

Audit Report

2025-11-18

This report was generated by an AI smart contract auditor agent [SavantChat](#) and triaged by smart contract auditor [Igor Gulamov](#).

Scope

[e34aaaa829078df2317ef53173101eb4db6a5757](#)

Audit scope:

- src/contracts/protocol/libraries/logic/GenericLogic.sol
- src/contracts/protocol/libraries/logic/LiquidationLogic.sol
- src/contracts/protocol/libraries/logic/SupplyLogic.sol
- src/contracts/protocol/libraries/logic/ValidationLogic.sol
- src/contracts/protocol/libraries/types/DataTypes.sol
- src/contracts/protocol/pool/Pool.sol
- src/contracts/protocol/pool/PoolConfigurator.sol
- src/contracts/protocol/tokenization/AToken.sol
- src/contracts/protocol/tokenization/VariableDebtToken.sol
- src/contracts/protocol/tokenization/base/DebtTokenBase.sol
- src/contracts/protocol/tokenization/base/IncentivizedERC20.sol

Limitations

AI Powered Audit

This AI-powered audit by Savant.Chat is based on the smart contract code at the time of evaluation; any subsequent changes may require re-auditing. Due to AI limitations, such as potential hallucination, the audit may not detect all issues, and findings—especially fix recommendations—should be reviewed by a security specialist. Users are advised to implement additional checks like manual reviews and regular re-audits. The AI auditor is provided "as is," and neither Savant.Chat, nor its developers, nor its providers, shall be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising from its use or reliance on its results. Users must independently verify findings and assume all risks, as the AI may produce incorrect information.

Human Triage Scope

The human reviewer's involvement was strictly limited to the triage of findings generated by the AI agent and the validation of specific fixes implemented by the team. The reviewer **did not** conduct an independent manual audit or line-by-line code inspection. Consequently, any vulnerabilities or logical errors not detected by the AI agent (false negatives) are outside the scope of this human review. The verification provided confirms only that the reported AI findings were addressed; it does not certify the overall security of the codebase.

Disclaimer & Non-Warranty

This report represents the auditor's opinion based solely on the scope defined above. It does not guarantee that the code is bug-free or invulnerable to exploits. The triage process is limited to validating specific AI outputs and does not constitute a comprehensive security audit. The auditor assumes no liability for any direct or indirect losses, including loss of funds, arising from the use of this code. This report is not financial or investment advice.

Conclusion

We triaged findings from the AI audit and checked the fixes done by the team.

We consider state [ce5d9ef3b8ad30eb7dbbd3e16a998a1b4de8e954](#) to be secure against the specific vulnerabilities flagged by the AI auditor.

Issues

Severity: Medium



Issue 1: Incomplete Isolation Mode Check Allows Bypassing Collateral Restrictions

Severity

Medium

Team's comment

Confirmed.

Status

Fixed at [4143c2a59e0d901e4d05a8f4fb1791d998ce3b45](#)

Summary

A user can end up with two or more collateral assets while one of them is configured as an isolated asset (`debtCeiling > 0`). This contradicts the Aave V3 Isolation Mode specification, which mandates that a user supplying an isolated asset as collateral must not be able to enable any other collateral asset. In this multi-collateral state, `getIsolationModeState` always returns `false` (it only returns `true` when exactly one collateral is enabled and it is isolated), which then incorrectly allows enabling additional collateral via `validateUseAsCollateral`.

Vulnerability Details

Root causes:

- `UserConfiguration.getIsolationModeState` returns `true` only if the user has exactly one collateral and that asset has `debtCeiling > 0`. With multiple collaterals, it returns `false` unconditionally, even if one of the collaterals is isolated.
- `ValidationLogic.validateUseAsCollateral` relies on `getIsolationModeState` to block new collateral during isolation. When the user already has multiple collaterals, `getIsolationModeState` returns `false`, and the check passes, allowing more collateral to be enabled.
- Governance can set `debtCeiling` for an asset that already has suppliers if its base `liquidationThreshold == 0` (common when an asset is used as collateral only via eMode). The guard that enforces `checkNoSuppliers()` runs only when the base LT is non-zero. This enables post-hoc isolation of an already-supplied collateral and creates the multi-collateral + isolated state contrary to the Tech Paper.

Relevant code behavior:

- `getIsolationModeState` : returns `true` only when there is exactly one collateral asset and it has a non-zero `debtCeiling`.

- `validateUseAsCollateral` : permits enabling new collateral when `isolationModeActive == false` and the target `reserveConfig.getDebtCeiling() == 0` .
- `PoolConfigurator.setDebtCeiling` : skips the no-suppliers check when `oldDebtCeiling == 0` and base `liquidationThreshold == 0` , allowing isolation to be enabled even with active suppliers.

This directly violates Isolation Mode invariants defined in the Technical Paper (Section 4.2): an asset with `debtCeiling > 0` is isolated, and users supplying an isolated asset as collateral must not be able to enable any other assets as collateral.

Proof of Concept

Location: `tests/protocol/pool/Pool.Issue7.IsolationMode.t.sol`

The PoC includes two tests:

1. Happy path: isolation correctly blocks new collateral

- Configure WBTC with base LT = 0; enable it as eMode collateral; user enters eMode and enables WBTC as collateral; governance then sets WBTC `debtCeiling > 0` .
- Attempt to enable USDX as collateral reverts with `UserInIsolationModeOrLtvZero` .

Run:

```
forge test --match-test test_Issue7IsolationMode_happy --via-ir -vv
```

2. Bug path: two collaterals with one isolated, `getIsolationModeState` returns false

- Configure WBTC with base LT = 0; enable as eMode collateral.
- User supplies and enables WBTC and USDX as collateral (two collaterals).
- Governance sets WBTC `debtCeiling > 0` (isolation) without removing suppliers (allowed because base LT = 0).
- User supplies WETH and successfully enables it as collateral. Now the user has three collaterals with one isolated (WBTC), which violates the Isolation Mode spec. `getIsolationModeState` remains `false` because there are multiple collaterals, so the isolation check in `validateUseAsCollateral` is bypassed.

Run:

```
forge test --match-test test_Issue7IsolationMode_bug --via-ir -vv
```

Impact

- Protocol invariant violation: user can hold multiple collaterals while one of them is isolated, explicitly contradicting the Isolation Mode specification.
- Isolation guard is effectively disabled in multi-collateral states because `getIsolationModeState` returns `false` , allowing further collateral enablement.
- Risk governance assumptions are broken: setting an asset isolated post-hoc should not leave users in a state where adding new collateral is allowed.

Recommended Mitigation

- Replace the single-collateral isolation check with a comprehensive check: introduce `userConfig.hasIsolatedCollateral(reservesData, reservesList)` that iterates all user

collaterals and returns `true` if any have `debtCeiling > 0`. Update `validateUseAsCollateral` to use this predicate:

- `return (!hasIsolatedCollateral && reserveConfig.getDebtCeiling() == 0);`
- Strengthen `setDebtCeiling` safeguards to enforce `checkNoSuppliers()` regardless of base LT when transitioning an asset from non-isolated to isolated, or at least when the asset is presently used as collateral through any mechanism (including `eMode` overrides).

References

- `src/contracts/protocol/libraries/configuration/UserConfiguration.sol` — `getIsolationModeState`
- `src/contracts/protocol/libraries/logic/ValidationLogic.sol` — `validateUseAsCollateral`
- `src/contracts/protocol/pool/PoolConfigurator.sol` — `setDebtCeiling`
- Technical Paper: `docs/Aave_V3_Technical_Paper/Aave_V3_Technical_Paper.md` (Section 4.2 Isolation Mode invariants)

Severity: Low



Issue 2: Incorrect AvgLTV Calculation with Zero-LTV Collateral Prevents Borrowing

Severity

Low

Team's comment

Confirmed.

Status

Will not fix.

Summary

The `calculateUserAccountData` function in the `GenericLogic` library incorrectly calculates the average Loan-to-Value (LTV) when a user has deposited collateral with `LTV > 0`, and the protocol admin subsequently reconfigures one or more of those assets to `LTV = 0`. The inclusion of zero-LTV collateral value in the calculation's denominator artificially dilutes the resulting `avgLtv`, causing a reduction in borrowing power for users with otherwise valid collateral.

Vulnerability Details

The `calculateUserAccountData` function computes a user's overall risk profile, including their `avgLtv`. The calculation process exhibits a critical flaw in how mixed-LTV collateral is handled:

1. The function calculates `totalCollateralInBaseCurrency` by summing the value of all assets a user has supplied as collateral, regardless of their individual LTVs.
2. Simultaneously, it calculates a weighted LTV sum, stored in `vars.avgLtv`. However, this summation explicitly skips any collateral asset for which the LTV is zero (lines 120-124).
3. Finally, the function calculates the final `avgLtv` by dividing the weighted LTV sum (which excludes zero-LTV assets) by the `totalCollateralInBaseCurrency` (which includes zero-LTV assets) (lines 167-169).

```
// src/contracts/protocol/libraries/logic/GenericLogic.sol:120-124
if (vars.ltv == 0) {
    vars.hasZeroLtvCollateral = true;
} else {
    vars.avgLtv += vars.userBalanceInBaseCurrency * vars.ltv;
}
}
```

```
// src/contracts/protocol/libraries/logic/GenericLogic.sol:167-169
vars.avgLtv = vars.totalCollateralInBaseCurrency != 0
    ? vars.avgLtv / vars.totalCollateralInBaseCurrency
    : 0;
```

When a user has deposited collateral assets and an admin reconfigures one of them to LTV = 0 (while maintaining liquidation threshold > 0), the denominator becomes disproportionately large compared to the numerator. This results in an `avgLtv` that is severely diluted and does not accurately reflect the user's actual borrowing power derived from their non-zero-LTV collateral.

The diluted `avgLtv` is then used to calculate `availableBorrowsBase` via `calculateAvailableBorrows` :

```
// src/contracts/protocol/libraries/logic/GenericLogic.sol:193-206
function calculateAvailableBorrows(
    uint256 totalCollateralInBaseCurrency,
    uint256 totalDebtInBaseCurrency,
    uint256 ltv
) internal pure returns (uint256) {
    uint256 availableBorrowsInBaseCurrency =
totalCollateralInBaseCurrency.percentMulFloor(ltv);

    if (availableBorrowsInBaseCurrency <= totalDebtInBaseCurrency) {
        return 0;
    }

    availableBorrowsInBaseCurrency = availableBorrowsInBaseCurrency -
totalDebtInBaseCurrency;
    return availableBorrowsInBaseCurrency;
}
```

Because the `ltv` parameter is diluted, `availableBorrowsInBaseCurrency` is calculated as lower than it should be, reducing the user's borrowing power.

Proof of Concept

Location: `tests/protocol/pool/Pool.Issue1.AvgLtv.t.sol`

The PoC demonstrates two scenarios:

Scenario 1: User deposits \$1,000 worth of WBTC (LTV 75%)

- Total Collateral: \$999.99981
- Available Borrows: \$749.9998575

- Maximum debt successfully borrowed: **\$749.999857**
- Avg LTV: 75%

Scenario 2: User deposits \$1M worth of USDX (LTV 75%) + \$1K worth of WBTC (LTV 75%), then admin reconfigures USDX to LTV 0%

- Before reconfiguration:
 - Total Collateral: \$1,000,999.99981
 - Available Borrows: \$750,749.9998575
 - Avg LTV: 75%
- After reconfiguration to LTV 0%:
 - Total Collateral: \$1,000,999.99981 (unchanged)
 - Available Borrows: \$700.69999986
 - **Maximum debt successfully borrowed: \$700.699999**
 - Avg LTV: 0%

Impact: The user loses approximately **\$49.30** (~6.6%) of borrowing power, despite having \$1,000 worth of valid 75% LTV collateral.

To run the PoC:

```
forge test --match-test test_Issue1_zeroLtvCollateralDilutesAvgLtv --via-ir -vv
```

Impact

1. **Loss of Borrowing Power:** Users with mixed collateral (including assets reconfigured to LTV 0%) experience a reduction in their borrowing capacity, even though they have valid collateral with LTV > 0.
2. **Denial of Service:** In extreme cases where the zero-LTV collateral significantly outweighs the non-zero-LTV collateral, users may be completely unable to borrow, despite having legitimate borrowing power.
3. **Unexpected Behavior:** This behavior contradicts the intended design described in the Aave V3 Technical Paper (Section 4.3), which states that 0 LTV assets should not provide borrowing power, but does not state that they should *reduce* the borrowing power of other assets.
4. **User Trust:** Users may lose confidence in the protocol if their borrowing power unexpectedly decreases without them taking any action (due to admin reconfiguration).

Recommended Mitigation

The `avgLtv` calculation should use a denominator that only includes the value of collateral with a non-zero LTV. This can be achieved by tracking the value of zero-LTV collateral and subtracting it from the total before the final division:

```
// src/contracts/protocol/libraries/logic/GenericLogic.sol

CalculateUserAccountDataVars memory vars;
uint256 zeroLtvCollateralInBaseCurrency = 0;

// ... existing code ...
```

```

if (vars.ltv == 0) {
    vars.hasZeroLtvCollateral = true;
    zeroLtvCollateralInBaseCurrency += vars.userBalanceInBaseCurrency;
} else {
    vars.avgLtv += vars.userBalanceInBaseCurrency * vars.ltv;
}

// ... existing code ...

unchecked {
    uint256 borrowableCollateralInBaseCurrency = vars.totalCollateralInBaseCurrency -
        zeroLtvCollateralInBaseCurrency;
    vars.avgLtv = borrowableCollateralInBaseCurrency != 0
        ? vars.avgLtv / borrowableCollateralInBaseCurrency
        : 0;
    vars.avgLiquidationThreshold = vars.totalCollateralInBaseCurrency != 0
        ? vars.avgLiquidationThreshold / vars.totalCollateralInBaseCurrency
        : 0;
}

```

This ensures that `avgLtv` accurately reflects the weighted average LTV of only those assets that contribute to borrowing power, preventing the dilution effect.

Affected Code

File: `src/contracts/protocol/libraries/logic/GenericLogic.sol`

Lines: 120-124, 167-169

Function: `calculateUserAccountData`



Issue 3: Outdated Documentation Regarding eMode Borrowing Priority

Severity

Low

Team's comment

Confirmed.

Status

Fixed at [cc7bc2c42e2dd96b381654cad83e57486f15ad9f](https://github.com/aave/aave-protocol-v3/pull/1549)

Summary

The NatSpec documentation comment in `IPoolConfigurator.sol` contains outdated and misleading information about the priority relationship between `eMode.borrowable` and `reserve.borrowingEnabled`. The comment states that `eMode.borrowable` always has less priority than `reserve.borrowable`, but this contradicts both the actual implementation in v3.6+ and the official changelog documentation.

Vulnerability Details

Outdated Documentation Comment

The interface `IPoolConfigurator.sol` contains the following misleading comment:

```
// src/contracts/interfaces/IPoolConfigurator.sol:447
/**
 * @notice Enables/disables an asset to be borrowable in a selected eMode.
 * - eMode.borrowable always has less priority then reserve.borrowable
 * @param asset The address of the underlying asset of the reserve
 * @param categoryId The eMode categoryId
 * @param borrowable True if the asset should be borrowable in the given eMode
category, false otherwise.
 */
function setAssetBorrowableInEMode(address asset, uint8 categoryId, bool borrowable)
external;
```

Actual Implementation Behavior

The implementation in `ValidationLogic.validateBorrow` demonstrates that for eMode users (when `userEModeCategory != 0`), only the `eMode.borrowable` bitmap is checked, and `reserve.borrowingEnabled` is **completely ignored**:

```
// src/contracts/protocol/libraries/logic/ValidationLogic.sol:148-158
if (params.userEModeCategory != 0) {
    require(
        EModeConfiguration.isReserveEnabledOnBitmap(
            eModeCategories[params.userEModeCategory].borrowableBitmap,
            reservesData[params.asset].id
        ),
        Errors.NotBorrowableInEMode()
    );
} else {
    require(vars.borrowingEnabled, Errors.BorrowingNotEnabled());
}
```

This means:

- **For eMode users (`userEModeCategory != 0`):** Only `eMode.borrowable` is checked; `reserve.borrowingEnabled` has **no effect**.
- **For non-eMode users (`userEModeCategory == 0`):** Only `reserve.borrowingEnabled` is checked.

Official Documentation Confirms Implementation

The Aave v3.6 changelog explicitly documents this behavior as an intentional design change:

Change: In Aave v3.6, `lt`, `ltv`, and `borrowingEnabled` are now fully decoupled — the configuration for `eMode = 0` no longer affects any non-default efficiency modes (`eMode ≠ 0`).

Borrowable-only in eMode: Previously, to make an asset borrowable only inside an eMode, teams had to enable it as borrowable outside of eMode as well. This meant assets were borrowable globally even when there was only a specific use case within an eMode. **Now assets can be borrowable exclusively within specific eModes.**

The v3.6 properties documentation confirms:

Borrow rules: An asset can be borrowed inside an eMode if the following condition is met:

- `eMode.borrowable` is **enabled**

If this condition is met, the asset can be borrowed. In all other cases, the asset **cannot** be borrowed.

There is **no mention** of `reserve.borrowingEnabled` affecting eMode borrows.

Contradiction Analysis

The comment in `IPoolConfigurator.sol` incorrectly states:

- ❌ "eMode.borrowable always has less priority then reserve.borrowable"

The actual behavior (confirmed by implementation and v3.6 changelog):

- ✅ For eMode ≠ 0: `eMode.borrowable` is the **only** check; `reserve.borrowingEnabled` is **ignored**
- ✅ This means `eMode.borrowable` effectively has **greater priority** (or rather, exclusive control) for eMode users

Impact

1. **Developer Confusion:** Developers reading the interface documentation will be misled about the actual behavior of the protocol, potentially leading to incorrect integration assumptions.
2. **Incorrect Risk Assessments:** Risk teams and auditors may incorrectly assume that `setReserveBorrowing(asset, false)` will disable borrowing for all users, including eMode users, when in fact it only affects non-eMode users.
3. **Documentation Inconsistency:** The codebase documentation should accurately reflect the implementation, especially after the v3.6 changes that explicitly decoupled these configurations.
4. **Maintenance Issues:** Outdated documentation creates technical debt and makes future maintenance more difficult.

Recommended Mitigation

Update the NatSpec comment in `IPoolConfigurator.sol` to accurately reflect the v3.6+ behavior:

```
/**
 * @notice Enables/disables an asset to be borrowable in a selected eMode.
 * @dev In Aave v3.6+, eMode borrowing configuration is fully decoupled from reserve
configuration.
 * @dev For users in eMode (eMode ≠ 0), only the eMode.borrowable bitmap is checked;
 *       reserve.borrowingEnabled has no effect on eMode users.
 * @dev For non-eMode users (eMode = 0), only reserve.borrowingEnabled is checked.
 * @param asset The address of the underlying asset of the reserve
 * @param categoryId The eMode categoryId
 * @param borrowable True if the asset should be borrowable in the given eMode
category, false otherwise.
 */
function setAssetBorrowableInEMode(address asset, uint8 categoryId, bool borrowable)
external;
```

Affected Code

File: `src/contracts/interfaces/IPoolConfigurator.sol`

Line: 447

Related Implementation: `src/contracts/protocol/libraries/logic/ValidationLogic.sol:148-158`

Reference Documentation:

- `docs/3.6/Aave-v3.6-features.md:18-49`
- `docs/3.6/Aave-v3.6-properties.md:47-53`

Note

This is a documentation issue, not a code vulnerability. The implementation correctly follows the v3.6 design specification, but the interface documentation comment is outdated and misleading.

 **Issue 4: Stale LTV Can Be Restored After Re-Freezing an Asset**

Severity

Low

Team's comment

Confirmed. The issue takes assumption about governance intent. If the governance would want to remove pending they could call `configureReserveAsCollateral` with `ltv = 0` to overwrite the pending.

Status

Acknowledged.

Summary

Freezing a reserve sets its LTV to 0 and snapshots the current LTV in `_pendingLtv`. If a reserve is later unfrozen (leaving live LTV = 0) and then frozen again, the helper `_setReserveLtvzero` returns early when `currentLtv == 0` and does not refresh `_pendingLtv`. As a result, any old non-zero snapshot persists. A subsequent restoration via `setReserveLtvzero(asset, false)` will re-apply that stale value, silently overriding governance's intent to keep the asset at LTV = 0.

This creates a state-machine inconsistency where a safe administrative sequence can reintroduce a higher LTV without an explicit parameter change.

Vulnerability Details

- `PoolConfigurator.setReserveFreeze(asset, true)` calls `_setReserveLtvzero(asset, true, currentConfig)`.
- In `_setReserveLtvzero`, the branch for `ltvZero == true` returns immediately when `currentConfig.getLtv() == 0`, skipping the write to `_pendingLtv[asset]`.
- If the asset was unfrozen and governance decided to keep LTV = 0 (but did not send a configuration tx that clears `_pendingLtv`), re-freezing leaves the old snapshot intact.
- Restoring LTV later (`setReserveLtvzero(asset, false)`) then applies that stale value.

Relevant design context is described in the Aave V3 Technical Paper (granular borrowing power control and admin roles): `docs/Aave_V3_Technical_Paper/Aave_V3_Technical_Paper.md`.

Impact

- Risk parameters can be unintentionally rolled back to an older, higher LTV after routine freeze cycles.
- This may contradict governance's updated risk stance and increase insolvency risk.

Proof of Concept (PoC)

Two tests demonstrate both the safe flow and the bug.

Test file: `tests/protocol/pool/Pool.Issue17.StaleLtv.t.sol`

1. Happy path (snapshot cleared explicitly):

- Freeze → Unfreeze (LTV remains 0).
- Admin explicitly sets live LTV to 0 via `configureReserveAsCollateral`, which clears `_pendingLtv`.
- Re-freeze → Unfreeze → Attempt to restore snapshot reverts, confirming no stale value persists.

2. Bug path (stale snapshot restored):

- Freeze (snapshot taken) → Unfreeze (live LTV 0, snapshot stays).
- Governance chooses to keep LTV 0 but sends no transaction to clear the snapshot.
- Re-freeze (early returns; snapshot remains stale) → Unfreeze → `setReserveLtvzero(asset, false)` restores the old non-zero LTV.

Run the PoC:

```
forge test --match-test Issue17 --via-ir -vv
```

Observed results:

- `test_Issue17_staleLtv_happy` passes with no stale restoration.
- `test_Issue17_staleLtv_bug` shows the stale LTV restored to the pre-freeze value.

Recommended Mitigation

Refresh or clear `_pendingLtv` on every `ltvZero == true` transition, even when current LTV is already 0. One minimal approach is to remove the early return and always set `_pendingLtv[asset]` to the current LTV (possibly 0), only SSTORE the live LTV to 0 if it is non-zero. Alternatively, clear `_pendingLtv` when unfreezing if the intent is to keep LTV at 0.

References

- Aave V3 Technical Paper: `docs/Aave_V3_Technical_Paper/Aave_V3_Technical_Paper.md`
- Implementation: `src/contracts/protocol/pool/PoolConfigurator.sol`
- PoC: `tests/protocol/pool/Pool.Issue17.StaleLtv.t.sol`



Issue 5: Inadequate Collateral Status Check Bypasses Isolation Mode Safeguards

Severity

Low

Team's comment

Confirmed.

Status

Fixed at [4143c2a59e0d901e4d05a8f4fb1791d998ce3b45](#)

Summary

`PoolConfigurator.setDebtCeiling` relies exclusively on the reserve's base liquidation threshold to decide whether `_checkNoSuppliers` must run before a debt ceiling is first enabled. Assets that are configured with `liquidationThreshold = 0` at the reserve level but whitelisted as collateral inside an

eMode category can still be actively used as collateral, exactly as described in the Aave V3 Technical Paper (§3.1 “E-Mode”). Because the base threshold remains zero, the configurator skips the supplier check, allowing an admin to move a live market with suppliers and debt into isolation mode while leaving `isolationModeTotalDebt` at zero. The debt ceiling invariant (`isolationModeTotalDebt <= debtCeiling`) is therefore broken and the risk cap defined by governance is rendered ineffective.

Vulnerability Details

1. `PoolConfigurator.setDebtCeiling` guards `_checkNoSuppliers` behind the condition `currentConfig.getLiquidationThreshold() != 0 && oldDebtCeiling == 0`.

```
// src/contracts/protocol/pool/PoolConfigurator.sol:306-317
uint256 oldDebtCeiling = currentConfig.getDebtCeiling();
if (currentConfig.getLiquidationThreshold() != 0 && oldDebtCeiling == 0) {
    _checkNoSuppliers(asset);
}
currentConfig.setDebtCeiling(newDebtCeiling);
_pool.setConfiguration(asset, currentConfig);
```

2. The base liquidation threshold accessor reads only the reserve configuration map and does not consider eMode overrides.

```
// src/contracts/protocol/libraries/configuration/ReserveConfiguration.sol:105-114
function getLiquidationThreshold(
    DataTypes.ReserveConfigurationMap memory self
) internal pure returns (uint256) {
    return (self.data & LIQUIDATION_THRESHOLD_MASK)
        >> LIQUIDATION_THRESHOLD_START_BIT_POSITION;
}
```

3. When a user joins an eMode category, `ValidationLogic.getUserReserveLtv` (and the associated liquidation threshold logic) substitutes the category parameters whenever the reserve id is enabled in the eMode collateral bitmap. An asset with base LT/LTV equal to zero can therefore become full collateral for users while they remain in that eMode.
4. `IsolationModeLogic.increaseIsolatedDebtIfIsolated` only increments `isolationModeTotalDebt` during borrow operations executed after isolation mode has already been activated. Any pre-existing debt is ignored.

Combining these behaviors, an admin can set a debt ceiling for an asset that has active suppliers and debt (because it is used as collateral inside an eMode) without triggering `_checkNoSuppliers`. The asset immediately becomes isolated with `isolationModeTotalDebt == 0`, yet users continue to carry non-zero debt that is now excluded from the ceiling accounting. Further borrows in isolation mode are checked against the stale zero value, so the governance-imposed cap no longer constrains risk.

Proof of Concept

Location: `tests/protocol/pool/Pool.Issue22.DebtCeilingEMode.t.sol`

The PoC contains two scenarios:

- `test_Issue22_debtCeilingChecksSuppliers_happy` — baseline where the reserve keeps a non-zero liquidation threshold. The function correctly reverts with `Errors.ReserveLiquidityNotZero` when suppliers exist, demonstrating the intended behavior.
- `test_Issue22_debtCeilingBypassesSuppliers_bug` — configures WBTC with base LT = 0, adds it as collateral inside a dedicated eMode category, supplies and borrows against it, then calls `setDebtCeiling`. Because the base LT remains zero, `_checkNoSuppliers` is skipped, the call succeeds, `isolationModeTotalDebt` stays zero, yet Alice continues to hold debt (`totalDebtBase > 0`).

Run the PoC with:

```
forge test --match-test Issue22 --via-ir -vv
```

Impact

- **Bypasses isolation risk limits:** Debt ceilings can be enabled on actively used collateral assets without zeroing out existing positions, leaving `isolationModeTotalDebt` at zero while real debt remains outstanding. Borrowers can continue drawing isolation-mode liquidity far beyond the governance-defined cap.
- **Protocol insolvency risk:** Isolation mode is the primary control for onboarding higher-risk assets. Breaking the accounting invariant exposes the protocol to defaults that isolation mode is meant to mitigate.
- **Admin error amplification:** A single misconfigured call by a privileged actor can silently remove a critical risk control without immediate visibility in on-chain accounting.

Recommended Mitigation

Ensure `_checkNoSuppliers` executes whenever a debt ceiling is introduced. Two safe approaches:

- Conservative: always call `_checkNoSuppliers(asset)` when `oldDebtCeiling == 0`, regardless of the base liquidation threshold.
- Targeted: augment the condition to also detect collateral enablement through eMode (e.g., scan eMode collateral bitmaps and verify that the asset is not enabled there with a non-zero category liquidation threshold/lvtzero settings).

Either approach prevents the isolation flag from being applied while suppliers or debt are outstanding.

Affected Code

- `src/contracts/protocol/pool/PoolConfigurator.sol` — `setDebtCeiling`
- `src/contracts/protocol/libraries/configuration/ReserveConfiguration.sol` — `getLiquidationThreshold`
- `src/contracts/protocol/libraries/logic/ValidationLogic.sol` — `getUserReserveLtv`
- `src/contracts/protocol/libraries/logic/IsolationModeLogic.sol` — `increaseIsolatedDebtIfIsolated`



Issue 6: Incorrect Allowance Accounting in Credit Delegation Due to Flawed Rounding Calculation

Severity

Low

Team's comment

Confirmed.

Status

Fixed at [0e4bedf6ef251e01b951ac812973f054beafc6b9](#)

Summary

In credit delegation borrows (`user != onBehalfOf`), the allowance consumed is computed using the delegatee's current scaled variable-debt balance, while the actual debt is minted to the delegator. Because `vToken` arithmetic rounds in favor of the protocol (round up), the true unscaled debt increase depends on the delegator's scaled balance. Using the delegatee's balance to estimate consumption can under-consume allowance (typically by 1 wei) relative to the delegator's new debt, allowing a malicious delegatee to accumulate delegator debt exceeding the approved allowance via many small borrows.

Vulnerability Details

Borrow path forwards both `user` (delegatee / `msg.sender`) and `onBehalfOf` (delegator):

```
function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
) public virtual override {
    BorrowLogic.executeBorrow(
        _reserves,
        _reservesList,
        _eModeCategories,
        _usersConfig[onBehalfOf],
        DataTypes.ExecuteBorrowParams({
            asset: asset,
            interestRateStrategyAddress: RESERVE_INTEREST_RATE_STRATEGY,
            user: _msgSender(),
            onBehalfOf: onBehalfOf,
            amount: amount,
            interestRateMode: DataTypes.InterestRateMode(interestRateMode),
            referralCode: referralCode,
            releaseUnderlying: true,
            oracle: ADDRESSES_PROVIDER.getPriceOracle(),
            userEModeCategory: _usersEModeCategory[onBehalfOf],
            priceOracleSentinel: ADDRESSES_PROVIDER.getPriceOracleSentinel()
        })
    );
}
```

Debt token `mint` calculates the allowance consumption delta based on the delegatee's scaled balance, while minting debt to the delegator:

```
function mint(
    address user,
    address onBehalfOf,
```

```

uint256 amount,
uint256 scaledAmount,
uint256 index
) external virtual override onlyPool returns (uint256) {
    uint256 scaledBalanceOfUser = super.balanceOf(user);

    if (user != onBehalfOf) {
        _decreaseBorrowAllowance(
            onBehalfOf,
            user,
            amount,
            (scaledBalanceOfUser + scaledAmount).getVTokenBalance(index) -
            scaledBalanceOfUser.getVTokenBalance(index)
        );
    }
    _mintScaled({
        caller: user,
        onBehalfOf: onBehalfOf,
        amountScaled: scaledAmount,
        index: index,
        getTokenBalance: TokenMath.getVTokenBalance
    });
    return scaledTotalSupply();
}

```

Allowance decrease semantics only require `oldAllowance >= amount` but spend up to `correctedAmount` (capped by current allowance):

```

function _decreaseBorrowAllowance(
    address delegator,
    address delegatee,
    uint256 amount,
    uint256 correctedAmount
) internal {
    uint256 oldBorrowAllowance = _borrowAllowances[delegator][delegatee];
    if (oldBorrowAllowance < amount) {
        revert InsufficientBorrowAllowance(delegatee, oldBorrowAllowance, amount);
    }

    uint256 consumption = oldBorrowAllowance >= correctedAmount
        ? correctedAmount
        : oldBorrowAllowance;
    uint256 newAllowance = oldBorrowAllowance - consumption;

    _borrowAllowances[delegator][delegatee] = newAllowance;
}

```

Variable debt rounding rules drive the mismatch:

```

function getVTokenMintScaledAmount(
    uint256 amount,

```

```

    uint256 variableBorrowIndex
) internal pure returns (uint256) {
    return amount.rayDivCeil(variableBorrowIndex);
}

```

```

function getVTokenBalance(
    uint256 scaledAmount,
    uint256 variableBorrowIndex
) internal pure returns (uint256) {
    return scaledAmount.rayMulCeil(variableBorrowIndex);
}

```

As a result, the delegator's unscaled debt increase can be `amount + 1` while allowance spent is only `amount`, allowing a 1 wei drift per borrow. Repeating many small borrows accumulates excess delegator debt beyond the approved allowance.

Proof of Concept

Location: `tests/protocol/tokenization/VariableDebtToken.Issue33.CreditDelegation.t.sol`

The PoC provides two tests:

- Happy path: zero balances, `index = RAY` → allowance spent equals debt increase.
- Bug path: delegatee preloaded with `scaledBalance = 1`, `index = RAY + 1` → debt increase is `amount + 1` while allowance spent is `amount`.

Key assertions:

```

assertEq(allowanceSpent, amount, 'happy: allowance should match requested amount');
assertEq(debtIncrease, amount, 'happy: debt increase should equal requested amount');

```

```

assertEq(allowanceSpent, amount, 'bug: allowance spent capped at approved amount');
assertEq(debtIncrease, amount + 1, 'bug: debt increase rounded up by 1 wei');
assertGt(debtIncrease, allowanceSpent, 'bug: debt increase should exceed allowance spent');

```

Run the PoC:

```

forge test --match-test Issue33 --via-ir -vv

```

Impact

- Delegator allowance is under-consumed relative to true debt minted, enabling a delegatee to borrow beyond the configured allowance via repeated small borrows.
- Each step is typically a +1 wei drift; economic impact per call is small but can accumulate over many transactions.
- Constraints: requires active delegatee control, many transactions, and specific rounding conditions; hence classified as Low severity.

Recommended Mitigation

Compute the corrected allowance consumption using the delegator's scaled balance, since the delegator's account receives the new debt and thus determines the rounding outcome. Conceptually:

```
uint256 scaledBalanceOfOnBehalfOf = super.balanceOf(onBehalfOf);
uint256 debtIncrease = (scaledBalanceOfOnBehalfOf +
scaledAmount).getVTokenBalance(index)
- scaledBalanceOfOnBehalfOf.getVTokenBalance(index);
_decreaseBorrowAllowance(onBehalfOf, user, amount, debtIncrease);
```

This aligns with the v3.5 documentation goal to burn the exact consumed allowance when available and ensures consumption tracks the actual delegator debt increase. Backwards compatibility is preserved via the existing cap in `_decreaseBorrowAllowance`.

Affected Code

- `src/contracts/protocol/tokenization/VariableDebtToken.sol` — `mint` allowance delta computed from `user` instead of `onBehalfOf`.
- `src/contracts/protocol/tokenization/base/DebtTokenBase.sol` — semantics confirm `amount` is the minimum check and `correctedAmount` is the intended precise consumption (capped by current allowance).