Aave V3

Security Assessment

November 29, 2021

Prepared By:
Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Simone Monica | *Trail of Bits*
simone.monica@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Troy Sargent | *Trail of Bits*
troy.sargent@trailofbits.com

Changelog:
November 29, 2021: Initial report delivered
January 21, 2022: Fix Log (Appendix E) added

Executive Summary

From October 25 to November 24, 2021, Aave engaged Trail of Bits to review the security of the Aave V3 system. Trail of Bits conducted this assessment over 12 person-weeks, with 3 engineers working from commit `14f6148` of the Aave V3 repository.

During the first week of the assessment, we focused on gaining an understanding of the system and its architecture, and we started reviewing the borrow logic, supply logic, and bitmap libraries, as well as the stable and variable debt tokens. During the second week, we focused on the isolation mode feature, the supply, bridge, and generic logic libraries, and the `Pool` and `PoolConfigurator` contracts. During the third week, we focused on the liquidation, reserve, and eMode libraries, as well as the `DefaultReserveInterestRateStrategy` contract. During the fourth and fifth weeks, we focused on the flash loan and validation libraries, the upgradeability base contracts, and the various configuration contracts.

Our review resulted in 15 findings ranging from high to informational severity. One high-severity issue relates to a mistake in the `borrow` function that could cause a delegator to be liquidated immediately after a delegatee borrows assets. Three high-severity issues pertain to incorrect isolation mode checks; for instance, because the system does not correctly update users' debt after they liquidate loans, users could be blocked from using isolated assets in the protocol after a certain number of liquidations have occurred.

Appendix C contains recommendations on how to integrate ERC20 tokens. We recommend sharing these recommendations with Aave users. In addition to the security findings, we identified code-quality issues not related to any particular vulnerability, which are discussed in Appendix D.

Overall, the Aave V3 system adheres to smart contract best practices. However, the new features add another layer of complexity to the system and increase the likelihood of issues. At the time of the audit, the provided specification was a work in progress that improved with every week of the engagement; we recommend continuing to improve the specification, as it is still under development. The codebase's test suite could be expanded to include unit tests to check the enforcement and functionality of the isolation mode, eMode, and credit delegation features.

We recommend that Aave take the following actions before deploying the Aave V3 system:

- Address all issues detailed in this report.
- Implement property-based testing using Echidna to verify that important system invariants hold.
- Improve the unit test coverage of the isolation mode, eMode, and credit delegation

features.

- Investigate the arithmetic for calculating interest and front-running vulnerabilities.
- Finalize the specification/documentation of the V3 system.

Update: On January 21, 2022, Trail of Bits reviewed the fixes implemented by Aave for the issues described in this report. A detailed review of the current status of each issue is provided in Appendix E.

Project Dashboard

Application Summary

Name	Aave V3
Version	`14f6148`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	October 25–November 24, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	12 person-weeks

Vulnerability Summary

Total High-Severity Issues	5	■■■■■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	2	■■
Total Informational Issues	7	■■■■■■■
Total	15	

Category Breakdown

Authentication	1	■
Configuration	1	■
Data Validation	8	■■■■■■■■
Auditing and Logging	1	■
Undefined Behavior	4	■■■■
Total	15	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** The system's privileged operations and roles are limited. However, the system would benefit from explicit documentation highlighting the roles, powers, and limitations of every privileged actor.
Arithmetic	**Further investigation required.** Due to the time limit of the engagement, we did not cover the system's arithmetic in depth. However, we found precision errors in the `DefaultReserveInterestRateStrategy` contract (TOB-AAVE-012). The arithmetic would benefit from more thorough testing and fuzzing, particularly in the interest rate and reserve calculations.
Assembly Use	**Strong.** Assembly is limited to two operations required for the upgradeability pattern in use: a check for the code size at an address, using `extcodesize`, and the loading/storing of a value in a specific storage slot.
Code Stability	**Satisfactory.** No changes to the code were made during the audit. However, there is an open pull request to increase the max number of supported reserves, which, when implemented, will affect several parts of the system.
Decentralization	**Not considered.** The system's decentralization was not considered within the scope of this engagement.
Upgradeability	**Satisfactory.** The `PoolAddressesProvider` contract can be used to upgrade the system's contracts. The system uses the `delegatecall` pattern. The system would benefit from more unit tests for contracts specific to upgradeability. Integrating `slither-check-upgradeability` into the continuous integration pipeline would reduce upgradeability mistakes.
Function Composition	**Moderate.** The implementation extensively uses libraries and passes storage and memory structs into library functions, which makes tracking the execution flow difficult. Several complex functions could be split into simpler functions (e.g., `GenericLogic.calculateUserAccountData` and `BorrowLogic.executeBorrow`). The stable and variable debt token contracts contain duplicate code.

Front-Running	**Further investigation required**. Due to the time limit of the engagement, we did not cover the system's front running vulnerabilities in-depth.
Key Management	**Not considered.** The system's key management was not considered within the scope of this engagement.
Monitoring	**Moderate.** We identified one event that does not contain sufficient information (TOB-AAVE-004) and one place in which events contain incorrect data (TOB-AAVE-015). Also, at the time of the audit, there was no incident response plan and no documentation on how a monitoring system would use the events.
Specification	**Weak.** The specification did not exist at the start of the audit but was gradually developed during the audit. Only near the end of the audit did the specification include a list of system invariants. The specification is still a work in progress and should be improved for public consumption. Furthermore, we identified areas in which the implementation does not adhere to the specification (TOB-AAVE-007, TOB-AAVE-009, TOB-AAVE-010, TOB-AAVE-011).
Testing and Verification	**Moderate.** The unit test line coverage is high. However, we recommend improving the testing of edge cases in the new features. By more thoroughly testing edge cases, several of the issues we identified could have been prevented (TOB-AAVE-007, TOB-AAVE-009, TOB-AAVE-010, TOB-AAVE-011, TOB-AAVE-015). Finally, integrating a fuzzer to test important system invariants will help catch issues and reduce the likelihood of a system compromise.

Engagement Goals

The engagement was scoped to provide a security assessment of the Aave V3 system, with a specific focus on the new features.

Specifically, we sought to answer the following questions:

- Can a user evade liquidation?
- Can a user wrongfully trigger a liquidation?
- Can a user borrow more than his or her collateral is worth?
- Does the system ensure that users pay the correct interest and flash loan fees?
- Can a user circumvent isolation mode?
- Can lenders always withdraw their funds (if available)?
- Are there any bugs preventing borrowers from repaying loans?
- Can an attacker trap the system?
- Are access controls implemented correctly?
- Is the credit delegation feature implemented correctly?
- Are user configurations and balances updated correctly when aTokens are transferred?
- Do updates to reserve configurations put users at risk of liquidation?
- Does the system integrate with Chainlink oracles correctly?

Coverage

`Pool`. The `Pool` contract facilitates interactions between users and an Aave protocol market; for every action (e.g., supply, borrow), the function in the corresponding library's logic contract is called. We reviewed the `Pool` contract to ensure that adequate access controls are in place and that the library's functions are called with the correct parameters.

`PoolConfigurator`. The `PoolConfigurator` contract is the entry point for permissioned actors to apply changes to the corresponding pool; for some actions, the contract relies on the `ConfiguratorLogic` library. We ensured that adequate access controls are in place and that appropriate checks are applied to the setter function's parameters.

`protocol/configuration/`. This folder contains the `PoolAddressesProviderRegistry`, `PoolAddressesProvider`, and `ACLManager` contracts. The `PoolAddressesProviderRegistry` contract is the registry that holds `PoolAddressesProvider` addresses for different markets. The `PoolAddressesProvider` contract holds the addresses of the protocols for particular markets; the owner of `PoolAddressesProvider` can set these addresses. The `ACLManager` contract is the main registry that contains system roles and permissions. We ensured that adequate access controls are in place and that the system's roles are used appropriately.

DefaultReserveInterestRateStrategy. The `DefaultReserveInterestRateStrategy` contract implements the calculation of the interest rates. We compared the implementation to the specification and checked for rounding errors.

protocol/libraries/configuration/. This folder contains the `ReserveConfiguration` and `UserConfiguration` libraries, which implement the bitmap logic for handling the reserve and user configurations, respectively. We ensured that the correct bits are set and obtained in the appropriate functions.

protocol/libraries/logic/. This folder contains the libraries that implement the logic for the various functionalities. Through static analysis and a manual review, we looked for common Solidity flaws and ensured that every action adheres to the specification.

- **BorrowLogic.** The `BorrowLogic` library implements the logic for actions related to borrowing (e.g., borrowing, repaying, rebalancing the stable borrow rate, and swapping the borrow rate mode).

- **BridgeLogic.** The `BridgeLogic` library implements the logic for Portal, which allows supplied assets to flow between markets on different networks.

- **ConfiguratorLogic.** The `ConfiguratorLogic` library implements the logic for initializing reserves and replacing reserves' aTokens/debt tokens, which is used in the `PoolConfigurator` contract.

- **EModeLogic.** The `EModeLogic` library implements the logic for users to activate an eMode category.

- **FlashLoanLogic.** The `FlashLoanLogic` library implements the logic for flash loan functionalities.

- **GenericLogic.** The `GenericLogic` library implements the logic to calculate and validate the state of a user, particularly the user's total collateral, total debt, average loan-to-value ratio, average liquidation threshold, and health factor.

- **LiquidationLogic.** The `LiquidationLogic` library implements the logic for liquidating a position when the user's health factor is less than one.

- **ReserveLogic.** The `ReserveLogic` library implements the logic for updating a reserve's state.

- **`SupplyLogic`.** The `SupplyLogic` library implements the logic for supplying and withdrawing liquidity, using supply tokens as collateral, and transferring supply tokens.

- **`ValidationLogic`.** The `ValidationLogic` library implements the logic for checking whether actions related to supplying, borrowing, eMode, liquidations, and changes to a reserve meet the proper constraints.

`protocol/libraries/aave-upgradeability/`, `dependencies/openzeppelin/upgradeability/`. These folders consist of base contracts that allow contracts to be upgraded. We looked for common Solidity and upgradeability flaws.

`protocol/tokenization/`. This folder consists of multiple token contracts that track supplied and borrowed assets with the corresponding interest accrual. We reviewed the balance and interest bookkeeping.

`AaveOracle`. The `AaveOracle` contract uses Chainlink to get asset prices. We reviewed the use of the Chainlink interface.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-AAVE-001

❑ **Add the `chainID` opcode to the signature scheme to prevent post-deployment forks from affecting the `permit` call.** TOB-AAVE-002

❑ **Document the risk of `permit` front-running and ensure that external contracts and scripts reflect this possibility.** Alternatively, develop user documentation and on-chain mitigations to reduce the likelihood of a successful phishing campaign. TOB-AAVE-003

❑ **Add a parameter to the `Repay` event that specifies which token was used to repay.** This will help to differentiate which token was used to repay. TOB-AAVE-004

❑ **Move the `_mint` and `_burn` functions from `IncentivizedERC20` to `VariableDebtToken`.** This will remove the duplicate code in multiple contracts. TOB-AAVE-005

❑ **Ensure that the system sets the `_addressesProvider` variable using the `initialize` function instead of the `constructor`.** This will set the variable in the proxy contract's storage instead of the implementation contract's storage. TOB-AAVE-006

❑ **Use `_usersEModeCateogory[onBehalfOf]` instead of `_usersEModeCateogory[msg.sender]` in the `borrow` function.** This will prevent a delegator from incorrectly being liquidated. TOB-AAVE-007

❑ **Add `configureReserveAsCollateral`'s checks to `setEModeCategory`.** This will prevent incorrect LTV values from being set. TOB-AAVE-008

❑ **Implement the correct isolation mode checks (figure 9.1) in all of the indicated locations.** This will prevent isolation mode from being circumvented. TOB-AAVE-009

❑ **Add the isolation mode checks to the liquidation process before aTokens are assigned as collateral.** This will prevent isolation mode from being circumvented. TOB-AAVE-010

❑ **Ensure that `isolationModeTotalDebt` decreases when a loan is liquidated.** This will keep the isolation mode debt accounting correct. TOB-AAVE-011

❑ **Clarify whether this behavior is intentional and update the relevant documentation.** If this behavior is not correct, use a consistent order of operations to maintain precision, and use the correct comparison operator. TOB-AAVE-012

❑ **Consider using the recommended `AggregatorInterfaceV3` interface and `latestRoundData` function.** TOB-AAVE-013

❑ **Check for contract existence before a `delegatecall`. Document that `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.** TOB-AAVE-014

❑ **Ensure that the system tracks the variable index in the `_userConfig` of `onBehalfOf`.** This will ensure that the internal accounting is correct and that the emitted events contain correct data. TOB-AAVE-015

Long term

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-AAVE-001

❑ **Identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.** TOB-AAVE-002

❑ **Document best practices for users of the Aave contracts.** In addition to taking other precautions, users must do the following: be extremely careful when signing a message; avoid signing messages from suspicious sources; and always require hashing schemes to be public. TOB-AAVE-003

❑ **Identify the data that could be useful to an off-chain system to monitor the activity in the contracts and add this data in the appropriate events.** TOB-AAVE-004

❑ **When a base contract is inherited by multiple classes and the base class contains a function that is called only from one derived class, consider moving that code to that derived class.** This removes dead/unused code and improves the readability of the codebase. TOB-AAVE-005

❑ **When developing upgradeable contracts, minimize the use of constructor functions unless they are absolutely necessary.** TOB-AAVE-006

❑ **Write extensive unit tests that test all of the expected post conditions.** TOB-AAVE-007

❑ **Clearly document the valid ranges of parameters and ensure that the system always checks them.** TOB-AAVE-008

❑ **If a multi-line check is required in several places, consider moving the check into a function and calling that function in all locations that require the check.** This will lower the risk of missing or incomplete checks across the codebase. TOB-AAVE-009, TOB-AAVE-010, TOB-AAVE-011

❑ **Ensure that the implementation of formulas is correct and consistent throughout the codebase.** TOB-AAVE-012

❑ **When interfacing with external contracts, stay up to date with their development.** When a recommendation is made to change the integration, weigh the pros and cons of adhering to or ignoring the recommendation to make a well-informed decision. TOB-AAVE-013

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using the `delegatecall` proxy pattern.** TOB-AAVE-014

❑ **Ensure that invariants related to borrows, repays, liquidations, eMode, and flash loans are not broken when credit delegation is used.** TOB-AAVE-015

Findings Summary

#	Title	Type	Severity
1	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
2	Lack of chainID validation allows attackers to reuse signatures across forks	Authentication	High
3	Risks associated with EIP-2612	Configuration	Informational
4	Insufficient Repay event parameters	Auditing and Logging	Informational
5	Base class functions that are used only in a single derived class could cause confusion	Undefined Behavior	Informational
6	Use of the constructor rather than the initialize function prevents the incentives controller from being updated after deployment	Undefined Behavior	Low
7	Incorrect eMode category fetched by borrow	Data Validation	High
8	Missing validation when setting eMode categories	Data Validation	Low
9	Missing/incorrect isolation mode checks circumvent collateral isolation mode	Data Validation	High
10	Isolation mode bypassed when liquidating and receiving aTokens	Data Validation	High
11	Isolation mode total debt does not decrease on liquidation, potentially blocking new loans using the isolated asset	Data Validation	High
12	Unclear behavior when calculating interest rates	Undefined Behavior	Informational
13	Use of deprecated Chainlink interface and function	Data Validation	Informational

14	Lack of contract existence check on delegatecall	Data Validation	Informational
15	Variable debt token incorrectly tracks debtor's previous index	Data Validation	Medium

1. Solidity compiler optimizations can be problematic

Severity: Informational
Type: Undefined Behavior
Target: `hardhat.config.ts`

Difficulty: Low
Finding ID: TOB-AAVE-001

Description
The Aave smart contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Aave smart contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

2. Lack of chainID validation allows attackers to reuse signatures across forks

Severity: High
Difficulty: High
Type: Authentication
Finding ID: TOB-AAVE-002
Target: `protocol/tokenization/AToken.sol`, `protocol/tokenization/VariableDebtToken.sol`, `protocol/tokenization/StableDebtToken.sol`, `protocol/tokenization/base/DebtTokenBase.sol`

Description

The `AToken`, `VariableDebtToken`, and `StableDebtToken` contracts implement EIP-2612 to provide EIP-712-signed approvals through the `permit` function. The domain separator and the `chainID` are included in the signature scheme. However, the `chainID` is fixed at the time of deployment. In the event of a post-deployment hard fork of the chain, the `chainID` cannot be updated, and signatures may be replayed across both versions of the chain. As a result, an attacker could reuse signatures to receive user funds on both chains. Explicitly including the `chainID` in the scheme of the signature passed to `permit` would mitigate this issue without requiring the regeneration of the entire domain separator.

The `chainID` is included in the `DOMAIN_SEPARATOR`, which is part of the signed data of a `permit` call. The `DOMAIN_SEPARATOR` is created once, during deployment.

```
function initialize(
  address treasury,
  address underlyingAsset,
  IAaveIncentivesController incentivesController,
  uint8 aTokenDecimals,
  string calldata aTokenName,
  string calldata aTokenSymbol,
  bytes calldata params
) external override initializer {
  uint256 chainId = block.chainid;

  DOMAIN_SEPARATOR = keccak256(
    abi.encode(
      EIP712_DOMAIN,
      keccak256(bytes(aTokenName)),
      keccak256(EIP712_REVISION),
      chainId,
      address(this)
    )
  );
```

Figure 2.1: `protocol/tokenization/AToken.sol#L60-L79`

```
function initialize(
  address underlyingAsset,
```

```
  IAaveIncentivesController incentivesController,
  uint8 debtTokenDecimals,
  string memory debtTokenName,
  string memory debtTokenSymbol,
  bytes calldata params
) external override initializer {
  uint256 chainId = block.chainid;

  [..]

  DOMAIN_SEPARATOR = keccak256(
    abi.encode(
      EIP712_DOMAIN,
      keccak256(bytes(debtTokenName)),
      keccak256(EIP712_REVISION),
      chainId,
      address(this)
    )
  );
```

Figure 2.2: `protocol/tokenization/VariableDebtToken.sol#L32-L57`

The signature scheme does not account for a change in the contract's chain after the initial deployment. As a result, if a fork of Ethereum is made after the contract's creation, every signature will be usable in both forks.

Depending on the `deadline` value in the signed data, this issue is unlikely to be exploited. The shorter the `deadline`, the more unlikely cross-chain replay will be.

Exploit Scenario
Alice permits Bob to approve 1,000 of her aTokens on the Ethereum mainnet. The Ethereum mainnet is forked into Ethereum 2.0, changing the `chainID`. Because of the hard-coded `chainID`, Bob can call `permit` with the same signature on both chains.

Recommendations
Short term, add the `chainID` opcode to the signature scheme to prevent post-deployment forks from affecting the `permit` call.

Long term, identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.

3. Risks associated with EIP-2612

Severity: Informational
Type: Configuration
Target: `protocol/tokenization/AToken.sol`, `protocol/tokenization/VariableDebtToken.sol`, `protocol/tokenization/StableDebtToken.sol`, `protocol/tokenization/base/DebtTokenBase.sol`

Difficulty: High
Finding ID: TOB-AAVE-003

Description
The use of EIP-2612 increases the risk of `permit` function front-running as well as phishing attacks.

EIP-2612 uses signatures as an alternative to the traditional `approve` and `transferFrom` flow. These signatures allow a third party to transfer tokens on behalf of a user, with verification of a signed message.

An external party can front-run the `permit` function by submitting the signature first. The use of EIP-2612 makes it possible for a different party to front-run the initial caller's transaction. As a result, the intended caller's transaction will fail (as the signature has already been used and the funds have been transferred). This may also affect external contracts that rely on a successful `permit()` call for execution. For more information, see the EIP-2612 documentation.

EIP-2612 also makes it easier for an attacker to steal a user's tokens through phishing by asking for signatures in a context unrelated to the Aave contracts. The hash message may look benign and random to users.

Exploit Scenario
Bob has 1,000 aTokens. Eve creates a system to interact with Aave contracts by executing operations on a user's behalf. Her system should ask users to sign a hash to approve the `AToken` contract for the funds they want to interact with. She maliciously generates a hash to transfer 1,000 aTokens from Bob to herself instead of to the `AToken` contract. Eve asks Bob to sign the hash to approve the funds. Bob signs the hash, and Eve uses it to steal Bob's tokens.

Recommendations
Short term, document the risk of `permit` front-running and ensure that external contracts and scripts reflect this possibility. Alternatively, develop user documentation and on-chain mitigations to reduce the likelihood of a successful phishing campaign.

Long term, document best practices for users of the Aave contracts. In addition to taking other precautions, users must do the following:

- Be extremely careful when signing a message
- Avoid signing messages from suspicious sources
- Always require hashing schemes to be public

4. Insufficient Repay event parameters

Severity: Informational
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-AAVE-004
Target: `protocol/libraries/logic/BorrowLogic.sol`, `interfaces/IPool.sol`

Description
Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. The Aave V3 system allows users to repay loans with the borrowed asset or with the corresponding aToken; however, the `Repay` event does not specify which type of token was used to repay the loan.

```
event Repay(
    address indexed reserve,
    address indexed user,
    address indexed repayer,
    uint256 amount
);
```

Figure 4.1: `protocol/libraries/logic/BorrowLogic.sol#L39-L44`

Exploit Scenario
Alice operates an off-chain monitoring system that relies on Aave events. Because the `Repay` event does not specify which token was used to repay, Alice's service is unable to accurately track the activity in the contract.

Recommendations
Short term, add a parameter to the `Repay` event that specifies which token was used to repay.

Long term, identify the data that could be useful to an off-chain system to monitor the activity in the contracts and add this data in the appropriate events.

5. Base class functions that are used only in a single derived class could cause confusion

Severity: Informational
Difficulty: High
Type: Undefined Behavior
Finding ID: TOB-AAVE-005
Target: `protocol/tokenization/IncentivizedERC20.sol`

Description

The `_burn` and `_mint` functions defined in `IncentivizedERC20` are called only in `VariableDebtToken`. The other derived class, `StableDebtToken`, defines its own `_burn` and `_mint` functions and does not call the functions in `IncentivizedERC20`. This could cause confusion and could increase the possibility that developers upgrading the contract update the wrong `_burn` and/or `_mint` function(s).

Removing the `_mint` and `_burn` functions from `IncentivizedERC20` and defining them within `VariableDebtToken` would remove this confusion and the unused functions in `StableDebtToken`.

Exploit Scenario

Alice, a developer, wants to implement an update to the `_mint` function of both debt tokens. She sees that they both inherit from `IncentivizedERC20` and updates the `_mint` function defined in that contract. She later discovers that the function is called only from `VariableDebtToken`, not `StableDebtToken`.

Recommendations

Short term, move the `_mint` and `_burn` functions from `IncentivizedERC20` to `VariableDebtToken`.

Long term, when a base contract is inherited by multiple classes and the base class contains a function that is called only from one derived class, consider moving that code to that derived class. This removes dead/unused code and improves the readability of the codebase.

6. Use of the constructor rather than the initialize function prevents the incentives controller from being updated after deployment

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-AAVE-006
Target: `protocol/tokenization/IncentivizedERC20.sol`

Description

The use of the `constructor` rather than the `initialize` function in the debt token contracts prevents `_incentivesController` from being updated after deployment (without upgrading the contract).

The `StableDebtToken` and `VariableDebtToken` contracts are upgradeable and use the `initialize` function. Nonetheless, these two contracts also have a `constructor`. This `constructor` calls `IncentivizedERC20`'s `constructor`, which sets the `_addressesProvider` variable. However, the `constructor` sets the variable in the implementation contract instead of the proxy contract's storage (i.e., the variable is the zero address in the proxy contract).

```
constructor(IPool pool) DebtTokenBase(pool) {}
```

Figure 6.1: `protocol/tokenization/StableDebtToken.sol#L34`

```
constructor(IPool pool)
   IncentivizedERC20(pool.getAddressesProvider(), 'DEBT_TOKEN_IMPL', 'DEBT_TOKEN_IMPL', 0)
```

Figure 6.2: `protocol/tokenization/base/DebtTokenBase.sol#L41-L42`

```
constructor(
  IPoolAddressesProvider addressesProvider,
  string memory name,
  string memory symbol,
  uint8 decimals
) {
  _addressesProvider = addressesProvider;
```

Figure 6.3: `protocol/tokenization/IncentivizedERC20.sol#L48-L54`

The `_addressesProvider` variable is used in the `onlyPoolAdmin` modifier, which is applied to the `setIncentivesController` function. Since the `_addressesProvider` variable is the zero address in the proxy contract, the `getACLManager` function will be called on a contract at the zero address, causing the `onlyPoolAdmin` modifier to revert. This prevents the `setIncentivesController` function from being called.

```
modifier onlyPoolAdmin() {
  IACLManager aclManager = IACLManager(_addressesProvider.getACLManager());
  require(aclManager.isPoolAdmin(msg.sender), Errors.CALLER_NOT_POOL_ADMIN);
  _;
}
```

Figure 6.4: `protocol/tokenization/IncentivizedERC20.sol#L22-L26`

```
function setIncentivesController(IAaveIncentivesController controller) external
onlyPoolAdmin {
  _incentivesController = controller;
}
```

Figure 6.5: `protocol/tokenization/IncentivizedERC20.sol#L22-L26`

Exploit Scenario
The Aave team tries to update the incentives controller of an existing stable debt token by calling the `setIncentivesController` function from the configured pool admin account. However, the transaction reverts.

Recommendations
Short term, ensure that the system sets the `_addressesProvider` variable using the `initialize` function instead of the `constructor`. This will set the variable in the proxy contract's storage instead of the implementation contract's storage.

Long term, when developing upgradeable contracts, minimize the use of constructor functions unless they are absolutely necessary.

7. Incorrect eMode category fetched by borrow

Severity: High
Type: Data Validation
Target: protocol/pool/Pool.sol

Difficulty: Low
Finding ID: TOB-AAVE-007

Description

The borrow function incorrectly fetches the eMode category of msg.sender instead of onBehalfOf. This error could cause a delegator to be liquidated immediately after the delegatee borrows assets; it could also allow the delegatee to borrow assets outside of the eMode selected by the delegator.

```
function borrow(
  address asset,
  uint256 amount,
  uint256 interestRateMode,
  uint16 referralCode,
  address onBehalfOf
) external override {
  BorrowLogic.executeBorrow(
    _reserves,
    _reservesList,
    _eModeCategories,
    _usersConfig[onBehalfOf],
    DataTypes.ExecuteBorrowParams(
      [...]
      _usersEModeCategory[msg.sender],
      _addressesProvider.getPriceOracleSentinel()
    )
  );
}
```

Figure 7.1: protocol/pool/Pool.sol#L190-L217

Exploit Scenario

Assume that DAI's loan-to-value ratio is 80% and the liquidation threshold is 85%, and assume that the stablecoin eMode's loan-to-value ratio is 97% and the liquidation threshold is 98%. Alice, without eMode enabled, delegates her full amount, 100 DAI, to Eve. Eve, with eMode enabled, borrows 97 USDC using delegated credit. Alice can now be liquidated.

Recommendations

Short term, use _usersEModeCateogory[onBehalfOf] instead of _usersEModeCateogory[msg.sender] in the borrow function.

Long term, write extensive unit tests that test all of the expected post conditions. Unit tests could have uncovered this issue.

8. Missing validation when setting eMode categories

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-AAVE-008
Target: `protocol/pool/PoolConfigurator.sol`

Description

The `setEModeCategory` function allows the risk or pool admin to set the loan-to-value ratio, liquidation threshold, and liquidation bonus for the different eMode categories, but the function does not validate the values. For example, the loan-to-value ratio must be less than the liquidation threshold, but the function does not check for this condition.

```
function setEModeCategory(
  uint8 categoryId,
  uint16 ltv,
  uint16 liquidationThreshold,
  uint16 liquidationBonus,
  address oracle,
  string calldata label
) external override onlyRiskOrPoolAdmins {
  _pool.configureEModeCategory(
    categoryId,
    DataTypes.EModeCategory(ltv, liquidationThreshold, liquidationBonus, oracle, label)
  );
  emit EModeCategoryAdded(categoryId, ltv, liquidationThreshold, liquidationBonus, oracle,
label);
}
```

Figure 8.1: `protocol/pool/PoolConfigurator.sol#L300-L313`

The checks that `setEModeCategory` should perform are implemented in `configureReserveAsCollateral`.

```
function configureReserveAsCollateral(
  address asset,
  uint256 ltv,
  uint256 liquidationThreshold,
  uint256 liquidationBonus
) external override onlyRiskOrPoolAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(asset);

  //validation of the parameters: the LTV can
  //only be lower or equal than the liquidation threshold
  //(otherwise a loan against the asset would cause instantaneous liquidation)
  require(ltv <= liquidationThreshold, Errors.PC_INVALID_CONFIGURATION);

  [...]
}
```

Figure 8.2: `protocol/pool/PoolConfigurator.sol#L140-L179`

Exploit Scenario

Bob, a pool admin, creates a new eMode category X and incorrectly sets the liquidation threshold to greater than the loan-to-value ratio. Consequently, every loan of category X tokens with eMode active will be immediately liquidable.

Recommendations
Short term, add `configureReserveAsCollateral`'s checks to `setEModeCategory`.

Long term, clearly document the valid ranges of parameters and ensure that the system always checks them.

9. Missing/incorrect isolation mode checks circumvent collateral isolation mode

Severity: High	Difficulty: Low
Type: Data Validation	Finding ID: TOB-AAVE-009

Target: `protocol/pool/SupplyLogic.sol`, `protocol/pool/BridgeLogic.sol`

Description

There are several places in the codebase in which the isolation mode checks for newly added collateral are either missing or incorrect. Only one isolated asset (or multiple non-isolated assets) should be marked as collateral. However, due to the missing or incorrect isolation mode checks, multiple isolated assets could be marked as collateral.

Figure 9.1 shows the correct isolation mode checks. An added asset should be marked as collateral only if the user has not yet added any collateral, or if the user has already added collateral and both the existing collateral and the added asset are non-isolated.

```
if (isFirstSupply) {
  (bool isolationModeActive, , ) = userConfig.getIsolationModeState(reserves, reservesList);
  if (
    ((!isolationModeActive && (reserveCache.reserveConfiguration.getDebtCeiling() == 0)) ||
      !userConfig.isUsingAsCollateralAny())
  ) {
    userConfig.setUsingAsCollateral(reserve.id, true);
    emit ReserveUsedAsCollateralEnabled(params.asset, params.onBehalfOf);
  }
}
```

Figure 9.1: `protocol/libraries/logic/SupplyLogic.sol#L69-L78`

Figure 9.2 shows that `mintUnbacked` does not contain the isolation mode checks. As a result, the function allows other assets to be marked as collateral even if an isolated asset has already been marked as collateral. The function also allows multiple isolated assets to be added as collateral.

```
if (isFirstSupply) {
  userConfig.setUsingAsCollateral(reserve.id, true);
  emit ReserveUsedAsCollateralEnabled(asset, onBehalfOf);
}
```

Figure 9.2: `protocol/libraries/logic/BridgeLogic.sol#L79-L82`

Figure 9.3 shows that `finalizeTransfer` contains incomplete isolation mode checks. If isolation mode is inactive, indicating that there may be multiple non-isolated assets used as collateral, the function still allows an isolated asset to be marked as collateral.

```
(bool isolationModeActive, , ) = toConfig.getIsolationModeState(reserves, reservesList);
if (!isolationModeActive) {
  toConfig.setUsingAsCollateral(reserveId, true);
```

```
  emit ReserveUsedAsCollateralEnabled(params.asset, params.to);
}
```

Figure 9.3: `protocol/libraries/logic/SupplyLogic.sol#L178-L182`

Exploit Scenario

Bob deposits an isolated asset X and marks it as collateral. The `mintUnbacked` function is called to deposit an isolated asset Y on behalf of Bob, marking asset Y as Bob's collateral. Bob now has two isolated assets as collateral.

Recommendations

Short term, implement the correct isolation mode checks (figure 9.1) in all of the indicated locations.

Long term, if a multi-line check is required in several places, consider moving the check into a function and calling that function in all locations that require the check. This will lower the risk of missing or incomplete checks across the codebase.

10. Isolation mode bypassed when liquidating and receiving aTokens

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-AAVE-010
Target: `protocol/libraries/logic/LiquidationLogic.sol`

Description
When a user liquidates and chooses to receive aTokens and his current aToken balance is zero, the isolation mode checks are not performed and the received aTokens are simply set as the liquidator's collateral. If the liquidator's account is in isolation mode, both an isolated asset and the received aToken will be set as his collateral. This breaks the following isolation mode invariant, as specified in Aave's documentation: "Borrowers supplying an isolated asset as collateral cannot supply other assets as collateral (though they can still supply to capture yield)."

```
if (params.receiveAToken) {
  vars.liquidatorPreviousATokenBalance =
IERC20(vars.collateralAtoken).balanceOf(msg.sender);
  vars.collateralAtoken.transferOnLiquidation(
    params.user,
    msg.sender,
    vars.maxCollateralToLiquidate
  );

  if (vars.liquidatorPreviousATokenBalance == 0) {
    DataTypes.UserConfigurationMap storage liquidatorConfig = usersConfig[msg.sender];
    liquidatorConfig.setUsingAsCollateral(collateralReserve.id, true);
```

Figure 10.1: `protocol/libraries/logic/LiquidationLogic.sol#L201-L211`

Exploit Scenario
Bob has an isolated asset X set as collateral. Bob liquidates Alice and chooses to receive aTokens. Bob now has both the isolated asset X and the received aTokens set as collateral.

Recommendations
Short term, add the isolation mode checks to the liquidation process before aTokens are assigned as collateral.

Long term, if a multi-line check is required in several places, consider moving the check into a function and calling that function in all locations that require the check. This will lower the risk of missing or incomplete checks across the codebase.

11. Isolation mode total debt does not decrease on liquidation, potentially blocking new loans using the isolated asset

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-AAVE-011
Target: `protocol/libraries/logic/LiquidationLogic.sol`

Description

When a user liquidates debt with an isolated asset as collateral, `isolationModeTotalDebt` does not decrease. As a result, the difference between the actual isolated debt and the debt according to `isolationModeTotalDebt` will increase with every liquidation. There is a cap on the amount that can be borrowed using an isolated asset as collateral; with each liquidation that occurs, `isolationModeTotalDebt` will grow closer to this cap. Eventually, the cap could be reached, blocking the user from taking out loans using that isolated asset as collateral.

Figure 11.1 shows that `isolationModeTotalDebt` increases when a user takes out a loan using an isolated asset as collateral. Figure 11.2 shows that `isolationModeTotalDebt` should decrease when a user repays a loan using an isolated asset as collateral. However, the implementation's liquidation logic does not decrease the `isolationModeTotalDebt` variable.

```
if (isolationModeActive) {
  reserves[isolationModeCollateralAddress].isolationModeTotalDebt += Helpers.castUint128(
    params.amount /
      10 **
        (reserveCache.reserveConfiguration.getDecimals() -
          ReserveConfiguration.DEBT_CEILING_DECIMALS)
  );
}
```

Figure 11.1: `protocol/libraries/logic/BorrowLogic.sol#L115-L122`

```
if (isolationModeActive) {
  uint128 isolationModeTotalDebt = reserves[isolationModeCollateralAddress]
    .isolationModeTotalDebt;

  uint128 isolatedDebtRepaid = Helpers.castUint128(
    paybackAmount /
      10 **
        (reserveCache.reserveConfiguration.getDecimals() -
          ReserveConfiguration.DEBT_CEILING_DECIMALS)
  );

  // since the debt ceiling does not take into account the interest accrued, it might happen
that amount repaid > debt in isolation mode
  if (isolationModeTotalDebt <= isolatedDebtRepaid) {
    reserves[isolationModeCollateralAddress].isolationModeTotalDebt = 0;
  } else {
    reserves[isolationModeCollateralAddress].isolationModeTotalDebt =
      isolationModeTotalDebt -
      isolatedDebtRepaid;
```

```
  }
}
```

Figure 11.2: `protocol/libraries/logic/BorrowLogic.sol#L200-L219`

Exploit Scenario
After 1,000 liquidations of loans with isolated asset X as collateral, `isolationModeTotalDebt` has grown so much that the max cap has been reached, blocking the user from taking out loans using isolated asset X as collateral.

Recommendations
Short term, ensure that `isolationModeTotalDebt` decreases when a loan is liquidated.

Long term, if a multi-line check is required in several places, consider moving the check into a function and calling that function in all locations that require the check. This will lower the risk of missing or incomplete checks across the codebase.

12. Unclear behavior when calculating interest rates

Severity: Informational	Difficulty: Low
Type: Undefined Behavior	Finding ID: TOB-AAVE-012
Target: `protocol/pool/DefaultReserveInterestRateStrategy.sol`	

Description

The functions used to calculate the stable and variable interest rates use inconsistent orders of operations and do not align with the specification in the Aave V2 white paper.

The following is the formula for the borrow rate:

$\#R_t^{asset}$, **borrow rate**

$$\#R_t^{asset} = \begin{cases} \#R_{base}^{asset} + \frac{U_t^{asset}}{U_{optimal}^{asset}} \#R_{slope1}^{asset}, \text{ if } U_t^{asset} < U_{optimal}^{asset} \\ \#R_{base}^{asset} + \#R_{slope1}^{asset} + \frac{U_t^{asset} - U_{optimal}}{1 - U_{optimal}} \#R_{slope2}^{asset}, \text{ if } U_t^{asset} \geq U_{optimal} \end{cases}$$

The current implementation checks that `vars.borrowUtilizationRate > OPTIMAL_UTILIZATION_RATE`, but the Aave V2 white paper specifies that it should be `vars.borrowUtilizationRate >= OPTIMAL_UTILIZATION_RATE`.

The `currentStableBorrowRate` and `currentVariableBorrowRate` functions use different orders of operations even though they use the same formula. This results in differing levels of precision for the same input. In particular, `currentStableBorrowRate` divides before multiplying, so the result is less precise than that of `currentVariableBorrowRate`.

```
function calculateInterestRates(DataTypes.CalculateInterestRatesParams memory params)
  external
  view
  override
  returns (
    uint256,
    uint256,
    uint256
  )
{
  [...]
if (vars.borrowUtilizationRate > OPTIMAL_UTILIZATION_RATE) {
  [...]
} else {
  vars.currentStableBorrowRate =
    vars.currentStableBorrowRate +
    _stableRateSlope1.rayMul(vars.borrowUtilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE));

  vars.currentVariableBorrowRate =
    _baseVariableBorrowRate +
    _variableRateSlope1.rayMul(vars.borrowUtilizationRate).rayDiv(OPTIMAL_UTILIZATION_RATE);
}
```

Figure 12.1: `protocol/pool/DefaultReserveInterestRateStrategy.sol#132-190`

Recommendations
Short term, clarify whether this behavior is intentional and update the relevant documentation. If this behavior is not correct, use a consistent order of operations to maintain precision, and use the correct comparison operator.

Long term, ensure that the implementation of formulas is correct and consistent throughout the codebase.

13. Use of deprecated Chainlink interface and function

Severity: Informational
Type: Data Validation
Target: `misc/AaveOracle.sol`

Difficulty: High
Finding ID: TOB-AAVE-013

Description

The `AaveOracle` contract uses the deprecated `IChainlinkAggregator` interface instead of the latest `AggregatorInterfaceV3` interface. Furthermore, the contract calls the deprecated `latestAnswer` function to retrieve the current price instead of the `latestRoundData` function recommended by Chainlink.

The `latestAnswer` function might return zero if no answer is reached, which should be handled appropriately by the caller. The `AaveOracle` contract correctly checks and handles this case.

```
int256 price = IChainlinkAggregator(source).latestAnswer();
if (price > 0) {
  return uint256(price);
} else {
  return _fallbackOracle.getAssetPrice(asset);
}
```

Figure 13.1: `misc/AaveOracle.sol#L128-L133`

The recommended `latestRoundData` function returns additional data (such as a timestamp and round ID), which could be used to check whether the answer was reported recently enough.

Recommendations

Short term, consider using the recommended `AggregatorInterfaceV3` interface and `latestRoundData` function.

Long term, when interfacing with external contracts, stay up to date with their development. When a recommendation is made to change the integration, weigh the pros and cons of adhering to or ignoring the recommendation to make a well-informed decision.

14. Lack of contract existence check on delegatecall

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-AAVE-014
Target: `dependencies/openzeppelin/upgradeability/Proxy.sol`

Description

The project uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the proxy can cause unexpected behavior. On the other hand, `upgradeTo(newImplementation)`, which updates the logic contract, checks that `newImplementation` is a contract.

```
function _delegate(address implementation) internal {
  //solium-disable-next-line
  assembly {
    // Copy msg.data. We take full control of memory in this inline assembly
    // block because it will not return to Solidity code. We overwrite the
    // Solidity scratch pad at memory position 0.
    calldatacopy(0, 0, calldatasize())

    // Call the implementation.
    // out and outsize are 0 because we don't know the size yet.
    let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

    // Copy the returned data.
    returndatacopy(0, 0, returndatasize())

    switch result
    // delegatecall returns 0 on error.
    case 0 {
      revert(0, returndatasize())
    }
    default {
      return(0, returndatasize())
    }
  }
}
```

Figure 14.1: `dependencies/openzeppelin/upgradeability/Proxy.sol#L32-L56`

A `delegatecall` to a destructed contract will return success as part of the EVM specification. The [Solidity documentation] includes the following warning:

> *The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.*

Figure 14.2: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

If `selfdestruct` were executed within the logic contract, future calls to the proxy would always succeed; this could result in unexpected behavior. However, there is no `selfdestruct` or `delegatecall` functionality in any of the logic contracts. For this reason, the severity of this issue is informational.

Exploit Scenario
Alice, a developer, introduces a bug through which the logic contract of upgradeable contract A can be destroyed. All calls to the corresponding proxy contract of contract A now succeed instead of reverting.

Recommendations
Short term, check for contract existence before a `delegatecall`. Document that `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, and the pitfalls of using the `delegatecall` proxy pattern.

References
- Contract upgrade anti-patterns

15. Variable debt token incorrectly tracks debtor's previous index

Severity: Medium
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-AAVE-015
Target: `protocol/libraries/logic/BorrowLogic.sol`, `protocol/libraries/logic/ValidationLogic.sol`, `protocol/tokenization/VariableDebtToken.sol`

Description
When a user borrows funds at a variable rate from the Aave protocol, the `mint` function stores `reserveCache.nextVariableBorrowIndex` in `_userConfig[user].additionalData` and then mints debt tokens to `onBehalfOf`. This causes unexpected behavior when credit delegation is used, because `_userConfig[onBehalfOf].additionalData` is uninitialized (i.e., zero), and the credit delegator's loan information cannot be accessed.

```
function mint(
  address user,
  address onBehalfOf,
  uint256 amount,
  uint256 index
) external override onlyPool returns (bool, uint256) {
  if (user != onBehalfOf) {
    _decreaseBorrowAllowance(onBehalfOf, user, amount);
  }

  uint256 amountScaled = amount.rayDiv(index);
  require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);

  uint256 scaledBalance = super.balanceOf(user);
  uint256 accumulatedInterest = scaledBalance.rayMul(index) -
    scaledBalance.rayMul(_userState[user].additionalData);

  _userState[user].additionalData = Helpers.castUint128(index);

  _mint(onBehalfOf, Helpers.castUint128(amountScaled));

  emit Transfer(address(0), onBehalfOf, amount + accumulatedInterest);
  emit Mint(user, onBehalfOf, amount + accumulatedInterest, index);

  return (scaledBalance == 0, scaledTotalSupply());
}
```

Figure 15.1: `contracts/protocol/tokenization/VariableDebtToken.sol#L87-L112`

When minting stable debt tokens, the system stores the loan information of the credit delegator in `_userConfig[onBehalfOf].additionalData`.

```
_userState[onBehalfOf].additionalData = Helpers.castUint128(vars.nextStableRate);
```

Figure 15.2: `contracts/protocol/tokenization/StableDebtToken.sol#L152`

The variable index accounts for the interest earned on variable debt. To calculate the compound interest rate, the system uses `block.timestamp`, which retains the same value in a single block; if the `reserve` is updated (i.e., a user repays a loan) in the same block in which the user took out the loan, zero interest accumulates. This does not present an issue when `user` and `onBehalfOf` are the same address, but it does cause issues when Aave's credit delegation functionality is used.

The incorrect tracking of the variable index breaks the invariant that users cannot borrow and repay within the same block; by borrowing and repaying in a single block, users can avoid paying interest and flash loan fees. Because `variableDebtPreviousIndex` is uninitialized for the credit delegator, his `variableDebtPreviousIndex` is zero. As a result, `variableDebtPreviousIndex < reserveCache.nextVariableBorrowIndex` will not revert when calculated in the same block, as expected.

```
function validateRepay(
  DataTypes.ReserveCache memory reserveCache,
  uint256 amountSent,
  DataTypes.InterestRateMode rateMode,
  address onBehalfOf,
  uint256 stableDebt,
  uint256 variableDebt
) internal view {
  (bool isActive, , , , bool isPaused) = reserveCache.reserveConfiguration.getFlags();
  require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
  require(!isPaused, Errors.VL_RESERVE_PAUSED);

  require(amountSent > 0, Errors.VL_INVALID_AMOUNT);

  uint256 variableDebtPreviousIndex =
IScaledBalanceToken(reserveCache.variableDebtTokenAddress)
    .getPreviousIndex(onBehalfOf);

  uint40 stableRatePreviousTimestamp = IStableDebtToken(reserveCache.stableDebtTokenAddress)
    .getUserLastUpdated(onBehalfOf);

  require(
    (stableRatePreviousTimestamp < uint40(block.timestamp) &&
      DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.STABLE) ||
      (variableDebtPreviousIndex < reserveCache.nextVariableBorrowIndex &&
        DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.VARIABLE),
    Errors.VL_SAME_BLOCK_BORROW_REPAY
  );
```

Figure 15.3: `contracts/protocol/libraries/logic/ValidationLogic.sol#L302-L328`

Additionally, the calculation for `accumulatedInterest` in `mint` and `burn` operations for variable debt tokens is incorrect since the uninitialized value (i.e., zero) is read from `_userConfig[onBehalfOf].additionalData`, and incorrect values will be emitted in the events.

```
function burn(
  address user,
  uint256 amount,
  uint256 index
) external override onlyPool returns (uint256) {
  uint256 amountScaled = amount.rayDiv(index);
  require(amountScaled != 0, Errors.CT_INVALID_BURN_AMOUNT);
```

```
  uint256 scaledBalance = super.balanceOf(user);
  uint256 accumulatedInterest = scaledBalance.rayMul(index) -
    scaledBalance.rayMul(_userState[user].additionalData);

  _userState[user].additionalData = Helpers.castUint128(index);

  _burn(user, Helpers.castUint128(amountScaled));

  if (accumulatedInterest > amount) {
    emit Transfer(address(0), user, accumulatedInterest - amount);
    emit Mint(user, user, accumulatedInterest - amount, index);
  } else {
    emit Transfer(user, address(0), amount - accumulatedInterest);
    emit Burn(user, amount - accumulatedInterest, index);
  }
  return scaledTotalSupply();
}
```

Figure 15.4: `contracts/protocol/libraries/logic/BorrowLogic.sol#L49-L103`

Exploit Scenario

Alice delegates collateral to a smart contract. When the contract borrows tokens at a variable rate, the index is stored under the smart contract's address instead of Alice's. Alice calls the smart contract, which borrows DAI at a variable rate to perform arbitrage activities. In the same transaction, Alice repays her debt without paying interest or flash loan fees.

Recommendations

Short term, ensure that the system tracks the variable index in the `_userConfig` of `onBehalfOf`.

Long term, ensure that invariants related to borrows, repays, liquidations, eMode, and flash loans are not broken when credit delegation is used.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.

High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Code Stability	Related to the recent frequency of code updates
Decentralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing and Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.

Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and
Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, [`slither-prop`](), that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with [Echidna]() and [Manticore]().

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's [`human-summary`]() printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's [`contract-summary`]() printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's [`human-summary`]() printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's [`human-summary`]() printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Code Quality Recommendations

`SupplyLogic.sol`

- **Execute the `isUsingAsCollaterAny` check first before executing the other checks.** This will short-circuit the execution. Figures D.1 and D.2 show the current implementation. Figure D.3 shows the optimized implementation.

```
((!isolationModeActive && (reserveCache.reserveConfiguration.getDebtCeiling() == 0)) ||
   !userConfig.isUsingAsCollateralAny())
```

Figure D.1: `protocol/logic/SupplyLogic.sol#L72-L73`

```
!isolationModeActive &&
  (reserveCache.reserveConfiguration.getDebtCeiling() == 0 ||
    !userConfig.isUsingAsCollateralAny()),
```

Figure D.2: `protocol/logic/SupplyLogic.sol#L208-L210`

```
!userConfig.isUsingAsCollateralAny() ||
!isolationModeActive && reserveCache.reserveConfiguration.getDebtCeiling() == 0)
```

Figure D.3: Optimized implementation

- **Use `fromConfig`, which is the equivalent of `usersConfig[params.from]`.** This will reduce the likelihood of errors and improve readability.

```
[...]
DataTypes.UserConfigurationMap storage fromConfig = usersConfig[params.from];

   if (fromConfig.isUsingAsCollateral(reserveId)) {
     if (fromConfig.isBorrowingAny()) {
       ValidationLogic.validateHFAndLtv(
         [...]
         usersConfig[params.from],
     [...]
```

Figure D.4: `protocol/logic/SupplyLogic.sol#L155-L163`

- **Remove the `toEModeCategory` field from the `FinalizeTransferParams` struct, as it is never used in `finalizeTransfer`.** This will improve the code quality and reduce gas usage.

`UserConfiguration.sol`

- **Define a constant `MAX_RESERVE_INDEX` variable and use that instead of the hardcoded 128 value that is used in multiple places.** This will improve the code quality and ensure that any future changes to the variable will be automatically applied to all places in which it is used.

`ValidationLogic.sol`

- **In the `validateFlashLoan` function, move the check verifying that the two arrays are of equal length from the end of the function (after the `for` loop) to the beginning of the function.** Failing early is considered best practice.
- **Use `vars.assetUnit`, which is the equivalent of `10**vars.reserveDecimals`.** This will reduce the likelihood of errors and improve readability.

```
function validateBorrow(
   mapping(address => DataTypes.ReserveData) storage reservesData,
   mapping(uint256 => address) storage reserves,
   mapping(uint8 => DataTypes.EModeCategory) storage eModeCategories,
   DataTypes.ValidateBorrowParams memory params
 ) internal view {
   [...]
   unchecked {
     vars.assetUnit = 10**vars.reserveDecimals;
   }
   [...]
   unchecked {
     vars.amountInBaseCurrency /= 10**vars.reserveDecimals;
   }
```

Figure D.5: `protocol/logic/ValidationLogic.sol#L126-L249`

`ReserveLogic.sol`

- **Remove the `avgStableRate` field from the `UpdateInterestRatesLocalVars` struct, as it is never used in `updateInterestRates`.** This will improve the code quality and reduce gas usage.
- **Remove the `avgStableRate` and `stableSupplyUpdatedTimestamp` fields from the `AccrueToTreasuryLocalVars` struct, as they are never used in `_accrueToTreasury`.** This will improve the code quality and reduce gas usage.

`LiquidationLogic.sol`

- **Remove the `userCompoundedBorrowBalance` and `liquidationBonus` fields from the `AvailableCollateralToLiquidateLocalVars` struct, as they are never used in `_calculateAvailableCollateralToLiquidate`.** This will improve the code quality and reduce gas usage.

E. Fix Log

Aave addressed the following issues in the codebase as a result of the assessment. Trail of Bits reviewed each of the fixes.

#	Title	Severity	Status
1	Solidity compiler optimizations can be problematic	Informational	Risk accepted by client
2	Lack of chainID validation allows attackers to reuse signatures across forks	High	Fixed
3	Risks associated with EIP-2612	Informational	Risk accepted by client
4	Insufficient Repay event parameters	Informational	Fixed
5	Base class functions that are used only in a single derived class could cause confusion	Informational	Fixed
6	Use of the constructor rather than the initialize function prevents the incentives controller from being updated after deployment	Low	Fixed
7	Incorrect eMode category fetched by borrow	High	Fixed
8	Missing validation when setting eMode categories	Low	Fixed
9	Missing/incorrect isolation mode checks circumvent collateral isolation mode	High	Fixed
10	Isolation mode bypassed when liquidating and receiving aTokens	High	Fixed
11	Isolation mode total debt does not decrease on liquidation, potentially blocking new loans using the isolated asset	High	Fixed
12	Unclear behavior when calculating interest rates	Informational	Fixed
13	Use of deprecated Chainlink interface and function	Informational	Risk accepted by client

14	Lack of contract existence check on delegatecall	Informational	Risk accepted by client
15	Variable debt token incorrectly tracks debtor's previous index	Medium	Fixed

For additional information, please refer to the Detailed Fix Log.

Detailed Fix Log

This section briefly describes Trail of Bits' review of the fixes made after the assessment.

TOB-AAVE-001: Solidity compiler optimizations can be problematic
Risk accepted. The Aave team provided the following rationale for its acceptance of this risk: "The benefits of optimizations (including reduced code size for deployments) outweighs the risks. The aave community will take care of monitoring the development of the solidity compiler and optimizer to evaluate any future issue."

TOB-AAVE-002: Lack of chainID validation allows attackers to reuse signatures across forks
Fixed. The `chainId` has been made dynamic. If a different `chainId` than the cached one is detected, then a fresh `chainId` is retrieved and used in the domain separator.
https://github.com/aave/aave-v3-core/pull/164

TOB-AAVE-003: Risks associated with EIP-2612
Risk accepted. The Aave team provided the following rationale for its acceptance of this risk: "Signing a message is part of everyday usage of the blockchain and of course users need to be aware of the risk. We will make sure the UI and documentation are explanatory enough for the users to avoid mistakes."

TOB-AAVE-004: Insufficient Repay event parameters
Fixed. The `Repay` event has been updated to include a boolean that indicates which token was used to perform the repay operation. The amount of repayment has also been added as an event parameter.
https://github.com/aave/aave-v3-core/pull/257

TOB-AAVE-005: Base class functions that are used only in a single derived class could cause confusion
Fixed. The contracts have been refactored to remove the unused inherited functions from the `StableDebtToken` contract.
https://github.com/aave/aave-v3-core/pull/553

TOB-AAVE-006: Use of the constructor rather than the initialize function prevents the incentives controller from being updated after deployment
Fixed. The `_addressesProvider` variable has been made `immutable`. Therefore, it is included in the bytecode of the implementation and can be access from the proxy contract.
https://github.com/aave/aave-v3-core/pull/225

TOB-AAVE-007: Incorrect eMode category fetched by borrow
Fixed. The eMode category is now fetched from `onBehalfOf` rather than `msg.sender`.
https://github.com/aave/aave-v3-core/pull/204

TOB-AAVE-008: [Missing validation when setting eMode categories]
Fixed. The `setEModeCategory` function's validation has been updated to check for erroneous loan-to-value settings.
https://github.com/aave/aave-v3-core/pull/207

TOB-AAVE-009: [Missing/incorrect isolation mode checks circumvent collateral isolation mode]
Fixed. The validation that determines whether a certain asset can be activated as collateral (with respect to isolation mode) has been refactored into a separate function. This function is called in the places that require such validation.
https://github.com/aave/aave-v3-core/pull/256

TOB-AAVE-010: [Isolation mode bypassed when liquidating and receiving aTokens]
Fixed. The validation that determines whether a certain asset can be activated as collateral (with respect to isolation mode) has been refactored into a separate function. This function is called to determine whether the aToken can be activated as collateral.
https://github.com/aave/aave-v3-core/pull/256

TOB-AAVE-011: [Isolation mode total debt does not decrease on liquidation, potentially blocking new loans using the isolated asset]
Fixed. The logic that increases and decreases isolation mode debt has been refactored into a separate library. This library is called to correctly update the debt when borrowing and liquidating.
https://github.com/aave/aave-v3-core/pull/251

TOB-AAVE-012: [Unclear behavior when calculating interest rates]
Fixed. The implementation has been updated to adhere to the specification.
https://github.com/aave/aave-v3-core/pull/252

TOB-AAVE-013: [Use of deprecated Chainlink interface and function]
Risk accepted. Aave explained that the recommended Chainlink interface uses more gas than the deprecated interface. The team has therefore decided to refrain from updating to the recommended interface until the Aave community decides to do so.

TOB-AAVE-014: [Lack of contract existence check on delegatecall]
Risk accepted. The Aave team provided the following rationale for its acceptance of this risk: "In V2 there was an attack vector because of this [reported] by ToB some time ago. The attack surface has been removed by getting rid of the `LendingPoolCollateralManager`. All the contracts are now initialized through a factory (either `PoolAddressesProvider` or `PoolConfigurator`)."

TOB-AAVE-015: [Variable debt token incorrectly tracks debtor's previous index]

Fixed. The implementation has been updated to use `onBehalfOf` instead of `user`.
https://github.com/aave/aave-v3-core/pull/295