SMART CONTRACT AUDIT REPORT

for

Aave V3

Prepared By: Yiqun Chen

PeckShield

January 10, 2022

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	Aave V3
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Stephen Bie, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 10, 2022	Xuxian Jiang	Final Release
1.0-rc	November 20, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Aave V3` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Aave V3

`Aave` is a popular decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. The audited `Aave V3` protocol includes new features for new usage scenarios and improves earlier versions on capital efficiency, security, decentralization, UX while at the same time providing new functionalities to leverage the capabilities of rollups and the growing ecosystem of competing L1s. The basic information of Aave V3 is as follows:

Table 1.1: Basic Information of Aave V3

Item	Description
Name	Aave
Website	https://aave.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 10, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/aave/aave-v3-core.git (14f6148)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/aave/aave-v3-core.git (c71d57d)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Aave V3` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	
Medium	3	
Low	4	
Informational	2	
Total	11	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Aave V3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Suggested immutable For Gas Efficiency	Coding Practice	Fixed
PVE-002	Medium	Improved Logic of Pool::_addReserveToList()	Business Logic	Fixed
PVE-003	Medium	Proper And Consistent Collateral Enabling	Business Logic	Fixed
PVE-004	Low	Improvement on UserConfiguration::_getFirstAssetAsCollateralId()	Coding Practice	Fixed
PVE-005	Informational	Redundant State/Code Removal	Coding Practice	Fixed
PVE-006	High	Proper Asset Price in GenericLogic::calculateUserAccountData()	Business Logic	Fixed
PVE-007	Medium	Proper EMode Category Use in Pool::borrow()	Business Logic	Fixed
PVE-008	Low	Possible Underflow Avoidance in BorrowLogic And UserConfiguration	Coding Practices	Confirmed
PVE-009	Low	Consistent Reserve Cache Use in rebalanceStableBorrowRate()	Coding Practice	Fixed
PVE-010	Low	Health Validation in EModeLogic::executeSetUserEMode()	Business Logic	Fixed
PVE-011	High	Potential Reentrancy Risk in flashLoanSimple()	Time and State	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Suggested Constant/Immutable Usages For Gas Efficiency

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show a number of key state variables defined in `PriceOracleSentinel`, including `_addressesProvider`, `_oracle`, and `_gracePeriod`. If there is no need to dynamically update these state variables, they can be declared as either constants or `immutable` for gas efficiency. In particular, the above three states can be defined as `immutable`.

```
contract PriceOracleSentinel is IPriceOracleSentinel {
  IPoolAddressesProvider public _addressesProvider;
  ISequencerOracle public _oracle;
  uint256 public _gracePeriod;
  ...
```

```
}
```

Listing 3.1: The `PriceOracleSentinel` Contract

Similarly, the `_addressesProvider` state in `AaveOracle` and `ACLManager` can be defined as `immutable` for gas efficiency.

Recommendation Revisit the state variable definition and make extensive use of `constant`/ `immutable` states.

Status The issue has been fixed by the following PR: `214`.

3.2 Improved Logic of Pool::_addReserveToList()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `Pool`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

The `Aave` protocol allows the governance to dynamically add new reserves into the protocol. To keep track of the list of active reserves, the protocol maintains the internal state `_reservesList`. While reviewing the accounting of active reserves, we notice the internal routine to add a new reserve needs to be improved.

To elaborate, we show below the `_addReserveToList()` function. It implements a rather straightforward logic in validating the new asset and then adding it into the internal `_reservesList`. It comes to our attention that the internal `for`-loop needs to terminate the execution once a vacant spot is located and populated. Note the current implementation will simply fill all available slots with the new reserve asset.

```
function _addReserveToList(address asset) internal {
  uint256 reservesCount = _reservesCount;

  require(reservesCount < _maxNumberOfReserves, Errors.P_NO_MORE_RESERVES_ALLOWED);

  bool reserveAlreadyAdded = _reserves[asset].id != 0  _reservesList[0] == asset;

  if (!reserveAlreadyAdded) {
    for (uint8 i = 0; i <= reservesCount; i++) {
      if (_reservesList[i] == address(0)) {
        _reserves[asset].id = i;
        _reservesList[i] = asset;
        _reservesCount = reservesCount + 1;
```

```
        }
      }
    }
  }
```

Listing 3.2: `Pool::_addReserveToList()`

Recommendation Revise the above `_addReserveToList()` function to proper add a new reserve asset.

Status The issue has been fixed by the following PR: `196`.

3.3 Proper And Consistent Collateral Enabling

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

The `Aave` protocol supports dynamic updates on the set of assets that are considered as collateral. This is important as these collateral assets directly determine the borrowing power of the respective users. In addition, these collateral assets have profound implications on the new features on `eMode` and `isolation mode`. While reviewing the logic to enable a collateral, we observe unnecessary inconsistency that may introduce unwanted confusion and errors.

To elaborate, we show below the `executeSupply()` function from the `SupplyLogic` library. To turn on the collateral, the logic requires `(!isolationModeActive && (reserveCache.reserveConfiguration.getDebtCeiling()== 0))` or `!userConfig.isUsingAsCollateralAny()` (lines 72 – 73).

```
46  function executeSupply(
47    mapping(address => DataTypes.ReserveData) storage reserves,
48    mapping(uint256 => address) storage reservesList,
49    DataTypes.UserConfigurationMap storage userConfig,
50    DataTypes.ExecuteSupplyParams memory params
51  ) external {
52    DataTypes.ReserveData storage reserve = reserves[params.asset];
53    DataTypes.ReserveCache memory reserveCache = reserve.cache();
54
55    reserve.updateState(reserveCache);
56
57    ValidationLogic.validateSupply(reserveCache, params.amount);
58
59    reserve.updateInterestRates(reserveCache, params.asset, params.amount, 0);
```

```
  IERC20(params.asset).safeTransferFrom(msg.sender, reserveCache.aTokenAddress, params
      .amount);

  bool isFirstSupply = IAToken(reserveCache.aTokenAddress).mint(
    params.onBehalfOf,
    params.amount,
    reserveCache.nextLiquidityIndex
  );

  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);
    }
  }

  emit Supply(params.asset, msg.sender, params.onBehalfOf, params.amount, params.
      referralCode);
}
```

Listing 3.3: `SupplyLogic::executeSupply()`

However, if we examine another function `mintUnbacked()` from the `BorrowLogic` library, the logic simply requires it is the `isFirstSupply`. The inconsistency on the same collateral-enabling logic among current libraries `SupplyLogic`, `BridgeLogic`, and `LiquidationLogic` needs to be resolved before production deployment.

```
function mintUnbacked(
  DataTypes.ReserveData storage reserve,
  DataTypes.UserConfigurationMap storage userConfig,
  address asset,
  uint256 amount,
  address onBehalfOf,
  uint16 referralCode
) external {
  DataTypes.ReserveCache memory reserveCache = reserve.cache();

  reserve.updateState(reserveCache);

  ValidationLogic.validateSupply(reserveCache, amount);

  uint256 unbackedMintCap = reserveCache.reserveConfiguration.getUnbackedMintCap();
  uint256 reserveDecimals = reserveCache.reserveConfiguration.getDecimals();

  uint256 unbacked = reserve.unbacked = reserve.unbacked + Helpers.castUint128(amount)
```

```
      ;

    require(
      unbackedMintCap > 0 && unbacked / (10**reserveDecimals) < unbackedMintCap,
      Errors.VL_UNBACKED_MINT_CAP_EXCEEDED
    );

    reserve.updateInterestRates(reserveCache, asset, 0, 0);

    bool isFirstSupply = IAToken(reserveCache.aTokenAddress).mint(
      onBehalfOf,
      amount,
      reserveCache.nextLiquidityIndex
    );

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

    emit MintUnbacked(asset, msg.sender, onBehalfOf, amount, referralCode);
  }
  }
```

Listing 3.4: `BridgeLogic::mintUnbacked()`

Recommendation Revise the above functions to be consistent on the enabling of a specific asset as collateral.

Status The issue has been fixed by the following PR: `256`.

3.4 Improvement on UserConfiguration::_getFirstAssetAsCollateralId()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UserConfiguration`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [7]

Description

The `Aave` protocol has a flexible mechanism to keep track of the configuration of current protocol users. This mechanism is mainly implemented in the `UserConfiguration` contract. In the process of reviewing this contract, we notice an internal helper function can be simplified.

To elaborate, we show below this helper routine, i.e., `_getFirstAssetAsCollateralId()`. As the name indicates, this routine is designed to return the address of the first asset used as collateral by the user. It turns out the `collateralData & ~(collateralData - 1)` computation is unnecessary and the step size of 2 can be avoided as well.

```
function _getFirstAssetAsCollateralId(DataTypes.UserConfigurationMap memory self)
  internal
  pure
  returns (uint256)
{
  unchecked {
    uint256 collateralData = self.data & COLLATERAL_MASK;
    uint256 firstCollateralPosition = collateralData & ~(collateralData - 1);
    uint256 id;

    while ((firstCollateralPosition >>= 2) > 0) {
      id += 2;
    }
    return id / 2;
  }
}
```

Listing 3.5: `UserConfiguration::_getFirstAssetAsCollateralId()`

Recommendation Simplify the above routine as the follows:

```
function _getFirstAssetAsCollateralId(DataTypes.UserConfigurationMap memory self)
  internal
  pure
  returns (uint256)
{
    uint256 collateralData = self.data & COLLATERAL_MASK;
    uint256 id;

    while ((collateralData >>= 2) > 0) {
      id += 1;
    }
    return id;
}
```

Listing 3.6: `UserConfiguration::_getFirstAssetAsCollateralId()`

Status The issue has been fixed by the following PR: `261`.

3.5 Redundant State/Code Removal

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The `Aave` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `Pool` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `ReserveLogic` library, there is an `AccrueToTreasuryLocalVars` structure with a number of member fields that are defined, but not used. Examples include the `avgStableRate` and `stableSupplyUpdatedTimestamp` fields. Also, another structure `UpdateInterestRatesLocalVars` defines an unused member field `avgStableRate`.

```
struct AccrueToTreasuryLocalVars {
  uint256 prevTotalStableDebt;
  uint256 prevTotalVariableDebt;
  uint256 currTotalVariableDebt;
  uint256 avgStableRate;
  uint256 cumulatedStableInterest;
  uint256 totalDebtAccrued;
  uint256 amountToMint;
  uint40 stableSupplyUpdatedTimestamp;
}
```

Listing 3.7: The `ReserveLogic` Library

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by the following PR: `212`.

3.6 Proper Asset Price in GenericLogic::calculateUserAccountData()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GenericLogic`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

For any lending protocol, there is a need to reliably and accurately measure the borrower's debt position and provide necessary means to liquidate underwater positions. The `Aave` protocol is no exception. While reviewing the implementation to measure the debt position, we notice the key function `calculateUserAccountData()` needs to be improved.

To illustrate, we show below this function. As the name indicates, the function is dedicated to calculate the user data across the reserves. For this end, it requires the total liquidity/collateral/borrow balances in the base currency used by the price feed, as well as the average `loan to value (LVT)`, the average liquidation ratio, and the health factor. However, it misuses the `eModeAssetPrice` as the price for each iterated reserve (lines 134-136), which leads to erroneous calculation of collateral value and borrow power. This issue is possibly introduced to support the `eMode` feature, but has been mistakenly used to consider all reserve assets to be part of the same `eMode` category.

```
68  function calculateUserAccountData(
69    mapping(address => DataTypes.ReserveData) storage reservesData,
70    mapping(uint256 => address) storage reserves,
71    mapping(uint8 => DataTypes.EModeCategory) storage eModeCategories,
72    DataTypes.CalculateUserAccountDataParams memory params
73  )
74    internal
75    view
76    returns (
77      uint256,
78      uint256,
79      uint256,
80      uint256,
81      uint256,
82      bool
83    )
84  {
85    if (params.userConfig.isEmpty()) {
86      return (0, 0, 0, 0, type(uint256).max, false);
87    }
88
89    CalculateUserAccountDataVars memory vars;
```

```
    if (params.userEModeCategory != 0) {
      vars.eModePriceSource = eModeCategories[params.userEModeCategory].priceSource;
      vars.eModeLtv = eModeCategories[params.userEModeCategory].ltv;
      vars.eModeLiqThreshold = eModeCategories[params.userEModeCategory].
          liquidationThreshold;

      if (vars.eModePriceSource != address(0)) {
        vars.eModeAssetPrice = IPriceOracleGetter(params.oracle).getAssetPrice(
          vars.eModePriceSource
        );
      }
    }

    while (vars.i < params.reservesCount) {
      if (!params.userConfig.isUsingAsCollateralOrBorrowing(vars.i)) {
        unchecked {
          ++vars.i;
        }
        continue;
      }

      vars.currentReserveAddress = reserves[vars.i];

      if (vars.currentReserveAddress == address(0)) {
        unchecked {
          ++vars.i;
        }
        continue;
      }

      DataTypes.ReserveData storage currentReserve = reservesData[vars.
          currentReserveAddress];

      (
        vars.ltv,
        vars.liquidationThreshold,
        ,
        vars.decimals,
        ,
        vars.eModeAssetCategory
      ) = currentReserve.configuration.getParams();

      unchecked {
        vars.assetUnit = 10**vars.decimals;
      }
      vars.assetPrice = vars.eModeAssetPrice > 0
        ? vars.eModeAssetPrice
        : IPriceOracleGetter(params.oracle).getAssetPrice(vars.currentReserveAddress);
    ...
  }
```

Listing 3.8: `GenericLogic::calculateUserAccountData`

Recommendation Apply the right price oracle in the above `calculateUserAccountData()` routine to compute the user account data.

Status The issue has been fixed by the following PR: `262`.

3.7 Proper EMode Category Use in Pool::borrow()

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Pool`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

The `Aave` protocol has a nice feature `credit delegation`, which allows a credit delegator to delegate the credit of their account's position to a borrower. This feature requires proper accounting of delegation allowance and actual expenditure. While examining its implementation, we notice a key function `borrow()` does not properly follow the `credit delegation` logic.

To elaborate, we show below this `borrow()` function. This is a core lending function and is used to borrow funds from the lending protocol. It comes to our attention that the encapsulated `DataTypes.ExecuteBorrowParams` parameters mistakenly uses `_usersEModeCategory[msg.sender]` as the user's `eMode` category. In the `credit delegation` situation, the real `eMode` category should be `_usersEModeCategory[onBehalfOf]`.

```
/// @inheritdoc IPool
function borrow(
  address asset,
  uint256 amount,
  uint256 interestRateMode,
  uint16 referralCode,
  address onBehalfOf
) external override {
  BorrowLogic.executeBorrow(
    _reserves,
    _reservesList,
    _eModeCategories,
    _usersConfig[onBehalfOf],
    DataTypes.ExecuteBorrowParams(
      asset,
      msg.sender,
      onBehalfOf,
      amount,
      interestRateMode,
      referralCode,
```

```
            true,
            _maxStableRateBorrowSizePercent,
            _reservesCount,
            _addressesProvider.getPriceOracle(),
            _usersEModeCategory[msg.sender],
            _addressesProvider.getPriceOracleSentinel()
          )
        );
      }
```

Listing 3.9: `Pool::borrow()`

Recommendation Ensure the `credit delegation` feature is consistently honored in all aspects of the lending protocol.

Status The issue has been fixed by the following PR: `204`.

3.8 Possible Underflow Avoidance in BorrowLogic And UserConfiguration

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BorrowLogic`, `UserConfiguration`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [3]

Description

The `Aave` protocol has established itself as the leading lending protocol. Within each lending protocol, there is a constant need of accommodating various precision issues. `SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. Since the version `0.8.0`, `Solidity` includes checked arithmetic operations by default, and this largely renders `SafeMath` unnecessary. While reviewing arithmetic operations in current implementation, we notice occasions that may introduce unexpected overflows/underflows.

For example, if we examine the `isUsingAsCollateralOne()` function, it may revert if the current `collateralData` (line 121) is equal to 0. Another example is when the underlying asset of a reserve has an unusual decimal, which may revert the following calculation of `reserveCache.reserveConfiguration.getDecimals()- ReserveConfiguration.DEBT_CEILING_DECIMALS`. Note this calculation appears in a number of routines. Its revert may bring in unnecessary frictions and cause issues for integration and composability.

```
function isUsingAsCollateralOne(DataTypes.UserConfigurationMap memory self)
  internal
  pure
  returns (bool)
{
  uint256 collateralData = self.data & COLLATERAL_MASK;
  return collateralData & (collateralData - 1) == 0;
}
```

Listing 3.10: `UserConfiguration::isUsingAsCollateralOne()`

Recommendation Revise the above calculation to avoid the unnecessary overflows and underflows.

Status The issue has been confirmed.

3.9 Consistent Reserve Cache Use in rebalanceStableBorrowRate()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BorrowLogic`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [3]

Description

For gas efficiency, the `Aave` protocol is engineered with the reserve cache mechanism, which necessitates the common steps to be followed when operating with the reserve data in different scenarios, including the cache generation, update, and eventual persistence. However, our analysis shows certain inconsistency in the reserve cache usages and the inconsistency needs to be resolved to avoid confusions and errors.

To elaborate, we show below two functions `executeSupply()` and `rebalanceStableBorrowRate()`. These functions are self-explanatory and it comes to our attention that the first function updates the reserve cache before applying the validation logic while the second function validates the reserve cache before updating it. As mentioned earlier, this inconsistency may introduce issues when using the stale cache state for validation.

```
function executeSupply(
  mapping(address => DataTypes.ReserveData) storage reserves,
  mapping(uint256 => address) storage reservesList,
  DataTypes.UserConfigurationMap storage userConfig,
  DataTypes.ExecuteSupplyParams memory params
```

```
  ) external {
    DataTypes.ReserveData storage reserve = reserves[params.asset];
    DataTypes.ReserveCache memory reserveCache = reserve.cache();

    reserve.updateState(reserveCache);

    ValidationLogic.validateSupply(reserveCache, params.amount);

    reserve.updateInterestRates(reserveCache, params.asset, params.amount, 0);

    ...
}
```

Listing 3.11: `SupplyLogic::executeSupply()`

```
  function rebalanceStableBorrowRate(
    DataTypes.ReserveData storage reserve,
    address asset,
    address user
  ) external {
    DataTypes.ReserveCache memory reserveCache = reserve.cache();

    IERC20 stableDebtToken = IERC20(reserveCache.stableDebtTokenAddress);
    IERC20 variableDebtToken = IERC20(reserveCache.variableDebtTokenAddress);
    uint256 stableDebt = IERC20(stableDebtToken).balanceOf(user);

    ValidationLogic.validateRebalanceStableBorrowRate(
      reserve,
      reserveCache,
      asset,
      stableDebtToken,
      variableDebtToken,
      reserveCache.aTokenAddress
    );

    reserve.updateState(reserveCache);
    ...
}
```

Listing 3.12: `BorrowLogic::rebalanceStableBorrowRate()`

Recommendation Revise the above functions to following a consistent approach to use the reserve cache mechanism.

Status The issue has been fixed by the following PR: `213`.

3.10 Health Validation in EModeLogic::executeSetUserEMode()

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: EModeLogic
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [6]

Description

As mentioned earlier, the Aave protocol introduces a new feature eMode, which stands for High Efficiency Mode and allows borrowers to extract the highest borrowing power out of their collateral. In other words, assets can now be categorized, with each category having a shared risk management parameters, e.g., LTV, liquidation threshold, liquidation bonus, and a custom price oracle. While reviewing the current eMode support, we notice the related executeSetUserEMode() function needs to be revisited.

```
function executeSetUserEMode(
  mapping(address => DataTypes.ReserveData) storage reserves,
  mapping(uint256 => address) storage reservesList,
  mapping(uint8 => DataTypes.EModeCategory) storage eModeCategories,
  mapping(address => uint8) storage usersEModeCategory,
  DataTypes.UserConfigurationMap storage userConfig,
  DataTypes.ExecuteSetUserEModeParams memory params
) external {
  ValidationLogic.validateSetUserEMode(
    reserves,
    reservesList,
    eModeCategories,
    userConfig,
    params.reservesCount,
    params.categoryId
  );

  uint8 prevCategoryId = usersEModeCategory[msg.sender];
  usersEModeCategory[msg.sender] = params.categoryId;

  if (prevCategoryId != 0 && params.categoryId == 0) {
    ValidationLogic.validateHealthFactor(
      reserves,
      reservesList,
      eModeCategories,
      userConfig,
      msg.sender,
      params.categoryId,
      params.reservesCount,
      params.oracle
    );
```

```
64     }
65     emit UserEModeSet(msg.sender, params.categoryId);
66   }
```

Listing 3.13: `EModeLogic::executeSetUserEMode()`

To elaborate, we show above its full implementation, which indicates that the health check is not performed unless the following condition is satisfied `prevCategoryId != 0 && params.categoryId == 0` (line 53). However, in the situation where `params.categoryId !=0`, there is still a need to invoke `validateHalthFactor()`. The reason is that the new category may have different `LTV` and/or `liquidationThreshold`.

Recommendation Revise the above `executeSetUserEMode()` function to enforce the health check.

Status The issue has been fixed by the following PR: `218`.

3.11 Potential Reentrancy Risk in flashLoanSimple()

- ID: PVE-011
- Severity: High
- Likelihood: High
- Impact: High
- Target: `FlashLoanLogic`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasions where the `checks-effects-interactions` principle is violated. Using the `FlashLoanLogic` as an example, the `flashLoanSimple()` function (see the code snippet below) is provided to deposit additional tokens into the option contract. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

In particular, the interaction with the external contract inside `flashLoanSimple()` (line 201) starts before effecting the update on the internal state. More importantly, it carries over the stale cache states and apply them for the calculation of protocol-wide interest rates, while ignoring the possibility that it may re-enter to update the protocol state. These updates may be lost since they are overwritten by the stale states!

```
function executeFlashLoanSimple(
  DataTypes.ReserveData storage reserve,
  DataTypes.FlashloanSimpleParams memory params
) external {
  FlashLoanSimpleLocalVars memory vars;

  DataTypes.ReserveCache memory reserveCache = reserve.cache();
  reserve.updateState(reserveCache);

  ValidationLogic.validateFlashloanSimple(reserveCache);

  vars.receiver = IFlashLoanSimpleReceiver(params.receiverAddress);

  vars.totalPremium = params.amount.percentMul(params.flashLoanPremiumTotal);
  vars.amountPlusPremium = params.amount + vars.totalPremium;
  IAToken(reserveCache.aTokenAddress).transferUnderlyingTo(params.receiverAddress,
      params.amount);

  require(
    vars.receiver.executeOperation(
      params.asset,
      params.amount,
      vars.totalPremium,
      msg.sender,
      params.params
    ),
    Errors.P_INVALID_FLASH_LOAN_EXECUTOR_RETURN
  );

  vars.premiumToProtocol = params.amount.percentMul(params.flashLoanPremiumToProtocol)
      ;
  vars.premiumToLP = vars.totalPremium - vars.premiumToProtocol;

  reserve.cumulateToLiquidityIndex(
    IERC20(reserveCache.aTokenAddress).totalSupply(),
    vars.premiumToLP
  );

  reserve.accruedToTreasury =
    reserve.accruedToTreasury +
    Helpers.castUint128(vars.premiumToProtocol.rayDiv(reserve.liquidityIndex));

  reserve.updateInterestRates(reserveCache, params.asset, vars.amountPlusPremium, 0);

  IERC20(params.asset).safeTransferFrom(
    params.receiverAddress,
    reserveCache.aTokenAddress,
    vars.amountPlusPremium
```

```
    );

    emit FlashLoan(
      params.receiverAddress,
      msg.sender,
      params.asset,
      params.amount,
      vars.totalPremium,
      0
    );
  }
```

Listing 3.14: `FlashLoanLogic::flashLoanSimple()`

Recommendation Revise the above `flashLoanSimple()` routine by applying necessary reentrancy prevention to avoid the use of cached state to overwrite legitimate protocol updates.

Status The issue has been fixed by the following PR: `#201`.

4 Conclusion

In this audit, we have analyzed the Aave V3 design and implementation. The system presents a unique, robust offering as a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Aave V3 improves early versions on capital efficiency, security, decentralization, UX while at the same time providing new functionalities to leverage the capabilities of rollups and the growing ecosystem of competing L1s. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[6] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[11] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[13] PeckShield. PeckShield Inc. https://www.peckshield.com.

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.