Security Assessment Report

Aave v3.6

November-2025

Prepared for:
Aave DAO

Code developed by:
BORED GHOSTS DEVELOPING

certora

aave

Table of contents

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Aave v3.6	Github Repo	Initial - 4143c2a Final - 4df386b	EVM

Project Overview

This document describes the verification of **Aave v3.6** code using manual code review. The work was undertaken from **October 27** to **November 3.**

The following contracts are considered in scope for this review:

- `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`

The team performed a manual audit of all the solidity contracts. Issues discovered during the review are listed in the following pages.

Protocol Overview

Aave v3.6.0 introduces major improvements to eMode configuration, allowing `lt`, `ltv`, and `borrowingEnabled` to be independently defined per eMode with the new `ltvZeroBitmap`, enabling granular risk control.

Automatic collateral enablement was removed from aToken transfers, liquidations, and isolated collaterals to save gas and simplify logic.

The release adds `renounceAllowance` and `renounceDelegation` functions for revoking approvals, aligns event emissions with OpenZeppelin's ERC20 standard for gas efficiency, and soft-deprecates eMode category labels.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	1	1	-
Informational	4	4	2
Total	5	5	2

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

aave

Detailed Findings

ID	Title	Severity	Status
L-01	Attacker Can Disable Victim's Auto-Collateral by Sending Minimal aToken	Low	Acknowledged
I-01	Excessive loop range over eMode IDs increases gas usage	Informational	Acknowledged
I-02	Inefficient categoryId != 0 check inside loop	Informational	Acknowledged
I-03	Validate HF / LTV0 comment is vague	Informational	Fixed
I-04	Misleading error messages when categoryId == 0 in validateSetUserEMode()	Informational	Fixed

Low Severity Issues

L-01. Attacker Can Disable Victim's Auto-Collateral by Sending Minimal aToken

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
File : SupplyLogic.sol	Status: Acknowledged	

Details:

When a user supplies an asset for the first time, the protocol auto-enables it as collateral if the recipient had zero aToken balance prior to the mint (isFirstSupply == true). Since v3.6 the protocol no longer auto-enables collateral on aToken transfers. Because the supply auto-enable still uses the "prior aToken balance == 0" heuristic, an attacker can front-run a victim's supply() by transferring a tiny amount of the aToken (e.g., 1 wei) to the victim. That makes IAToken.mint() return isFirstSupply == false and prevents the protocol from auto-enabling collateral for the victim.

Impact
An attacker cannot directly steal funds, but can easily prevent a victim's newly supplied tokens from being treated as collateral.

This breaks common front-end flows where users supply and then immediately borrow (often in a follow-up transaction). Those subsequent borrow attempts can fail, forcing users to submit an extra transaction to enable collateral and pay additional gas.

The same fragility undermines integrations: smart contracts, position managers, and batched strategies that assumed first-supply auto-enable can fail or revert, causing operational issues or missed opportunities.

The attack is cheap and scalable, a tiny aToken transfer is sufficient, so an attacker could target multiple addresses or high-value UX paths at low cost, if motivated.

In most cases the outcome is an inconvenience, but in time-sensitive or integration-heavy scenarios (for example, certain arbitrage or liquidation workflows) the inability to enable collateral could lead to meaningful disruptions rather than just annoyance.

Recommendation:

The protocol should make collateral enablement explicit and atomic. Add an enableAsCollateral boolean to supply() so the caller can request collateral be enabled within the same transaction that mints aTokens, removing the raceable "zero balance" heuristic. For safety, require msg.sender == onBehalfOf when enableAsCollateral == true, so only self-supplies can trigger atomic auto-enable and third parties cannot grief by pre-funding.

To support relayers, extend supplyWithPermit(...) to accept the same enableAsCollateral flag plus a user-signed enable-permit (an EIP-712 authorization containing the relevant parameters, a nonce, and a deadline); with a valid enable-permit the pool may allow enableAsCollateral == true even when msg.sender != onBehalfOf, enabling relayers to perform atomic supply+enable on behalf of users.

This removes the race condition inherent in the prior balance-based heuristic while preserving both direct-user and relayer-driven workflows.

Customer's response: Acknowledged, but won't fix, since addressing this would break existing functionality that allows users to deposit on behalf of others and supports auto-enabling collateral for the initial supply.

Informational Issues

I-01. Excessive loop range over eMode IDs increases gas usage

Details:
In PoolConfigurator.setReserveFreeze, the code loops up to 255 eMode IDs:

```javascript
for (uint256 j = 1; j <= type(uint8).max; j++) {
  collateralEnabledBitmap =
_pool.getEModeCategoryCollateralBitmap(uint8(j));
  if (EModeConfiguration.isReserveEnabledOnBitmap(collateralEnabledBitmap,
reserveData.id)) {
    ltvzeroBitmap = _pool.getEModeCategoryLtvzeroBitmap(uint8(j));
    _setEmodeLtvZero(ltvzeroBitmap, asset, reserveData.id, uint8(j), true);
  }
}
```

Real deployments use far fewer eModes (usually <10), so iterating to 255 adds unnecessary gas overhead.

Recommendation:
Limit the loop range, either iterate up to the highest configured categoryId or use a fixed conservative cap (e.g., MAX_EMODES = 32) that is on the lower side.

Customer's response: Acknowledged. There are already around 28 eModes on mainnet. Setting an arbitrary limit seems very dangerous. Iterating to "highest configured" is not possible as the protocol does not maintain a counter and allows gaps.

I-02. Inefficient categoryId != 0 check inside loop

Details:
In validateSetUserEMode(), categoryId != 0 is evaluated on every iteration unnecessarily:

```
JavaScript

// ensure that in the target eMode (even if it's eMode 0), the assets can still be
// borrowed and be used as collateral

    unchecked {
      while (unsafe_cachedUserConfig != 0) {
        .
        .
        if (isBorrowed) {
          require(
            categoryId != 0
EModeConfiguration.isReserveEnabledOnBitmap(eModeCategory.borrowableBitmap, i)
              : reservesData[reservesList[i]].configuration.getBorrowingEnabled(),
            Errors.InvalidDebtInEmode(reservesList[i])
          );
        }
        .
        .
        }
        ++i;
      }
    }
```

This adds redundant branching overhead per iteration and slightly impacts gas.

Recommendation:
Evaluate once before the loop and reuse:

```
JavaScript

bool inEMode = categoryId != 0;
```

```
...
if (isBorrowed) {
  require(
    inEMode
      ? EModeConfiguration.isReserveEnabledOnBitmap(eModeCategory.borrowableBitmap,
i)
      : reservesData[reservesList[i]].configuration.getBorrowingEnabled(),
    Errors.InvalidDebtInEmode(reservesList[i])
  );
}
```

This saves minor gas and keeps the logic equally readable.

Customer's response: Acknowledged. In practice, gas costs may actually increase for users with ≤ 3 positions due to certain optimizer internals.

I-03. Validate HF / LTV0 comment is vague

In the validateHFAndLtvzero() function of the ValidationLogic library, there is a comment that, while technically correct, is a bit vague.

Details:

```JavaScript
// User must either:
// 1. not have any ltv-zero collateral or
// 2. interact with asset that have 0 ltv on their position
```

Recommendation:
Make the comment more explicit and readable, for example:

```JavaScript
// User must withdraw any LTV0 collateral before withdrawing any other collateral.
```

Customer's response: Fixed in commit b2801b7.

Fix Review: Fix looks good.

I-04. Misleading error messages when categoryId == 0 in validateSetUserEMode()

Details:
The function reverts with eMode-specific error types even when categoryId == 0 (default mode).:

```
JavaScript

if (isBorrowed) {
  require(
    categoryId != 0
      ?
 EModeConfiguration.isReserveEnabledOnBitmap(eModeCategory.borrowableBitm
ap, i)
      : reservesData[reservesList[i]].configuration.getBorrowingEnabled(),
    Errors.InvalidDebtInEmode(reservesList[i])
  );
}

if (isEnabledAsCollateral) {
  require(
    getUserReserveLtv(reservesData[reservesList[i]], eModeCategory,
 categoryId) != 0,
    Errors.InvalidCollateralInEmode(reservesList[i])
  );
}
```

When categoryId == 0, these reverts still use InvalidDebtInEmode and InvalidCollateralInEmode, which is misleading since the user is not in eMode.

Impact:
Informational / UX; confusing revert reasons when interacting in default mode.

Recommendation:
Use context-appropriate error types based on whether the user is in eMode or not.
Because custom errors cannot be conditionally selected inline, handle each case explicitly:

```
JavaScript

if (isBorrowed) {
```

```
  bool borrowAllowed = categoryId != 0
    ?
 EModeConfiguration.isReserveEnabledOnBitmap(eModeCategory.borrowableBitm
ap, i)
    : reservesData[reservesList[i]].configuration.getBorrowingEnabled();

  if (categoryId != 0) {
    require(borrowAllowed, Errors.InvalidDebtInEmode(reservesList[i]));
  } else {
    require(borrowAllowed, Errors.InvalidDebt(reservesList[i]));
  }
}

if (isEnabledAsCollateral) {
  uint256 ltv = getUserReserveLtv(reservesData[reservesList[i]],
eModeCategory, categoryId);

  if (categoryId != 0) {
    require(ltv != 0, Errors.InvalidCollateralInEmode(reservesList[i]));
  } else {
    require(ltv != 0, Errors.InvalidCollateral(reservesList[i]));
  }
}
```

Customer's response: Fixed in commit [ce5d9ef](ce5d9ef).

Fix Review: Fix looks good.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.