch the remaining accrued interest of the user
        uint256 remainingFees = vGHO.getBalanceFromInterest(user);

        // we DO NOT reset the user's `accumulatedDebtInterest`
        // user will repay those fees in time if and when it will borrow GHO
again

        // discount the fees from the reserve deficit
        // AAVE DAO will not receive them
        currentReserve.deficit -= remainingFees.toUint128();

        // TODO: emit an event if needed
}
```

👉 **Note:** the above code should be refactored and re-ordered to emit the `DeficitCreated` event with the deficit already discounted.

The cons of this solution are:

1. AAVE won't receive immediately the whole amount of deserved fees when UMBRELLA eliminates the deficit
2. It's possible that the borrower will never re-borrow `GHO` again or not enough to repay the existing accumulated fees
3. It creates an inconsistency where the borrower's `vGHO` balance is zero (no debt, has been cleaned) but `accumulatedDebtInterest`, which represents debt accrued interest, is greater than zero

StErMi: The recommendations have been implemented in the commit 9d0ee8dfc5944b31c2fa4d2383ec0b628ef4f646.
The AAVE DAO will receive part of the fees only when `vGHO` is the liquidated debt asset. The remaining one will be eliminated (not paid) and discounted from the deficit.

[L-01] Liquidation logic allows the creation of "dust" collateral position even when full debt or collateral positions are not cleaned

Context

- LiquidationLogic.sol#L542-L558
- LiquidationLogic.sol#L243-L251

Description

Following the "Liquidation: Forced position cleanup" section of the specification file, the `LiquidationLogic.executeLiquidationCall` should allow the liquidation of debt if and only if at least one of the following scenarios is true:

1. the liquidator fully liquidates the debt position
2. the liquidator fully seizes the collateral position
3. the resulting debt and collateral positions are above the `MIN_LEFTOVER_BASE` threshold

The above rule is needed to prevent the liquidator to leave the protocol with "dust" position in the collateral or debt side that create problem in case of cleaning bad debt or cleaning liquidatable debt for the next liquidators.

Following the code inside the `_calculateAvailableCollateralToLiquidate` function, we see that the total collateral seized during the operation is split into the following variables:

- `actualCollateralToLiquidate` which is the amount of collateral received by the liquidator

- `liquidationProtocolFeeAmount` which is the amount of collateral that will be sent to the protocol as part of the liquidation fee

The total collateral, removed from the borrower's collateral position is equal to the sum of the two.
The final collateral value that will remain in the borrower's collateral position is then equal to `userCollateralBalance - (actualCollateralToLiquidate + liquidationProtocolFeeAmount)`

But the logic inside the branch

```
    if (
      vars.actualDebtToLiquidate < vars.userReserveDebt &&
      vars.actualCollateralToLiquidate + vars.liquidationProtocolFeeAmount <
      vars.userCollateralBalance
    )
```

does not consider the `liquidationProtocolFeeAmount` inside the calculation, allowing the creation of "dust" collateral positions

```
bool isCollateralMoreThanLeftoverThreshold = ((vars.userCollateralBalance -
        vars.actualCollateralToLiquidate) * vars.collateralAssetPrice) /
        vars.collateralAssetUnit >=
        MIN_LEFTOVER_BASE;

require(
        isDebtMoreThanLeftoverThreshold &&
isCollateralMoreThanLeftoverThreshold,
        Errors.MUST_NOT_LEAVE_DUST
);
```

Recommendations

BGD should include the `vars.liquidationProtocolFeeAmount` value in the logic that calculates `isCollateralMoreThanLeftoverThreshold` given that the fee sent to the protocol is **not** included in the `actualCollateralToLiquidate` amount, but it's still "removed" from the final collateral balance of the borrower.

```
  bool isCollateralMoreThanLeftoverThreshold = ((vars.userCollateralBalance
-
-    vars.actualCollateralToLiquidate) * vars.collateralAssetPrice) /
+    vars.actualCollateralToLiquidate - vars.liquidationProtocolFeeAmount) *
vars.collateralAssetPrice) /
```

```
    vars.collateralAssetUnit >=
    MIN_LEFTOVER_BASE;
```

StErMi: The recommendations have been implemented in the commit 57411c9a3c2a7a9fe88cf259cbae5bea6d6267a4. Now, the Liquidation Protocol Fee `liquidationProtocolFeeAmount` value is correctly included when `isCollateralMoreThanLeftoverThreshold` is calculated.

[L-02] Collateral position below `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD` do not trigger a 100% close factor like documented in the specification

Context

- Aave-v3.3-features.md#L63-L64
- LiquidationLogic.sol#L189-L194

Description

The specification document of the "Liquidation: Position size dependent 100% close factor" section states that

> Therefore in order to reduce the accumulation of minor debt positions, a new mechanism is introduced:
> Liquidations up to a 100% close factor are now allowed whenever the total principal or the total debt of the user on the specific reserve being liquidated is below a `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD`

From the specification it is clear that if the debt **or** the collateral value in USD is below the `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD`, the liquidation should be allowed to liquidate the 100% of the debt, even if the `HF` is above the `CLOSE_FACTOR_HF_THRESHOLD` threshold.

The current implementation is instead only checking the debt value

```
uint256 maxLiquidatableDebt = vars.userReserveDebt;
// but if debt is above or equal MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD and
health factor CLOSE_FACTOR_HF_THRESHOLD
// this amount may be adjusted
if (
  vars.userReserveDebtInBaseCurrency >= MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD
&&
  vars.healthFactor > CLOSE_FACTOR_HF_THRESHOLD
) {
        // CF logic
}
```

Recommendations

If the specification is correct, BGD should:

1. Update the dev comment before the `if` branch. The comment does not take in consideration the collateral specification condition
2. Update the code to calculate the `userReserveCollateralInBaseCurrency` value and use it in the `if` branch to allow a 100% close factor in case such value is `<` `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD`

StErMi: The recommendations have been implemented in the commit 027aaed77f89f3d7453764bd94159b5693450169

[L-03] `using-as-collateral` flag is not turned back to `true` when the borrower is self-liquidating with `receiveAToken = TRUE` and seize the whole collateral

Context

- LiquidationLogic.sol#L392-L413

Description

During the liquidation process, if the whole collateral of the borrower is seized, the logic will correctly turn the `using-as-collateral` flag to `false`

```
    if (
      vars.actualCollateralToLiquidate + vars.liquidationProtocolFeeAmount
==
      vars.userCollateralBalance
    ) {
      userConfig.setUsingAsCollateral(collateralReserve.id, false);
      emit ReserveUsedAsCollateralDisabled(params.collateralAsset,
params.user);
    }
```

If the liquidation is executed with the `receiveAToken` input parameter equal to `true`, the `_liquidateATokens` internal function will execute

```
  function _liquidateATokens(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ExecuteLiquidationCallParams memory params,
    LiquidationCallLocalVars memory vars
  ) internal {
    uint256 liquidatorPreviousATokenBalance =
IERC20(vars.collateralAToken).balanceOf(msg.sender);
    vars.collateralAToken.transferOnLiquidation(
      params.user,
      msg.sender,
      vars.actualCollateralToLiquidate
    );

    if (liquidatorPreviousATokenBalance == 0) {
      DataTypes.UserConfigurationMap storage liquidatorConfig =
usersConfig[msg.sender];
      if (
        ValidationLogic.validateAutomaticUseAsCollateral(
          reservesData,
          reservesList,
          liquidatorConfig,
          collateralReserve.configuration,
          collateralReserve.aTokenAddress
        )
      ) {
        liquidatorConfig.setUsingAsCollateral(collateralReserve.id, true);
        emit ReserveUsedAsCollateralEnabled(params.collateralAsset,
msg.sender);
      }
    }
  }
```

The above function is correctly executed **before** the actual seizing (transfer) of the `AToken` but it also means that the `liquidatorPreviousATokenBalance` value will return a value that is `>`

`0`, failing to re-enable the `using-as-collateral` flag of the self-liquidating borrower that will re-receive the collateral when `vars.collateralAToken.transferOnLiquidation` is executed.

By no re-enabling it, the `borrower` will end up in a worst health factor situation compared to the one before the liquidation. Below, you can find some explanatory examples that compare the current and the "fixed" behavior:

Scenario 1

- no liquidation bonus, no liquidation protocol fee to make it simple for the example
- $1000 DAI as debt
- $300 WETH as collateral (`use-as-collateral` flag is `TRUE`)
- $700 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- WETH LT = 83%
- USDT LT = 78%
- HF = 0,795

The `liquidator` liquidate itself by repaying $300 DAI debt and seizing $300 of WETH with `receiveAToken` flag to `true`

With the existing logic, we have the following outcome:

- $700 DAI debt
- $0 WETH as collateral (they have been resupplied, but the `use-as-collateral` flag is `FALSE`)
- $700 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- HF has been **decreased to 0,78** (less debt but also less collateral)
- WETH is not seen as a collateral anymore
- The `borrower` is still liquidatable with an even worst HF

With the "fix" we would have

- $700 DAI debt
- $300 WETH as collateral (`use-as-collateral` flag is `TRUE`)
- $700 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- HF has been **increased to 1,135** (less debt, same collateral)
- WETH is still a collateral
- The `borrower` is NOT liquidatable anymore (better HF)

So the result is that the `borrower` has basically repaid some of their debt and is not liquidatable anymore

Scenario 2

The main difference compared to "Scenario 1" is that I have reduced the USDT collateral amount

- no liquidation bonus, no liquidation protocol fee to make it simple for the example
- $1000 DAI as debt
- $300 WETH as collateral (`use-as-collateral` flag is `TRUE`)
- $300 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- WETH LT = 83%
- USDT LT = 78%
- HF = 0,483

With the existing logic, we have the following outcome:

- $700 DAI debt
- $0 WETH as collateral (they have been resupplied, but the `use-as-collateral` flag is `FALSE`)
- $300 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- HF has been **decreased to 0,483** (less debt but also less collateral)
- WETH is not seen as a collateral anymore
- The `borrower` is still liquidatable with an even worst HF

With the "fix" we would have

- $700 DAI debt
- $300 WETH as collateral (`use-as-collateral` flag is `TRUE`)
- $700 USDT as collateral (`use-as-collateral` flag is `TRUE`)
- HF has been **increased to 0,69** (less debt, same collateral)
- WETH is still a collateral
- The `borrower` is STILL liquidatable anymore (better HF)

The main difference is that because the borrower is STILL liquidatable, the WETH collateral that is still "enabled" as collateral can be seized by the liquidator.

Recommendations

BGD should handle this self-liquidation edge case scenario, ensuring that the `using-as-collateral` flag is turned back to true if the borrower is self-liquidating with `receiveAToken = true` and seizes his whole collateral.

StErMi: The recommendations have been implemented in the commit [d9450ad8ae89c685bde11383457580bfd522ce7b]()

[L-04] `Umbrella` `using-as-collateral` flag is not updated in `ReserveLogic.executeEliminateDeficit`

Context

- [ReserveLogic.sol#L330-L386]()

Description

The `Umbrella` contract that executes the `ReserveLogic.executeEliminateDeficit` function should be treated as a "normal" AAVE account that could end up depleting his whole `params.asset` collateral balance after writing off the reserve deficit.

The current implementation of `executeEliminateDeficit` is not checking if the `params.asset` is a collateral token for the `Umbrella` and is not turning the `using-as-collateral` config flag to `false` if the `balanceWriteOff` amount is equal to the total user's balance before the burn operation.

Recommendations

BGD should follow the same logic used in a repay with `AToken` or withdraw operation and update the `using-as-collateral` flag to `false` if the asset was used as collateral and the user has no more balance.

StErMi: The recommendations have been implemented in the commit [91ed89b5462e0109013bf9a68fdc2c273a0c95a3]()

[L-05] Reserve's deficit could be too small to be eliminated

Context

- [ReserveLogic.sol#L355-L360]()
- [ScaledBalanceTokenBase.sol#L100-L101]()

Description

If the `amount` specified by `UMBRELLA` or the `reserve.deficit` is smaller than the `reserveCache.nextLiquidityIndex` index, the `AToken.burn` operation will revert when `ScaledBalanceToken._burnScaled` will try to burn the scaled down amount:

```
  function _burnScaled(address user, address target, uint256 amount, uint256 index) internal {
    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.INVALID_BURN_AMOUNT);

    // ... other code
  }
```

If such a scenario happens, the `UMBRELLA` contract won't be able to write off the reserve's deficit until more deficit is accrued to avoid the burn operation to revert.

Recommendations

BGD should at least document the above edge case scenario and brainstorm a possible solution to the issue.

BGD: ACK, won't fix

As the deficit is stored as a scaled-up token balance but is detached from the index growth, this scenario is indeed possible. It does not pose any problem for the system though to have some temporary, minimal non-eliminateable deficit.

[I-01] General informational issues

Description

Code refactoring

- [x] ~~Errors.sol#L103: the error `USER_NOT_IN_BAD_DEBT` is never user in the codebase and can be removed. The errors code after this one should be decreased by 1 to fill the ID gap.~~
- [x] ~~ReserveLogic.sol#L17: `ValidationLogic` is never used in `ReserveLogic`, the import can be removed.~~
- [ ] Consider renaming **everything** inside the liquidation process related to the `user` variable to some declination of `borrower` to make things clear. This includes `ExecuteLiquidationCallParams.user`, `ExecuteLiquidationCallParams.userEModeCategory`, `Pool.liquidationCall` parameters, `LiquidationLogic.executeLiquidationCall` parameters and all the internal variables inside the `LiquidationLogic` lib.

- [x] ~~LiquidationLogic.sol#L217: `vars.debtToRepayInBaseCurrency` returned by `_calculateAvailableCollateralToLiquidate` in `LiquidationLogic` is fetched but never used. Consider removing it everywhere~~
- [ ] ReserveLogic.sol#L44: consider declaring the `caller` input of the `DeficitCovered` event as `indexed`. While it's true that right now only the `Umbrella` contract can call it, things could change in the future.
- [x] ~~Pool.sol#L20: `IAccessControl` is never used in `Pool`, the import can be removed.~~

Natspec typos, errors or improvements

- [x] ~~ReserveLogic.sol#L322: the `@notice` comment of the `executeEliminateDeficit` does not take in consideration that reserve could reference a **non-virtual** account token (like `GHO`). Update it accordingly.~~
- [x] ~~IPool.sol#L807: the `@notice` comment of the `eliminateReserveDeficit` does not take in consideration that reserve could reference a **non-virtual** account token (like `GHO`). Update it accordingly.~~
- [x] ~~Pool.sol#L73: avoid using hard-coded strings in the `onlyUmbrella` modifier and define the role/name `UMBRELLA` as a constant~~
- [x] ~~BorrowLogic.sol#L217: the highlighted comment should be moved before or inside (as the very first thing) the `if (params.useATokens) {` branch~~

Recommendations

BGD should fix all the suggestions listed in the above section

StErMi: Part of the issues has been resolved, the remaining ones have been acknowledged by BGD.

- commit 86d07df27b56fa02034323fa1763da7f031e798d removes the unused `USER_NOT_IN_BAD_DEBT` error
- commit 21f26164c344393068a02dcb49ae8118d073ecb4 removes the unused `ValidationLogic` import from `ReserveLogic`
- commit 027aaed77f89f3d7453764bd94159b5693450169 removes the `debtToRepayInBaseCurrency` struct variable (and everything related to it) which was unused in the whole liquidation logic
- commit 905ade8e6780f75671ce2e1bb5ef960a471804aa removes the unused `IAccessControl` import from `Pool`
- commit 9cfb2fcc7f38d2c4d5662ef5167a90abb6405475 solves some of the recommendations

[I-02] Inconsistent behavior between liquidation and bad-debt-burning operations relative to the reserve state

Context

- LiquidationLogic.sol#L590
- ValidationLogic.sol#L389-L399

Description

With the current implementation of `ValidationLogic.validateLiquidationCall` a liquidation operation cannot be performed on a `collateralAsset` that is **not active or paused** or on a `debtAsset` that is **not active or paused**.

Unlike the validation of the liquidation operation, when the borrower has not more collateral, the protocol will try to burn the **bad debt** of **all** the **active** reserves where the borrower has an opened debt position, even if the reserve is **paused**.

This behavior creates an inconsistency between the liquidation logic, that would revert if the debt reserve is paused, and the burn-bad-debt logic, which allows the burning of bad debt of **paused** reserves. It should also be noted that the same logic, of reverting the operation when the reserve is paused, is also followed by for the `repay` operation. In general, any user-facing operation (supply, withdraw, repay and so on) will revert when the reserve is paused.

Recommendations

BGD should clearly define which are the scenarios where the bad debt should and need to be burned, no matter what the state of the reserve, and which scenarios instead should be correctly skip such operation.

In specific, there should be a clean explanation on why a **non-active** reserve should not be bad-debt cleaned, while a **paused** one can be bad-debt cleaned and why those operations would fail during a "normal" user-facing operation when the reserve is paused.

StErMi: The recommendations have been implemented in the commit b21b164c6018152e6bc51794dc840779b3088884, specifically in the commit 48b187b8ea146b858549be27e14b73a2529a516c

[I-03] Consider tracking the bad-debt-burned information that should be accounted in the

isolated debt

Context

- LiquidationLogic.sol#L277-L286

Description

As the dev comment states, when `IsolationModeLogic.updateIsolatedDebtIfIsolated` is invoked in the `executeLiquidationCall` function, the isolated debt will be reduced only by the amount of debt that the liquidator is repaying and not the actual debt burned, that could include bad debt.

If there's a bad debt burning, the information is tracked by the event `emit DeficitCreated(user, debtAsset, outstandingDebt);`, which **does not include** which was the isolated asset associated to the debt.

This means that when the DAO wants to increase the isolated asset debt ceiling to compensate for the "missed update" of the bad-debt burned portion, it will need to go thought the transaction history to calculate which was the isolated asset used by the user at the time of burning of bad-debt.

Recommendations

BGD should consider triggering an additional event, or adding this information to the existing `DeficitCreated`, to track which was the isolated asset associated to the borrowing configuration of the user.

BGD should also consider adding an authed function, executable only by the Pool Configurator or another admin role, that will decrease the reserve data `isolationModeTotalDebt` state value instead of increasing the ceiling. A "safe" solution would be to track in a `mapping` the missed `isolationModeTotalDebt` not decreased during the deficit creation and allow the admin to decrease the asset reserve attribute `isolationModeTotalDebt` only by that value and then reset it to zero.

BGD: Ack

The `isolationModeTotalDebt` only paints a very rough metric as it does not account for accrual of debt at all.
Therefore risk-providers need to accumulate info over all user positions to understand the `actualIsolationModeTotalDebt`.
Considering that, we think that adding another event won't help in regards to transparency, while at the same time adding gas on an already expensive operation.

[I-04] Consider documenting the following self-liquidating scenarios

Context

- [LiquidationLogic.sol]

Description

⚠️ After analyzing the code and the logic, we could consider the behavior of the liquidation process in this edge case scenario as if the self-liquidation borrower has executed a "normal" liquidation (with `receiveAToken = FALSE`) + a re-supply of the same amount of the seized collateral. With that in mind, we could consider the behavior of the current implementation of `LiquidationLogic` to work "as intended".

Let's assume that the `borrower` is self-liquidating itself and is executing the liquidation process with the input `receiveAToken` set to `true`.

When the above edge case is executed, instead of receiving the collateral as `underlying`, AAVE is "transferring" the `AToken` from the `borrower` to the `liquidator`, increasing the collateral value by the amount of `actualCollateralToLiquidate`.

Note that the `collateralAsset` is already enabled as a collateral; otherwise the liquidation process would have reverted when `validateLiquidationCall` was performed.

Let's also assume that the protocol is configured with the following settings:

- `MIN_BASE_MAX_CLOSE_FACTOR_THRESHOLD = 1000$`
- `MIN_LEFTOVER_BASE = 500$`

The `isDebtMoreThanLeftoverThreshold && isCollateralMoreThanLeftoverThreshold` is not respected

Let's assume we have this context

- $600 DAI debt
- $200 USDC collateral
- $100 USDT collateral

The borrower executes a liquidation to liquidate $200 DAI debt and seize $200 USDC as `aUSDC`.
The below code should prevent the creation of "dust" collateral or debt positions

```
if (
  vars.actualDebtToLiquidate < vars.userReserveDebt &&
  vars.actualCollateralToLiquidate + vars.liquidationProtocolFeeAmount <
  vars.userCollateralBalance
) {
  bool isDebtMoreThanLeftoverThreshold = ((vars.userReserveDebt -
vars.actualDebtToLiquidate) *
    vars.debtAssetPrice) /
    vars.debtAssetUnit >=
    MIN_LEFTOVER_BASE;

  bool isCollateralMoreThanLeftoverThreshold = ((vars.userCollateralBalance
-
    vars.actualCollateralToLiquidate) * vars.collateralAssetPrice) /
    vars.collateralAssetUnit >=
    MIN_LEFTOVER_BASE;

  require(
    isDebtMoreThanLeftoverThreshold &&
isCollateralMoreThanLeftoverThreshold,
    Errors.MUST_NOT_LEAVE_DUST
  );
}
```

But it's skipped, even if the `DAI` debt position will result in a "dust" position because the collateral is fully seized.

The problem is that because `liquidator == borrower` and `receiveAToken = TRUE` the final result of the liquidation is this:

- $400 DAI debt
- $200 USDC collateral
- $100 USDT collateral

with BOTH a "dust" position for DAI (debt) and USDC (collateral)

bad-debt full removal is triggered even if the borrower ends up with some `collateral > 0`

Let's assume we have this context

- $900 WETH debt
- $1000 DAI debt
- $600 USDC collateral

The liquidator liquidates $900 WETH of debt to seize $600 USDC collateral.
The current logic applies the "bad-debt-removal" logic in two parts of the code

```
bool hasNoCollateralLeft = vars.totalCollateralInBaseCurrency ==
  vars.collateralToLiquidateInBaseCurrency;

_burnDebtTokens(
  vars.debtReserveCache,
  debtReserve,
  userConfig,
  params.user,
  params.debtAsset,
  vars.userReserveDebt,
  vars.actualDebtToLiquidate,
  hasNoCollateralLeft
);

// >>> CODE to update isolated debt

if (params.receiveAToken) {
  _liquidateATokens(reservesData, reservesList, usersConfig,
collateralReserve, params, vars);
} else {
  _burnCollateralATokens(collateralReserve, params, vars);
}

// Transfer fee to treasury if it is non-zero
// >>> CODE to transfer fee to treasury

if (hasNoCollateralLeft && userConfig.isBorrowingAny()) {
  _burnBadDebt(reservesData, reservesList, userConfig, params.reservesCount,
params.user);
}
```

Both of them will be applied inside `_burnDebtTokens` (removal of bad debt + increase in the deficit) when `hasNoCollateralLeft == true` but as you can see, such value is initialized with "non-finalized" values.

After the execution, of the whole liquidation, the `borrower` will indeed have a `collateral > 0` because he's also the `liquidator` in this scenario.

Recommendations

BGD should consider adding documenting the above edge case scenarios in the already existing [`Aave-v3.3-features.md`] document. Integrators, developers and security researchers should be aware of these edge cases and why the liquidation logics act in this way.

StErMi: The recommendations have been implemented in the commit b21b164c6018152e6bc51794dc840779b3088884, specifically in the commit 48b187b8ea146b858549be27e14b73a2529a516c

[I-05] Suggestions and refactoring of the `ReserveLogic.executeEliminateDeficit` function

Context

- ReserveLogic.sol#L330-L386

Description

The current implementation of the `ReserveLogic.executeEliminateDeficit` function could be refactored to improve the DX and increase the safety of the logic for future modifications

1. `params.amount` should be sanity checked against the `msg.sender` (`Umbrella`) `AToken` balance. The logic will anyway revert during the `AToken.burn` execution, but it would use a less specific error. The error used, should be the one already used in the `validateWithdraw` logic: `Errors.NOT_ENOUGH_AVAILABLE_USER_BALANCE`
2. Following the same flow of other AAVE operations (supply, borrow, repay, ...) BGD should consider moving the `reserve.updateInterestRatesAndVirtualBalance` execution **before** any `burn` or `transfer`
3. While it's true that for reserves with the virtual account turned off it does not make sense to call a no-op `updateInterestRatesAndVirtualBalance`, the function could be called anyway in case of changes in the logic. In addition to that, the `updateInterestRatesAndVirtualBalance` also emits the `ReserveDataUpdated` event. Such function is already called in other part of the codebase where the liquidity added or taken is equal to zero.

Recommendations

BGD should consider implementing the recommendations suggested in the above section

StErMi: The recommendations have been implemented in the commit 91ed89b5462e0109013bf9a68fdc2c273a0c95a3

[I-06] `reserveCache.reserveLastUpdateTimestamp` is not synched with `reserve.lastUpdateTimestamp`

Context

- ReserveLogic.sol#L105
- ReserveLogic.sol#L113

Description

With AAVE 3.3 the `ReserveLogic.updateState` function is using the `reserveCache.reserveLastUpdateTimestamp` instead of `reserve.lastUpdateTimestamp` to early return if the indexes have been already calculated for the current block.

```diff
-    if (reserve.lastUpdateTimestamp == uint40(block.timestamp)) {
+    if (reserveCache.reserveLastUpdateTimestamp == uint40(block.timestamp))
{
       return;
     }
```

The change is correct and will bring a reduction in gas cost.

The `reserveCache.reserveLastUpdateTimestamp` is initialized with the `reserve.lastUpdateTimestamp` value inside the `cache()` function, and that function is **always** called **before** the execution of `updateState(...)`.
The `reserveCache.reserveLastUpdateTimestamp` value is currently **only** used inside `_updateIndexes`.

Given the above context and premises, the change can be considered safe with the current codebase but BGD should consider to always keep, unless there's a valid reason, those values always in sync to prevent de-sync issues if in the future `reserveCache.reserveLastUpdateTimestamp` is used **after** that the `reserve.lastUpdateTimestamp` value is updated at the end of the `updateState` flow.

Recommendations

BGD should consider to always keep in sync the `reserveCache` and `reserve` timestamp attributes

```
    function updateState(
      DataTypes.ReserveData storage reserve,
      DataTypes.ReserveCache memory reserveCache
    ) internal {
      // OTHER CODE

      //solium-disable-next-line
      reserve.lastUpdateTimestamp = uint40(block.timestamp);
  +    reserveCache.reserveLastUpdateTimestamp = uint40(block.timestamp);
    }
```

StErMi: The recommendations have been implemented in the commit [e2315b3db525f05a4dfa9bb6afead58249fa899f]()

[DISCUSSION] `vGHO` `_ghoUserState[user].accumulatedDebtInterest` is not resetted during bad-debt liquidation

`GHO` has this configuration

- `reserveFactor == 0` → no `aGHO` shares are accrued when `ReserveLogic._accrueToTreasury` is executed
- `liquidationProtocolFee == 0` → `GHO` cannot be supplies as collateral, so it means that it cannot be seized during liquidation. Having `liquidationProtocolFee > 0` would make no sense

The only way for the AAVE Protocol to receive fees from `GHO` borrowing is by tracking the borrower accrued `vGHO` debt inside `_ghoUserState[user].accumulatedDebtInterest`.

When someone repay the borrower debt via `repay` or `liquidate` operations, the `aGHO.handleRepayment` function is executed

```
  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);
    }
  }
```

`amount` will be the `GHO` amount that the repayer has transferred to the `AToken` contract itself (which is a `GHO` facilitator)
If the `onBehalfOf` user (the borrower in this case) has some debt interest accrued, the interest will be decreased via `_ghoVariableDebtToken.decreaseBalanceFromInterest` and the rest of `GHO` will be permanently burned.

This mean that `_ghoUserState[user].accumulatedDebtInterest` represents the full amount of `GHO` that should be received by the AAVE protocol as fees from the borrower.

Let's discuss two different scenarios on the new implementation of `LiquidationLogic`

Scenario 1) Liquidator repay part of the `GHO` debt, but part of it is burned as bad debt

Let's assume that

- there's not enough collateral to allow the full liquidation. Not all `GHO` is burned and bad-debt logic is executed
- borrower has `1000 GHO` debt
- borrower has `500 GHO` of accrued interest on the above debt

The liquidator should repay `1500 GHO` debt but given the collateral available can repay only `100 GHO`

The current liquidation logic will

- burn `1400 vGHO`
- transfer `100 GHO` from liquidator to the `aGHO` contract
- execute `aGHO.handleRepayment(liquidator, borrower, 100 GHO)`

`handleRepayment` will

- reduce the `_ghoUserState[user].accumulatedDebtInterest` by `100 GHO`, now it will be equal to `400 GHO`
- born `0 GHO` because `repaidAmount (100 GHO) < balanceFromInterest (500 GHO)`

The end results that the borrower has no more `vGHO` debt but has still a `400 GHO` debt accounted in the `_ghoUserState[user].accumulatedDebtInterest` which represents the amount of fees that still need to be given to the AAVE protocol

When in the future the same borrower will borrow more `GHO` from AAVE, the `_ghoUserState[user].accumulatedDebtInterest` won't start from zero (a clean state) but from `400 GHO` and will increase again in time.

As a consequence, when someone decreases the borrower `GHO` debt via repay or liquidate, the repaid amount won't be burned but used to decrease the AAVE protocol fees that have not been cleaned from the bad-debt scenario.

There are two options here

1. `_ghoUserState[user].accumulatedDebtInterest` is resetted to `0` during a bad-debt scenario (after correctly executing `handleRepayment` and paying the correct amount of debt)
2. if AAVE protocol thinks that those fees (the remaining accrued debt) is fair to be paid to the protocol, they could track them and redirect to the treasure when the `deficit` is repaid in `GHO`

Scenario 2) Liquidator repay part of the `DebtToken` debt and trigger the bad-debt flow. `GHO` bad-debt is burned via `_burnBadDebt`

This is more or less the same as "Scenario 1" with the main difference that here the `aGHO.handleRepayment` function is never called
All the accrued interest (AAVE fee) accounted in `_ghoUserState[user].accumulatedDebtInterest` remain untouched

The consequences are the same as the one reported in "Scenario 1"

Conclusions

When a bad-debt liquidation is triggered and `GHO` is involved, how should the `_ghoUserState[user].accumulatedDebtInterest` value be handled?

As already said, that value represents the `GHO` amount that the AAVE protocol should receive as fee (the amount of `GHO` received by `aGHO` that should not be burned when repaid from a debt).
`GHO`, unlike other assets, does not accrue the protocol fees as `aToken` shares because `GHO` cannot be supplied.
In a bad-debt liquidation, the protocol still gets the `aToken` shares accounted for when `reserve.updateState` is triggered and will be able to anyway mint those shares and then withdraw them when the deficit is repaid by Umbrella.

The same won't happen for `GHO` unless the borrower will start again to borrow `GHO` and then repay it. In this case, because `_ghoUserState[user].accumulatedDebtInterest` has not been resetted, the protocol will start to get their `GHO` back as fees, but it's not the very same thing.

Issue: `ReserveLogic.executeEliminateDeficit` burn more `GHO` than it should

The amount of `GHO` registered in the `reserve.deficit` value could include part of the fess that should be received by the AAVE Protocol if during the bad-debt liquidation the borrower had a `_ghoUserState[user].accumulatedDebtInterest` value greater than zero after `aGHO.handleRepayment`.

This mean that part of the `balanceWriteOff` should not be burned but remain in the `aGHO` contract as fees to the protocol.
The problem is that this won't happen because when

```
  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 handeling the repayment on behalf
of the protocol.
    address(this),
    balanceWriteOff
  );
```

is executed, the `address(this)` (the `POOL` itself), has `_ghoUserState[POOL].accumulatedDebtInterest` equal to zero and as a consequence, the whole `balanceWriteOff` amount of `GHO` will be burned.

Issue:

debt = 1000
accumulatedDebtInterest = 500

full bad-debt liquidation -> deficit = 1000
executeEliminateDeficit will recover and burn 1000 GHO

FIX

deficit -> OK
increase `POOL.accumulatedDebtInterest` by the `_ghoUserState[borrower].accumulatedDebtInterest`
reset `_ghoUserState[borrower].accumulatedDebtInterest`
when `executeEliminateDeficit` is executed, only part of the `GHO` will be burned, the remaining goes to the reserve as it should

Test

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

import {UpgradeTest, IERC20} from './UpgradeTest.t.sol';
import {DeploymentLibrary} from '../scripts/Deploy.s.sol';
import {Payloads} from './Payloads.sol';
import {IPool} from 'aave-v3-origin/contracts/interfaces/IPool.sol';
import {IPriceOracleGetter} from 'aave-v3-
origin/contracts/interfaces/IPriceOracleGetter.sol';
import {IERC20Detailed} from 'aave-v3-
origin/contracts/dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {OwnableUpgradeable} from 'openzeppelin-contracts-
upgradeable/contracts/access/OwnableUpgradeable.sol';
import {AaveV3Ethereum, AaveV3EthereumAssets} from 'aave-address-
book/AaveV3Ethereum.sol';
import {AaveSafetyModule} from 'aave-address-book/AaveSafetyModule.sol';

import 'forge-std/Test.sol';

interface VGHO {
  function getBalanceFromInterest(address user) external view returns
(uint256);

  function rebalanceUserDiscountPercent(address user) external;
}

contract MainnetTest is UpgradeTest('mainnet', 20930840) {
  using stdStorage for StdStorage;

  IPool pool = IPool(0x87870Bca3F3fD6335C3F4ce8392D69350B4fA4E2);

  // address holding excess funds for borrowing etc
  address whale = makeAddr('whale');
  // liquidator
  address liquidator = makeAddr('liquidator');
  // address getting liquidated
  address borrower = makeAddr('borrower');

  function _getPayload() internal virtual override returns (address) {
    return DeploymentLibrary._deployMainnet();
  }

  function _getDeployedPayload() internal virtual override returns (address)
```

```
{
    return Payloads.PROTO;
  }

  // test stub
  function test_yourTest() external {
    this.test_execution();
  }

  function _setupOracle(uint256 usdcPrice, uint256 daiPrice, uint256
ghoPrice) public {
    address oracle = pool.ADDRESSES_PROVIDER().getPriceOracle();
    vm.mockCall(
      oracle,
      abi.encodeWithSelector(
        IPriceOracleGetter.getAssetPrice.selector,
        AaveV3EthereumAssets.USDC_UNDERLYING
      ),
      abi.encode(usdcPrice)
    );
    vm.mockCall(
      oracle,
      abi.encodeWithSelector(
        IPriceOracleGetter.getAssetPrice.selector,
        AaveV3EthereumAssets.DAI_UNDERLYING
      ),
      abi.encode(daiPrice)
    );
    vm.mockCall(
      oracle,
      abi.encodeWithSelector(
        IPriceOracleGetter.getAssetPrice.selector,
        AaveV3EthereumAssets.GHO_UNDERLYING
      ),
      abi.encode(ghoPrice)
    );
  }

  function testLiquidateDAI() external {
    uint256 snapshotId = vm.snapshot();

    // SCENARIO 1) liquidator liquidate part of the DAI debt.
aGHO.handleRepayment is NEVER called because GHO bad debt is burned by
`_burnBadDebt`
    // supply 10k  USDC
    // borrow 1k   GHO
    // borrow 6.5k DAI
    (uint256 daiDeficit1, uint256 ghoDeficit1, uint256
ghoBalanceFromInterestBorrower1) = executeGHOTest(10_000e6, 6500e18,
1000e18, 30 days, AaveV3EthereumAssets.DAI_UNDERLYING);

    vm.revertTo(snapshotId);

    // SCENARIO 2) liquidator liquidate part of the GHO debt.
aGHO.handleRepayment is executed
    // supply 10k  USDC
    // borrow 6.5k   GHO
    // borrow 1k DAI
    // Note: I warped 10*365 days here because I needed to accrue far more
interest in the vGHO `accumulatedDebtInterest` attribute of the borrower
    // this to prove that even if `aGHO.handleRepayment` is executed, it
could not be enough to clean it to 0
    (uint256 daiDeficit2, uint256 ghoDeficit2, uint256
ghoBalanceFromInterestBorrower2) = executeGHOTest(10_000e6, 1000e18,
6500e18, 10*365 days, AaveV3EthereumAssets.GHO_UNDERLYING);

    console.log('daiDeficit1', daiDeficit1);
    console.log('ghoDeficit1', ghoDeficit1);
    console.log('ghoBalanceFromInterestBorrower1',
ghoBalanceFromInterestBorrower1);
    console.log('');
    console.log('daiDeficit2', daiDeficit2);
    console.log('ghoDeficit2', ghoDeficit2);
    console.log('ghoBalanceFromInterestBorrower2',
ghoBalanceFromInterestBorrower2);
  }

  function executeGHOTest(uint256 usdcSupplyAmount, uint256 daiBorrowAmount,
uint256 ghoBorrowAmount, uint256 warpTime, address assetToLiquidate)
internal returns (uint256, uint256, uint256) {
    this.test_execution();

    // make things easy 1 USDC = 1 DAI = 1 GHO = 1 USD
    _setupOracle(100_000_000, 100_000_000, 100_000_000);


    // Supply USDC, borrow GHO
    deal(AaveV3EthereumAssets.USDC_UNDERLYING, borrower, usdcSupplyAmount);
    vm.startPrank(borrower);
    IERC20(AaveV3EthereumAssets.USDC_UNDERLYING).approve(address(pool),
usdcSupplyAmount);
    pool.supply(AaveV3EthereumAssets.USDC_UNDERLYING, usdcSupplyAmount,
borrower, 0);
    pool.borrow(AaveV3EthereumAssets.GHO_UNDERLYING, ghoBorrowAmount, 2, 0,
borrower);
    pool.borrow(AaveV3EthereumAssets.DAI_UNDERLYING, daiBorrowAmount, 2, 0,
borrower);
    vm.stopPrank();

    // 30 days pass by
    vm.warp(block.timestamp + warpTime);

    // now USDC goes down to 0.1 USD
    _setupOracle(10_000_000, 100_000_000, 100_000_000);

    // liquidator liquidate the whole debt specified as input
    uint256 debtAccrued = IERC20(assetToLiquidate).balanceOf(borrower);

    vm.startPrank(liquidator);
    deal(assetToLiquidate, liquidator, debtAccrued);
    IERC20(assetToLiquidate).approve(address(pool), type(uint256).max);
    pool.liquidationCall(AaveV3EthereumAssets.USDC_UNDERLYING,
assetToLiquidate, borrower, type(uint256).max, false);
    vm.stopPrank();

    uint256 daiDeficit =
pool.getReserveDeficit(AaveV3EthereumAssets.DAI_UNDERLYING);
    uint256 ghoDeficit =
pool.getReserveDeficit(AaveV3EthereumAssets.GHO_UNDERLYING);
    uint256 ghoBalanceFromInterestBorrower =
VGHO(AaveV3EthereumAssets.GHO_V_TOKEN).getBalanceFromInterest(borrower);

    return (daiDeficit, ghoDeficit, ghoBalanceFromInterestBorrower);
  }


}
```