SMART CONTRACT AUDIT REPORT

for

AAVE

Prepared By: Shuxiao Wang

Hangzhou, China
December 3, 2020

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	Aave V2
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 3, 2020	Xuxian Jiang	Final Release
1.0-rc2	November 4, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	November 2, 2020	Xuxian Jiang	Release Candidate #1
0.5	October 28, 2020	Xuxian Jiang	Add More Findings #4
0.4	October 20, 2020	Xuxian Jiang	Add More Findings #3
0.3	October 13, 2020	Xuxian Jiang	Add More Findings #2
0.2	October 6, 2020	Xuxian Jiang	Add More Findings #1
0.1	September 29, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Aave V2** design document and related smart contract source code, 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 V2

Aave is a 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. Aave V2 not only addresses some of the suboptimal solutions implemented in V1 (e.g., by allowing for `AToken` upgradeability and simplified overall architecture amenable for automated fuzzers and formal verification tools), but also provides additional features, e.g., debt tokenization, collateral trading, and new flashloans.

The basic information of Aave V2 is as follows:

Table 1.1: Basic Information of Aave V2

Item	Description
Issuer	Aave
Website	https://aave.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 3, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Aave V2 assumes a trusted price oracle with timely market price feeds for supported assets and a lending oracle with timely market lending rates. These two oracles are not part of this audit.

- https://github.com/aave/protocol-v2.git (f756f44)

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

- https://github.com/aave/protocol-v2.git (7509203)

1.2 About PeckShield

PeckShield Inc. [19] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 [14]:

- 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.

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) [13], 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 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.

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

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Aave V2 implementation. 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	1	■
High	2	■ ■
Medium	6	■ ■ ■ ■ ■ ■
Low	8	■ ■ ■ ■ ■ ■ ■ ■
Informational	5	■ ■ ■ ■ ■
Total	22	

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 1 critical-severity vulnerability, 2 high-severity vulnerabilities, 6 medium-severity vulnerabilities, 8 low-severity vulnerabilities, and 5 informational recommendations.

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.

Table 2.1: Key Aave V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Sanity Checks of registerAddressesProvider()	Coding Practices	Fixed
PVE-002	Low	Race Condition Between delegateBorrowAllowance() And borrow()	Time and State	Confirmed
PVE-003	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-004	Low	Simplification And Improvement of the repay() Logic	Coding Practices	Fixed
PVE-005	Medium	Validation of transferFrom() Return Values	Coding Practices	Fixed
PVE-006	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-007	Low	Improved STABLE_BORROWING_MASK	Numeric Errors	Fixed
PVE-008	Low	Inaccurate Burn Events in AToken	Business Logic	Fixed
PVE-009	Informational	Asset Consistency Between Reserve and AToken	Time and State	Fixed
PVE-010	Medium	Inaccurate Calculation of Mints To Treasury	Business Logic	Fixed
PVE-011	High	Premature Updates of updateInterestRates() Before DebtToken Changes	Business Logic	Fixed
PVE-012	High	Late Updates of updateInterestRates() After AToken Changes	Business Logic	Fixed
PVE-013	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-014	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-015	Critical	Possible Fund Loss From (Permissive) Smart Wallets With Allowances to LendingPool	Business Logic	Fixed
PVE-016	Medium	Improved Business Logic in validateWithdraw()	Business Logic	Fixed
PVE-017	Informational	Improved Event Generation With Indexed Assets	Business Logic	Fixed
PVE-018	Low	Performance Optimization in _updateIndexes()	Coding Practices	Fixed
PVE-019	Low	Inconsistent Handling of healthFactor Corner Cases	Coding Practices	Fixed
PVE-020	Medium	Inaccurate previousStableDebt Calculation in _mintToTreasury()	Business Logic	Fixed
PVE-021	Medium	Flashloan-Lowered StableBorrowRate For Mode-Switching Users	Time and State	Confirmed
PVE-022	Medium	Bypassed Enforcement of LIQUIDATION_CLOSE_FACTOR_PERCENT	Business Logic	Fixed

3 | Detailed Results

3.1 Improved Sanity Checks of registerAddressesProvider()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LendingPoolAddressesProviderRegistry`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

Aave V2 implements a modular architecture that makes the entire protocol extensible and pluggable. To facilitate the modular architecture, Aave V2 has a contract named `LendingPoolAddressesProviderRegistry` that contains the list of active addresses providers. Each address provider has an internal mapping record that can be used to retrieve the current implementation or logic contract for different components in Aave V2.

During our analysis of `LendingPoolAddressesProviderRegistry`, we notice that it has a restricted public routine, i.e., `registerAddressesProvider()`. This routine can only be invoked by the privileged owner and, as the name indicates, allows for the registration of a new address provider. Each registered address provider has an associated `id` for unique identification.

```
/**
 * @dev adds a lending pool to the list of registered lending pools
 * @param provider the pool address to be registered
 **/
function registerAddressesProvider(address provider, uint256 id) external override
    onlyOwner {
  _addressesProviders[provider] = id;
  _addToAddressesProvidersList(provider);
  emit AddressesProviderRegistered(provider);
}

/**
 * @dev removes a lending pool from the list of registered lending pools
```

```
  * @param provider the pool address to be unregistered
  **/
function unregisterAddressesProvider(address provider) external override onlyOwner {
  require(_addressesProviders[provider] > 0, Errors.PROVIDER_NOT_REGISTERED);
  _addressesProviders[provider] = 0;
  emit AddressesProviderUnregistered(provider);
}
```

Listing 3.1: LendingPoolAddressesProviderRegistry.sol

When there is a need to unregister a previously registered address provider, the corresponding `id` is simply reset to 0. With that, there is a need to apply additional sanity checks in `registerAddressesProvider()` to ensure the associated `id` is not equal to 0.

Recommendation Ensure the associated `id>0` for the registered address provider as follows:

```
/**
  * @dev adds a lending pool to the list of registered lending pools
  * @param provider the pool address to be registered
  **/
function registerAddressesProvider(address provider, uint256 id) external override
    onlyOwner {
  require(id !=0, Errors.PROVIDER_NOT_REGISTERED);
  _addressesProviders[provider] = id;
  _addToAddressesProvidersList(provider);
  emit AddressesProviderRegistered(provider);
}

/**
  * @dev removes a lending pool from the list of registered lending pools
  * @param provider the pool address to be unregistered
  **/
function unregisterAddressesProvider(address provider) external override onlyOwner {
  require(_addressesProviders[provider] > 0, Errors.PROVIDER_NOT_REGISTERED);
  _addressesProviders[provider] = 0;
  emit AddressesProviderUnregistered(provider);
}
```

Listing 3.2: LendingPoolAddressesProviderRegistry.sol (revised)

Status The issue has been confirmed and accordingly fixed by this merge request: `82`.

3.2 Race Condition Between delegateBorrowAllowance() And borrow()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LendingPool`
- Category: Time and State [9]
- CWE subcategory: CWE-362 [5]

Description

`LendingPool` is a core pool contract in Aave V2 that implements a variety of innovative features. One of them is the so-called `credit delegation`, which in essence allows an user to take uncollateralized loans as long as the user receives delegation from other users that provide the collateral. The feature is mainly implemented with a pair of related routines, i.e., `delegateBorrowAllowance()` and `borrow()`.

To elaborate, we show below related code snippet of these two routines. The `delegateBorrowAllowance()` routine sets the intended allowance (`_borrowAllowance` at line 197) to borrow on a certain type of debt asset for a specific user address while the allowance will be reduced when the user indeed requests to `borrow()` from the pool.

```
181   /**
182    * @dev Sets allowance to borrow on a certain type of debt asset for a certain user
          address
183    * @param asset The underlying asset of the debt token
184    * @param user The user to give allowance to
185    * @param interestRateMode Type of debt: 1 for stable, 2 for variable
186    * @param amount Allowance amount to borrow
187    **/
188   function delegateBorrowAllowance(
189     address asset,
190     address user,
191     uint256 interestRateMode,
192     uint256 amount
193   ) external override {
194     _whenNotPaused();
195     address debtToken = _reserves[asset].getDebtTokenAddress(interestRateMode);

197     _borrowAllowance[debtToken][msg.sender][user] = amount;
198     emit BorrowAllowanceDelegated(asset, msg.sender, user, interestRateMode, amount);
199   }

201   /**
202    * @dev Allows users to borrow a specific amount of the reserve currency, provided
          that the borrower
203    * already deposited enough collateral.
204    * @param asset the address of the reserve
```

```
   * @param amount the amount to be borrowed
   * @param interestRateMode the interest rate mode at which the user wants to borrow.
       Can be 0 (STABLE) or 1 (VARIABLE)
   * @param referralCode a referral code for integrators
   * @param onBehalfOf address of the user who will receive the debt
   **/
  function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
  ) external override {
    _whenNotPaused();
    ReserveLogic.ReserveData storage reserve = _reserves[asset];

    if (onBehalfOf != msg.sender) {
      address debtToken = reserve.getDebtTokenAddress(interestRateMode);

      _borrowAllowance[debtToken][onBehalfOf][msg
        .sender] = _borrowAllowance[debtToken][onBehalfOf][msg.sender].sub(
        amount,
        Errors.BORROW_ALLOWANCE_ARE_NOT_ENOUGH
      );
    }
    _executeBorrow(
      ExecuteBorrowParams(
        asset,
        msg.sender,
        onBehalfOf,
        amount,
        interestRateMode,
        reserve.aTokenAddress,
        referralCode,
        true
      )
    );
  }
```

Listing 3.3: LendingPool.sol

This pair of routines resembles the ERC20-specified `approve()` / `transferFrom()` pair and shares a similar known race condition issue [2]. Specifically, when a user intends to reduce the `_borrowAllowance` borrow amount previously approved from, say, 10 `DAI` to 1 `DAI`. The user may race to borrow up to the previously approved `_borrowAllowance` (the 10 `DAI`) and then additionally borrow the new amount just approved (1 `DAI`). This breaks the user's intention of restricting the borrow allowance to the new amount, **not** the sum of old amount and new amount.

In order to properly approve the `_borrowAllowance`, there also exists a known workaround: users can utilize the `increaseBorrowApproval()` and `decreaseBorrowApproval()` functions versus the traditional `delegateBorrowAllowance()` function.

Recommendation Add the suggested workaround functions `increaseBorrowApproval()` and `decreaseBorrowApproval()`. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

Status This issue has been confirmed. Like in the `approval()`/`transferFrom()` pattern, there is no easy fix. The team plans to make sure builders and users are aware of this limitation.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

In Aave V2, the `LendingPool` contract is designed to be the main entry for interaction with borrowing/lending users. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and mints the corresponding `AToken` to represent the depositor's share in the lending pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of Aave V2. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
/**
 * @dev deposits The underlying asset into the reserve. A corresponding amount of the
    overlying asset (aTokens)
 * is minted.
 * @param asset the address of the reserve
 * @param amount the amount to be deposited
 * @param referralCode integrators are assigned a referral code and can potentially
    receive rewards.
 **/
function deposit(
  address asset,
  uint256 amount,
  address onBehalfOf,
  uint16 referralCode
) external override {
  _whenNotPaused();
  ReserveLogic.ReserveData storage reserve = _reserves[asset];
```

```
    ValidationLogic.validateDeposit(reserve, amount);

    address aToken = reserve.aTokenAddress;

    reserve.updateState();
    reserve.updateInterestRates(asset, aToken, amount, 0);

    bool isFirstDeposit = IAToken(aToken).balanceOf(onBehalfOf) == 0;
    if (isFirstDeposit) {
      _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
    }

    IAToken(aToken).mint(onBehalfOf, amount, reserve.liquidityIndex);

    //transfer to the aToken contract
    IERC20(asset).safeTransferFrom(msg.sender, aToken, amount);

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

Listing 3.4: LendingPool.sol

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Aave V2 for borrowing/lending. In fact, Aave V2 is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been acknowledged by the team. Since a specific AToken can be developed for each asset, a different AToken implementation will be used to eventually list balance changing tokens.

3.4 Simplification And Improvement of the repay() Logic

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LendingPool
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

The LendingPool contract implements a number of core functionalities in borrowing and lending. One of them is the repay() operation that allows to repay partial or full amount of a borrower's debt position. While reviewing the repay() logic, we notice that its execution logic can be further improved.

To elaborate, we show below the code snippet of repay(). The execution logic is rather straightforward in firstly updating/validating the borrower's debt position, then calculating either partial or full paybackAmount and performing the intended repayment, and finally updating the pool's interest rates and redirecting the payment to the AToken contract.

```
function repay(
  address asset,
  uint256 amount,
  uint256 rateMode,
  address onBehalfOf
) external override {
  _whenNotPaused();

  ReserveLogic.ReserveData storage reserve = _reserves[asset];

  (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(onBehalfOf,
      reserve);

  ReserveLogic.InterestRateMode interestRateMode = ReserveLogic.InterestRateMode(
      rateMode);

  //default to max amount
  uint256 paybackAmount = interestRateMode == ReserveLogic.InterestRateMode.STABLE
    ? stableDebt
    : variableDebt;

  if (amount != type(uint256).max && amount < paybackAmount) {
```

```
271       paybackAmount = amount;
272     }

274     ValidationLogic.validateRepay(
275       reserve,
276       amount,
277       interestRateMode,
278       onBehalfOf,
279       stableDebt,
280       variableDebt
281     );

283     reserve.updateState();

285     //burns an equivalent amount of debt tokens
286     if (interestRateMode == ReserveLogic.InterestRateMode.STABLE) {
287       IStableDebtToken(reserve.stableDebtTokenAddress).burn(onBehalfOf, paybackAmount);
288     } else {
289       IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
290         onBehalfOf,
291         paybackAmount,
292         reserve.variableBorrowIndex
293       );
294     }

296     address aToken = reserve.aTokenAddress;
297     reserve.updateInterestRates(asset, aToken, paybackAmount, 0);

299     if (stableDebt.add(variableDebt).sub(paybackAmount) == 0) {
300       _usersConfig[onBehalfOf].setBorrowing(reserve.id, false);
301     }

303     IERC20(asset).safeTransferFrom(msg.sender, aToken, paybackAmount);

305     emit Repay(asset, onBehalfOf, msg.sender, paybackAmount);
306   }
```

Listing 3.5: LendingPool.sol

The optimization is related to the `paybackAmount` calculation (lines 270 – 272): `if (amount != type(uint256).max && amount < paybackAmount) paybackAmount = amount`. The condition of `amount != type(uint256).max` is essentially a `no-op` and therefore the calculation can be simplified as `if (amount < paybackAmount) paybackAmount = amount`.

Moreover, Aave V2 provides a number of helper routines to validate the given arguments to a number of core operations, including `deposit()`, `withdraw()`, `borrow()`, `repay()`, and etc. In `repay()`, the way to perform `validateRepay()` needs to be revised. In particular, the amount to be included in `validateRepay()` should not be the function argument (lines 274 – 281). Instead, it should be the `paybackAmount` (lines 266 – 272).

Recommendation Revise the `repay()` logic as follows:

```
function repay(
  address asset,
  uint256 amount,
  uint256 rateMode,
  address onBehalfOf
) external override {
  _whenNotPaused();

  ReserveLogic.ReserveData storage reserve = _reserves[asset];

  (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(onBehalfOf,
      reserve);

  ReserveLogic.InterestRateMode interestRateMode = ReserveLogic.InterestRateMode(
      rateMode);

  //default to max amount
  uint256 paybackAmount = interestRateMode == ReserveLogic.InterestRateMode.STABLE
    ? stableDebt
    : variableDebt;

  if (amount < paybackAmount) {paybackAmount = amount; }

  ValidationLogic.validateRepay(
    reserve,
    paybackAmount,
    interestRateMode,
    onBehalfOf,
    stableDebt,
    variableDebt
  );

  reserve.updateState();

  //burns an equivalent amount of debt tokens
  if (interestRateMode == ReserveLogic.InterestRateMode.STABLE) {
    IStableDebtToken(reserve.stableDebtTokenAddress).burn(onBehalfOf, paybackAmount);
  } else {
    IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
      onBehalfOf,
      paybackAmount,
      reserve.variableBorrowIndex
    );
  }

  address aToken = reserve.aTokenAddress;
  reserve.updateInterestRates(asset, aToken, paybackAmount, 0);

  if (stableDebt.add(variableDebt).sub(paybackAmount) == 0) {
    _usersConfig[onBehalfOf].setBorrowing(reserve.id, false);
  }
```

```
  IERC20(asset).safeTransferFrom(msg.sender, aToken, paybackAmount);

  emit Repay(asset, onBehalfOf, msg.sender, paybackAmount);
}
```

Listing 3.6: LendingPool.sol

Status The issue has been confirmed and accordingly fixed by this merge request: 82.

3.5 Validation of transferFrom() Return Values

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LendingPool
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

Another core functionality implemented in the LendingPool contract is the new flashloan feature. This feature allows the creation of a variety of tools for refinance, collateral swap, arbitrage and liquidations. It further addresses the limitation of earlier versions by allowing the flashloans to be used within the Aave protocol (by carefully taking care of potential reentrancy concerns and other limitations).

To elaborate, we show below the code snippet of flashLoan(). This routine works in two different modes: The first mode simply maintains an invariant in guaranteeing the final balance of lending pool is larger than the earlier balance plus the premium charged for this flashloan; The second mode allows the flashloan to be borrowed from the pool with the assumption of this borrow is backed up with earlier collateral.

```
function flashLoan(
  address receiverAddress,
  address asset,
  uint256 amount,
  uint256 mode,
  bytes calldata params,
  uint16 referralCode
) external override {
  _whenNotPaused();
  ReserveLogic.ReserveData storage reserve = _reserves[asset];
  FlashLoanLocalVars memory vars;

  vars.aTokenAddress = reserve.aTokenAddress;
```

```
    vars.premium = amount.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000);

    ValidationLogic.validateFlashloan(mode, vars.premium);

    ReserveLogic.InterestRateMode debtMode = ReserveLogic.InterestRateMode(mode);

    vars.receiver = IFlashLoanReceiver(receiverAddress);

    //transfer funds to the receiver
    IAToken(vars.aTokenAddress).transferUnderlyingTo(receiverAddress, amount);

    //execute action of the receiver
    vars.receiver.executeOperation(asset, amount, vars.premium, params);

    vars.amountPlusPremium = amount.add(vars.premium);

    if (debtMode == ReserveLogic.InterestRateMode.NONE) {
      IERC20(asset).transferFrom(receiverAddress, vars.aTokenAddress, vars.
          amountPlusPremium);

      reserve.updateState();
      reserve.cumulateToLiquidityIndex(IERC20(vars.aTokenAddress).totalSupply(), vars.
          premium);
      reserve.updateInterestRates(asset, vars.aTokenAddress, vars.premium, 0);

      emit FlashLoan(receiverAddress, asset, amount, vars.premium, referralCode);
    } else {
      // If the transfer didn't succeed, the receiver either didn't return the funds, or
           didn't approve the transfer.
      _executeBorrow(
        ExecuteBorrowParams(
          asset,
          msg.sender,
          msg.sender,
          vars.amountPlusPremium,
          mode,
          vars.aTokenAddress,
          referralCode,
          false
        )
      );
    }
  }
```

Listing 3.7: LendingPool.sol

While reviewing the first mode, we notice that when the flashloan is transferred to the borrower, it is properly handled with `safeTransfer()` (line 245 in `AToken` contract) that safely validates the return value. However, when the flashloan is being returned back the pool with the necessary premium, it is handled with `transferFrom()` that does not validate the return value. To better accommodate various idiosyncrasies associated with different implementations/customizations of ERC20 tokens, we strongly suggest to replace all occurrences of transferFrom() with the safe version of safeTransferFrom() from OpenZeppelin. The issue is also applicable to other two unsafe transfers in LendingPoolCollateralManager.

Recommendation Replace all occurrences of transferFrom() with the safe version of safeTransferFrom () from OpenZeppelin. Similarly, replace unsafe transfer(), if any, with safeTransfer() as well.

Status This issue has been confirmed and accordingly fixed by replacing transferFrom() with safeTransferFrom() in the merge request: 56.

3.6 Improved Precision By Multiplication-Before-Division

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: DefaultReserveInterestRateStrategy
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [4]

Description

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. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the calculateInterestRates() (in DefaultReserveInterestRateStrategy contract) as an example. This routine is used to calculate the interest rate due to changes related to the lending pool, either from new borrows or deposits.

```
function calculateInterestRates(
  address reserve,
  uint256 availableLiquidity,
  uint256 totalStableDebt,
  uint256 totalVariableDebt,
  uint256 averageStableBorrowRate,
  uint256 reserveFactor
)
  external
  override
  view
  returns (
```

```
      uint256,
      uint256,
      uint256
    )
  {

    CalcInterestRatesLocalVars memory vars;

    vars.totalBorrows = totalStableDebt.add(totalVariableDebt);
    vars.currentVariableBorrowRate = 0;
    vars.currentStableBorrowRate = 0;
    vars.currentLiquidityRate = 0;

    uint256 utilizationRate = vars.totalBorrows == 0
      ? 0
      : vars.totalBorrows.rayDiv(availableLiquidity.add(vars.totalBorrows));

    vars.currentStableBorrowRate = ILendingRateOracle(addressesProvider.getLendingRateOracle())
      .getMarketBorrowRate(reserve);

    if (utilizationRate > OPTIMAL_UTILIZATION_RATE) {
      uint256 excessUtilizationRateRatio = utilizationRate.sub(OPTIMAL_UTILIZATION_RATE).rayDiv(
        EXCESS_UTILIZATION_RATE
      );

      vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(_stableRateSlope1).add(
        _stableRateSlope2.rayMul(excessUtilizationRateRatio)
      );

      vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(_variableRateSlope1).add(
        _variableRateSlope2.rayMul(excessUtilizationRateRatio)
      );
    } else {
      vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(
        _stableRateSlope1.rayMul(utilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE))
      );
      vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(
        utilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE).rayMul(_variableRateSlope1)
      );
    }

    vars.currentLiquidityRate = _getOverallBorrowRate(
      totalStableDebt,
      totalVariableDebt,
      vars.currentVariableBorrowRate,
      averageStableBorrowRate
    )
      .rayMul(utilizationRate)
```

```
179             .percentMul(PercentageMath.PERCENTAGE_FACTOR.sub(reserveFactor));
181         return (vars.currentLiquidityRate, vars.currentStableBorrowRate, vars.
                currentVariableBorrowRate);
182     }
```

Listing 3.8: DefaultReserveInterestRateStrategy.sol

We notice the calculation of the `currentVariableBorrowRate` (lines $167 - 169$) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `baseVariableBorrowRate.add(utilizationRate.rayMul(_variableRateSlope1).rayDiv(OPTIMAL_UTILIZATION_RATE))`. Similarly, the calculation of `calculateAvailableCollateralToLiquidate()` in `LendingPoolCollateralManager` contract (line 584) can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been confirmed and accordingly fixed by these two merge requests: `82` and `88`.

3.7 Improved STABLE_BORROWING_MASK

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ReserveConfiguration`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

For gas efficiency and improved scalability, Aave V2 introduces a bitmask to store the reserve configuration. The bitmask is defined as follows:

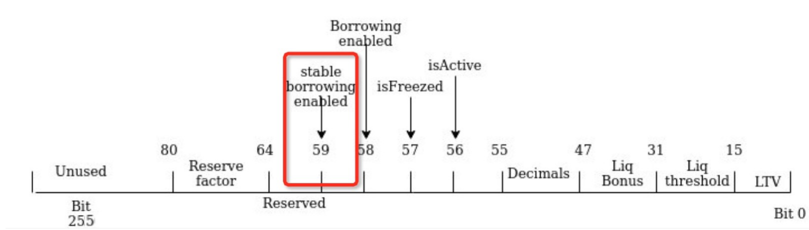

Figure 3.1: The Reserve Configuration Bitmask in Aave V2

Specifically, the bitmask has a 256 bit size and is divided into 11 segments: `LTV`, `Liquidation Threshold`, `Liquidation Bonus`, `Decimal`, `isActive`, `isFreezed`, `Bororowing Enabled`, `Stable Bororowing Enabled`, `Reserved`, `Reserve Factor`, and `Unused`. The above segments occupy 16 bits, 16 bits, 16 bits, 8 bits, 1 bit, 1 bit, 1 bit, 1 bit, 5 bits, 16 bits, and 175 bits, respectively.

```
library ReserveConfiguration {
  uint256 constant LTV_MASK = 0xFFFFFFFFFFFFFFFF0000;
  uint256 constant LIQUIDATION_THRESHOLD_MASK = 0xFFFFFFFFFFFF0000FFFF;
  uint256 constant LIQUIDATION_BONUS_MASK = 0xFFFFFFF0000FFFFFFFF;
  uint256 constant DECIMALS_MASK = 0xFFFFFF00FFFFFFFFFFFF;
  uint256 constant ACTIVE_MASK = 0xFFFFFEFFFFFFFFFFFFFF;
  uint256 constant FROZEN_MASK = 0xFFFFFDFFFFFFFFFFFFFF;
  uint256 constant BORROWING_MASK = 0xFFFFFBFFFFFFFFFFFFFF;
  uint256 constant STABLE_BORROWING_MASK = 0xFFFF07FFFFFFFFFFFFFF;
  uint256 constant RESERVE_FACTOR_MASK = 0xFFFFFFFFFFFFFFFF;
  ...
}
```

Listing 3.9: ReserveConfiguration.sol

We have examined the bitmask for each specific segment and notice that the `STABLE_BORROWING_MASK = 0xFFFF07FFFFFFFFFFFFFF` takes 5 bits, instead of 1. With an incorrect mask for `Stable Borrowing Enabled`, it unnecessarily affects the adjacent 4 bits (even though these 4 bits are currently reserved)[1] via the affiliated `getStableRateBorrowingEnabled()`/`setStableRateBorrowingEnabled()` routines.

```
  /**
   * @dev enables or disables stable rate borrowing on the reserve
   * @param self the reserve configuration
   * @param enabled true if the stable rate borrowing needs to be enabled, false
      otherwise
   **/
  function setStableRateBorrowingEnabled(ReserveConfiguration.Map memory self, bool
      enabled)
    internal pure
  {
    self.data = (self.data & STABLE_BORROWING_MASK) | (uint256(enabled ? 1 : 0) << 59);
  }

  /**
   * @dev gets the stable rate borrowing state of the reserve
   * @param self the reserve configuration
   * @return the stable rate borrowing state
   **/
  function getStableRateBorrowingEnabled(ReserveConfiguration.Map storage self)
    internal
    view
    returns (bool)
  {
```

[1]A follow-up discussion with the team further shows that the `LIQUIDATION_BONUS_MASK` misses an F in its mask. In other words, it should be $0xFFFFFFFF0000FFFFFFFF$ instead of $0xFFFFFFF0000FFFFFFFF$.

```
    return ((self.data & ~STABLE_BORROWING_MASK) >> 59) != 0;
}
```

Listing 3.10: ReserveConfiguration.sol

Recommendation Match the mask of each segment with the number of occupied bits (by the segment).

Status This issue has been confirmed and accordingly fixed by correcting the wrong mask in the merge request: 63.

3.8 Inaccurate Burn Events in AToken

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AToken
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the AToken contract as an example. This contract is designed to tokenize the assets deposited into the lending pool. The tokenized AToken can therefore be minted, transferred, or burned. While examining the events that reflect the AToken dynamics, we notice the emitted Burn event (line 111) contains incorrect information. Specifically, the event is defined as event Burn(address indexed from, address indexed target, uint256 value, uint256 index) with a number of parameters: the first parameter from encodes the address that performs the redeem/burn operation; the second parameter target shows the amount to be redeemed, while the last parameter index indicates the last index of the reserve when the redeem happens. The emitted event contains an incorrect from information, which should not be msg.sender. Instead, from here should be user, the first function argument to burn().

```
93  /**
94   * @dev burns the aTokens and sends the equivalent amount of underlying to the target.
95   * only lending pools can call this function
96   * @param amount the amount being burned
```

```
    **/
  function burn(
    address user,
    address receiverOfUnderlying,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool {
    _burn(user, amount.rayDiv(index));

    //transfers the underlying to the target
    IERC20(UNDERLYING_ASSET_ADDRESS).safeTransfer(receiverOfUnderlying, amount);

    //transfer event to track balances
    emit Transfer(user, address(0), amount);
    emit Burn(msg.sender, receiverOfUnderlying, amount, index);
  }
```

Listing 3.11: AToken.sol

Recommendation Properly emit the `Burn` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been confirmed and accordingly fixed by this merge request: `107`.

3.9 Asset Consistency Between Reserve and AToken

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ReserveLogic`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.2, `LendingPool` is a core pool contract in Aave V2. At the heart of `LendingPool` is the concept of reserve: every pool holds a reserve that is specific to the supported crypto-currency, with the total amount in Ethereum defined as total liquidity. A reserve accepts deposits from lenders with assets actually stored in the asset-specific `AToken`. Users can borrow these funds, granted that they lock a greater value as collateral, which backs the borrow position.

Evidently, there is a one-to-one mapping between the supported `asset` and its `AToken`. There also exists one-to-one mapping between the supported `asset` and the respective `reserve`. As a result, the mapping between a `reserve` and `AToken` should also be one-to-one. Accordingly, it is helpful to enforce the consistency so that the `reserve` is initialized with the corresponding `AToken` that shares the same underlying asset.

To elaborate, we show below the initialization routine (`initReserve()`) of a new reserve. The initialization routine takes five parameters, i.e., `asset`, `aTokenAddress`, `stableDebtAddress`, `variableDebtAddress`, and `interestRateStrategyAddress`. Naturally, the first parameter `asset` needs to be consistent with the underlying asset behind the second parameter `aTokenAddress`. Note that `aTokenAddress` has an internal immutable member `UNDERLYING_ASSET_ADDRESS`. Therefore, we can enforce the consistency between `asset` and `UNDERLYING_ASSET_ADDRESS`.

```
/**
 * @dev initializes a reserve
 * @param asset the address of the reserve
 * @param aTokenAddress the address of the overlying aToken contract
 * @param interestRateStrategyAddress the address of the interest rate strategy
    contract
 **/
function initReserve(
  address asset,
  address aTokenAddress,
  address stableDebtAddress,
  address variableDebtAddress,
  address interestRateStrategyAddress
) external override {
  _onlyLendingPoolConfigurator();
  _reserves[asset].init(
    aTokenAddress,
    stableDebtAddress,
    variableDebtAddress,
    interestRateStrategyAddress
  );
  _addReserveToList(asset);
}
```

Listing 3.12: LendingPool.sol

Recommendation Enforce the asset consistency between the `reserve` and the asset-corresponding `AToken`.

Status The issue has been confirmed and accordingly fixed by this merge request: `82`. The fix involves the `LendingPoolConfigurator`, where the `initReserve()` function now fetches the asset from the `AToken` and enforces the correctness of the underlying asset and the lending pool address across `AToken`, variable debt token and stable debt token. The fix involved a small change to DebtTokenBase to have a common interface for the `POOL` and `UNDERLYING_ASSET_ADDRESS` between `ATokens` and `DebtTokens`.

3.10 Inaccurate Calculation of Mints To Treasury

- ID: PVE-010
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: AToken
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [4]

Description

As mentioned in Section 3.6, 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. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one issue related to precision loss.

Specifically, Aave V2 implements the WadRayMath library that provides mul and div functions for wads (decimal numbers with 18 digits precision) and rays (decimals with 27 digits). If an arithmetic calculation operates on rays, not wads, the higher precision is helpful to reduce potential precision loss. As an example, we show below the code snippet of rayDiv() that divides two ray numbers, with the result rounding half up to the nearest ray.

```
/**
 * @dev divides two ray, rounding half up to the nearest ray
 * @param a ray
 * @param b ray
 * @return the result of a/b, in ray
 **/
function rayDiv(uint256 a, uint256 b) internal pure returns (uint256) {
  require(b != 0, Errors.DIVISION_BY_ZERO);

  uint256 halfB = b / 2;

  uint256 result = a * RAY;

  require(result / RAY == a, Errors.MULTIPLICATION_OVERFLOW);

  result += halfB;

  require(result >= halfB, Errors.ADDITION_OVERFLOW);

  return result / b;
}
```

Listing 3.13: WadRayMath.sol

For reduced precision loss, Aave V2 takes the convention in using `rays` for the calculation of interest rates and indexes, and using `wads` for token balances and amounts. We have accordingly examined the enforcement of this convention and notice that one routine, i.e., `mintToTreasury()`, violates this convention.

To elaborate, we show the `mintToTreasury()` routine below. To accommodate the ever-changing indexes in the lending pool, `AToken` internally keeps the scaled number. Therefore, the minted amount to the treasury should be calculated as `amount.rayDiv(index)`, not `amount.div(index)`. In other words, the current implementation (line 134) violates this convention and yields the wrong amount minted to the treasury. As the decimal difference between `wads` and `rays` is 9, the currently minted amount to the treasury becomes dramatically smaller with only $1/(10 ** 9)$ of the intended amount.

```
133   function mintToTreasury(uint256 amount, uint256 index) external override
          onlyLendingPool {
134     _mint(RESERVE_TREASURY_ADDRESS, amount.div(index));

136     //transfer event to track balances
137     emit Transfer(address(0), RESERVE_TREASURY_ADDRESS, amount);
138     emit Mint(RESERVE_TREASURY_ADDRESS, amount, index);
139   }
```

Listing 3.14: AToken.sol

Recommendation Replace the `amount.div(index)` division in `mintToTreasury()` with `amount.rayDiv(index)`.

```
133   function mintToTreasury(uint256 amount, uint256 index) external override
          onlyLendingPool {
134     _mint(RESERVE_TREASURY_ADDRESS, amount.rayDiv(index));

136     //transfer event to track balances
137     emit Transfer(address(0), RESERVE_TREASURY_ADDRESS, amount);
138     emit Mint(RESERVE_TREASURY_ADDRESS, amount, index);
139   }
```

Listing 3.15: AToken.sol

Status The issue has been confirmed and accordingly fixed by this merge request: `65`.

3.11 Premature Updates of updateInterestRates() Before DebtToken Changes

- ID: PVE-011
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `LendingPoolCollateralManager`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

The `LendingPool` contract provides a number of core routines for borrowing/lending users to interact with, including `deposit()`, `withdraw()`, `borrow()`, `repay()`, `flashloan()`, and etc. To facilitate the execution of each core routine, Aave V2 validates the given arguments to these core routines with corresponding validation routines in `ValidationLogic`, such as `validateDeposit()`, `validateWithdraw()`, `validateBorrow()`, `validateRepay()`, `validateFlashloan()`, and etc.

More importantly, all the actions performed in each core routine follow a specific sequence:

- Step I: It firstly validates the given arguments as well as current state. If current state cannot meet the pre-conditions required for the intended action, the transaction will be reverted.
- Step II: It then updates reserve state to reflect the latest borrow/liquidity indexes (up to the current block height) and further calculates the new amount that will be minted to the treasury. The updated indexes are necessary to get the reserve ready for the execution of the intended action.
- Step III: It next "executes" the intended action that may need to update the user accounting and reserve balance as the action could involve transferring assets into or out of the reserve. The updates could lead to minting or burning of tokens that are related to lending/borrowing positions of current user. The tokens are represented as `ATokens`, `StableDebtTokens`, or `VariableDebtTokens`.
- Step IV: Due to possible changes to the reserve from the action, such as resulting in a different utilization rate from either borrowing or lending, it also needs to accordingly adjust the interest rates to accurately accrue interests.
- Step V: By following the known best practice of the `checks-effects-interactions` pattern, it finally performs the external interactions, if any.

One of the advanced features implemented in Aave V2 is the tokenization of both lending and borrowing positions. When a user deposits assets into a specific reserve, the user receives the corresponding amount of `ATokens` to represent the liquidity deposited and accrue the interests. When a user opens or increases a borrow position, the user receives the corresponding amount of `DebtTokens` (either `StableDebtTokens` or `VariableDebtTokens` depending on the borrow mode) to represent the debt position and further accrue the debt interests.

The above order sequence needs to be properly maintained. Our analysis shows that in several routines, the `updateInterestRates()` (Step IV) is executed prematurally before the updates to the internal accounting data associated with users (Step III).

To elaborate, we show below the code snippet from `liquidationCall()` that handles the liquidation request for a default user. Specifically, `updateInterestRates()` (line 227) is performed before user debt updates (lines 234 – 251). This out-of-order execution could lead to higher interest rates being calculated and accrued at the cost of borrowing users!

```
224     //update the principal reserve
225     principalReserve.updateState();

227     principalReserve.updateInterestRates(
228       principal,
229       principalReserve.aTokenAddress,
230       vars.actualAmountToLiquidate,
231       0
232     );

234     if (vars.userVariableDebt >= vars.actualAmountToLiquidate) {
235       IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
236         user,
237         vars.actualAmountToLiquidate,
238         principalReserve.variableBorrowIndex
239       );
240     } else {
241       IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
242         user,
243         vars.userVariableDebt,
244         principalReserve.variableBorrowIndex
245       );

247       IStableDebtToken(principalReserve.stableDebtTokenAddress).burn(
248         user,
249         vars.actualAmountToLiquidate.sub(vars.userVariableDebt)
250       );
251     }
```

Listing 3.16: LendingPoolCollateralManager.sol

Recommendation Maintain the right order between `updateInterestRates()` and debt token changes. An example revision to the above code snippet is shown below.

```
    //update the principal reserve
    principalReserve.updateState();

    if (vars.userVariableDebt >= vars.actualAmountToLiquidate) {
      IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
        user,
        vars.actualAmountToLiquidate,
        principalReserve.variableBorrowIndex
      );
    } else {
      IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
        user,
        vars.userVariableDebt,
        principalReserve.variableBorrowIndex
      );

      IStableDebtToken(principalReserve.stableDebtTokenAddress).burn(
        user,
        vars.actualAmountToLiquidate.sub(vars.userVariableDebt)
      );
    }

    principalReserve.updateInterestRates(
      principal,
      principalReserve.aTokenAddress,
      vars.actualAmountToLiquidate,
      0
    );
```

Listing 3.17: LendingPoolCollateralManager.sol

Status The issue has been confirmed and accordingly fixed by this merge request: `88`.

3.12 Late Updates of updateInterestRates() After AToken Changes

- ID: PVE-012
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `LendingPoolCollateralManager`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

As mentioned in Section 3.11, the `LendingPool` contract provides a number of core routines for borrowing/lending users to access its functionalities. In the same section, we also elaborate an issue that is introduced due to the premature updates of system-wide interest rates before the debt positions have been finalized. In this section, we further analyze the interest rate calculation and report another issue that is caused from the out-of-order execution between the interest rate update and the AToken changes.

Specifically, when there is any transfer into or out of the reserve from an user, the user's lending position, represented by the holding amount of AToken, will be properly updated. In the meantime, we need to properly maintain the execution order in order to accurately calculate the interest rates of updated reserves.

To elaborate, we show below the code snippet of the swapLiquidity() routine.

```
function swapLiquidity(
  address receiverAddress,
  address fromAsset,
  address toAsset,
  uint256 amountToSwap,
  bytes calldata params
) external returns (uint256, string memory) {
  ReserveLogic.ReserveData storage fromReserve = _reserves[fromAsset];
  ReserveLogic.ReserveData storage toReserve = _reserves[toAsset];

  SwapLiquidityLocalVars memory vars;

  (vars.errorCode, vars.errorMsg) = ValidationLogic.validateSwapLiquidity(
    fromReserve,
    toReserve,
    fromAsset,
    toAsset
  );

  if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
      .NO_ERROR) {
    return (vars.errorCode, vars.errorMsg);
  }

  vars.fromReserveAToken = IAToken(fromReserve.aTokenAddress);
  vars.toReserveAToken = IAToken(toReserve.aTokenAddress);

  fromReserve.updateState();
  toReserve.updateState();

  if (vars.fromReserveAToken.balanceOf(msg.sender) == amountToSwap) {
    _usersConfig[msg.sender].setUsingAsCollateral(fromReserve.id, false);
  }

  fromReserve.updateInterestRates(fromAsset, address(vars.fromReserveAToken), 0,
      amountToSwap);

  vars.fromReserveAToken.burn(
    msg.sender,
```

```
492           receiverAddress,
493           amountToSwap,
494           fromReserve.liquidityIndex
495         );
496         // Notifies the receiver to proceed, sending as param the underlying already
                transferred
497         ISwapAdapter(receiverAddress).executeOperation(
498           fromAsset,
499           toAsset,
500           amountToSwap,
501           address(this),
502           params
503         );

505         vars.amountToReceive = IERC20(toAsset).balanceOf(receiverAddress);
506         if (vars.amountToReceive != 0) {
507           IERC20(toAsset).transferFrom(
508             receiverAddress,
509             address(vars.toReserveAToken),
510             vars.amountToReceive
511           );

513           if (vars.toReserveAToken.balanceOf(msg.sender) == 0) {
514             _usersConfig[msg.sender].setUsingAsCollateral(toReserve.id, true);
515           }

517           vars.toReserveAToken.mint(msg.sender, vars.amountToReceive, toReserve.
                  liquidityIndex);
518           toReserve.updateInterestRates(
519             toAsset,
520             address(vars.toReserveAToken),
521             vars.amountToReceive,
522             0
523           );
524         }

526         ...
527     }
```

Listing 3.18: LendingPoolCollateralManager.sol

If we pay attention to `toReserve.updateInterestRates()` (lines 518 – 523), the interest rates have been inappropriately updated after `toAsset` has been transferred into the respective reserve and the related `ATokens` have been minted. In other words, the `amountToReceive` number has been taken into account twice, leading to the wrong calculation of having a doubled transfer-in amount: `2 * amountToReceive`. As a result, this calculation makes the reserve in having a lower utilization rate at the cost of lending users with less accrued interests!

Recommendation Maintain the right order between `updateInterestRates()` and `AToken` changes. An example revision to the above code snippet is shown below.

```
  function swapLiquidity(
    address receiverAddress,
    address fromAsset,
    address toAsset,
    uint256 amountToSwap,
    bytes calldata params
  ) external returns (uint256, string memory) {
    ReserveLogic.ReserveData storage fromReserve = _reserves[fromAsset];
    ReserveLogic.ReserveData storage toReserve = _reserves[toAsset];

    SwapLiquidityLocalVars memory vars;

    (vars.errorCode, vars.errorMsg) = ValidationLogic.validateSwapLiquidity(
      fromReserve,
      toReserve,
      fromAsset,
      toAsset
    );

    if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
        .NO_ERROR) {
      return (vars.errorCode, vars.errorMsg);
    }

    vars.fromReserveAToken = IAToken(fromReserve.aTokenAddress);
    vars.toReserveAToken = IAToken(toReserve.aTokenAddress);

    fromReserve.updateState();
    toReserve.updateState();

    if (vars.fromReserveAToken.balanceOf(msg.sender) == amountToSwap) {
      _usersConfig[msg.sender].setUsingAsCollateral(fromReserve.id, false);
    }

    fromReserve.updateInterestRates(fromAsset, address(vars.fromReserveAToken), 0,
        amountToSwap);

    vars.fromReserveAToken.burn(
      msg.sender,
      receiverAddress,
      amountToSwap,
      fromReserve.liquidityIndex
    );
    // Notifies the receiver to proceed, sending as param the underlying already
        transferred
    ISwapAdapter(receiverAddress).executeOperation(
      fromAsset,
      toAsset,
      amountToSwap,
      address(this),
      params
    );
```

```
    vars.amountToReceive = IERC20(toAsset).balanceOf(receiverAddress);
    if (vars.amountToReceive != 0) {
      toReserve.updateInterestRates(
        toAsset,
        address(vars.toReserveAToken),
        vars.amountToReceive,
        0
      );
      IERC20(toAsset).transferFrom(
        receiverAddress,
        address(vars.toReserveAToken),
        vars.amountToReceive
      );

      if (vars.toReserveAToken.balanceOf(msg.sender) == 0) {
        _usersConfig[msg.sender].setUsingAsCollateral(toReserve.id, true);
      }

      vars.toReserveAToken.mint(msg.sender, vars.amountToReceive, toReserve.
          liquidityIndex);
    }

    ...
  }
```

Listing 3.19: LendingPoolCollateralManager.sol

Status The issue has been confirmed and accordingly fixed by this merge request: `87`. And the `transferFrom()` of the currency has been relocated to after the calculation of the interest rates.

3.13 Inconsistency Between Document and Implementation

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in line 359 of `LendingPool::rebalanceStableBorrowRate()`, line 211 of `StableDebtToken::_calculateBalanceIncrease()`, and line 301 of `LendingPoolCollateralManager::repayWithCollateral()`, Using the `rebalanceStableBorrowRate()` routine as an example, the preceding function summary indicates that the stable interest rate of a user will not be rebalanced if current liquidity rate is larger than the user stable rate. However, the enforcement (lines $394-399$) indicates two other specific requirements in `usageRatio` and `currentLiquidityRate`, but not related to the user stable rate.

```
358  /**
359   * @dev rebalances the stable interest rate of a user if current liquidity rate > user
         stable rate.
360   * this is regulated by Aave to ensure that the protocol is not abused, and the user
        is paying a fair
361   * rate. Anyone can call this function.
362   * @param asset the address of the reserve
363   * @param user the address of the user to be rebalanced
364   **/
365  function rebalanceStableBorrowRate(address asset, address user) external override {
366
367    _whenNotPaused();
368
369    ReserveLogic.ReserveData storage reserve = _reserves[asset];
370
371    IERC20 stableDebtToken = IERC20(reserve.stableDebtTokenAddress);
372    IERC20 variableDebtToken = IERC20(reserve.variableDebtTokenAddress);
373    address aTokenAddress = reserve.aTokenAddress;
374
375    uint256 stableBorrowBalance = IERC20(stableDebtToken).balanceOf(user);
376
377    //if the utilization rate is below 95%, no rebalances are needed
378    uint256 totalBorrows = stableDebtToken.totalSupply().add(variableDebtToken.
         totalSupply()).wadToRay();
379    uint256 availableLiquidity = IERC20(asset).balanceOf(aTokenAddress).wadToRay();
380    uint256 usageRatio = totalBorrows == 0
381      ? 0
382      : totalBorrows.rayDiv(availableLiquidity.add(totalBorrows));
383
384    //if the liquidity rate is below REBALANCE_UP_THRESHOLD of the max variable APR at
         95% usage,
385    //then we allow rebalancing of the stable rate positions.
386
387    uint256 currentLiquidityRate = reserve.currentLiquidityRate;
388    uint256 maxVariableBorrowRate = IReserveInterestRateStrategy(
389      reserve
390        .interestRateStrategyAddress
391    )
392      .getMaxVariableBorrowRate();
393
394    require(
395      usageRatio >= REBALANCE_UP_USAGE_RATIO_THRESHOLD &&
396      currentLiquidityRate <=
397        maxVariableBorrowRate.percentMul(REBALANCE_UP_LIQUIDITY_RATE_THRESHOLD),
398      Errors.INTEREST_RATE_REBALANCE_CONDITIONS_NOT_MET
399    );
400
```

```
    reserve.updateState();

    IStableDebtToken(address(stableDebtToken)).burn(user, stableBorrowBalance);
    IStableDebtToken(address(stableDebtToken)).mint(user, stableBorrowBalance, reserve.
        currentStableBorrowRate);

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

    emit RebalanceStableBorrowRate(asset, user);

  }
```

Listing 3.20: LendingPool.sol

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been confirmed and accordingly fixed by this merge request: 87.

3.14 Removal of Unused Code

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: GenericLogic
- Category: Coding Practices [10]
- CWE subcategory: CWE-563 [6]

Description

Aave V2 makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, VersionedInitializable, and Ownable, to facilitate its code implementation and organization. For example, the LendingPool 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 GenericLogic contract, there is a defined constant that is not used anymore: HEALTH_FACTOR_CRITICAL_THRESHOLD, This constant is apparently left behind from a deprecated feature.

```
library GenericLogic {
  using ReserveLogic for ReserveLogic.ReserveData;
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using ReserveConfiguration for ReserveConfiguration.Map;
  using UserConfiguration for UserConfiguration.Map;
```

```
  uint256 public constant HEALTH_FACTOR_LIQUIDATION_THRESHOLD = 1 ether;
  uint256 public constant HEALTH_FACTOR_CRITICAL_THRESHOLD = 0.98 ether;
  ...
}
```

Listing 3.21: GenericLogic.sol

In addition, there are a number of unused constant variables defined in `LendingPoolAddressesProvider` and these unused constants can be removed as well. Examples include `WALLET_BALANCE_PROVIDER`, `LENDING_POOL_CORE`, `LENDING_POOL_FLASHLOAN_PROVIDER`, and `DATA_PROVIDER`.

Recommendation Consider the removal of the unused code and the unused constants.

Status The issue has been confirmed and accordingly fixed by this merge request: `87`.

3.15 Possible Fund Loss From (Permissive) Smart Wallets With Allowances to LendingPool

- ID: PVE-015
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `LendingPoolCollateralManager`, `LendingPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

Among all core functionalities provided in `LendingPool`, `flashloan` is a disruptive one that allows users to borrow from the reserves within a single transaction, as long as the user returns the borrowed amount plus additional premium. In this section, we report an issue related to the `flashloan` feature. The `flashloan` feature improves earlier versions by allowing the borrowed flashloans to be used in Aave V2 as well (with proper workarounds against potential `reentrancy` risks). Moreover, it seamlessly integrates the borrow functionality to avoid returning back the flashloan within the same transaction.

To elaborate, we show below the code snippet of `flashLoan()` behind the feature.

```
  function flashLoan(
    address receiverAddress,
    address asset,
    uint256 amount,
    uint256 mode,
    bytes calldata params,
    uint16 referralCode
  ) external override {
    _whenNotPaused();
    ReserveLogic.ReserveData storage reserve = _reserves[asset];
```

```
    FlashLoanLocalVars memory vars;

    vars.aTokenAddress = reserve.aTokenAddress;

    vars.premium = amount.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000);

    ValidationLogic.validateFlashloan(mode, vars.premium);

    ReserveLogic.InterestRateMode debtMode = ReserveLogic.InterestRateMode(mode);

    vars.receiver = IFlashLoanReceiver(receiverAddress);

    //transfer funds to the receiver
    IAToken(vars.aTokenAddress).transferUnderlyingTo(receiverAddress, amount);

    //execute action of the receiver
    vars.receiver.executeOperation(asset, amount, vars.premium, params);

    vars.amountPlusPremium = amount.add(vars.premium);

    if (debtMode == ReserveLogic.InterestRateMode.NONE) {
      IERC20(asset).transferFrom(receiverAddress, vars.aTokenAddress, vars.
          amountPlusPremium);

      reserve.updateState();
      reserve.cumulateToLiquidityIndex(IERC20(vars.aTokenAddress).totalSupply(), vars.
          premium);
      reserve.updateInterestRates(asset, vars.aTokenAddress, vars.premium, 0);

      emit FlashLoan(receiverAddress, asset, amount, vars.premium, referralCode);
    } else {
      // If the transfer didn't succeed, the receiver either didn't return the funds, or
           didn't approve the transfer.
      _executeBorrow(
        ExecuteBorrowParams(
          asset,
          msg.sender,
          msg.sender,
          vars.amountPlusPremium,
          mode,
          vars.aTokenAddress,
          referralCode,
          false
        )
      );
    }
  }
```

Listing 3.22: LendingPool.sol

This particular routine implements the `flashloan` feature in a straightforward manner: It firstly transfers the funds to the specified receiver, then invokes the designated operation (`executeOperation`

- line 573), next transfers back the funds from the receiver or creates an equivalent borrow.

However, our analysis shows that the above logic may be abused to cause fund loss of an innocent user if the user previously specified certain allowances to `LendingPool`. Specifically, if a flashloan is launched by specifying the innocent user an the `receiverAddress` argument, the `flashLoan()`) execution follows the logic by firstly transferring the loan amount to `receiverAddress`, invoking `executeOperation()` on the receiver, and then transferring the `amountPlusPremium` (no larger than the allowed spending amount) from the receiver back to the pool. Note that this flashloan is not initiated by the `receiverAddress`, who unfortunately pays the premium associated with the flashloan.

The same issue is also applicable to two other routines, i.e., `swapLiquidity()` and `repayWithCollateral()`. Note the exploitation can be used to directly steal the funds of innocent users for the attacker's benefits. In the meantime, we need to mention that the `executeOperation()` call will be invoked on the given `receiverAddress`. The compiler will place a sanity check in ensuring the `receiverAddress` is indeed a contract, hence restricting the attack vector only applicable to contract-based smart wallets. However, current smart wallets may have a fallback routine that could allow the `executeOperation()` call to proceed without being reverted.[2]

Recommendation Revisit the design of affected routines in possibly avoiding initiating the `transferFrom()` call from the lending pool. Moreover, the revisited design may validate the `executeOperation()` call so that it is required to successfully transfer back the expected assets, if any.

Status The issue has been confirmed and accordingly fixed by this merge request: `86`. Note that due to this issue, both `swapLiquidity()` and `repayWithCollateral` functions have been removed.

3.16 Improved Business Logic in validateWithdraw()

- ID: PVE-016
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ValidationLogic`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

Aave V2 centralizes the validation logic in `ValidationLogic` contract to streamline the process of a variety of core functionalities in `LendingPool`. Specifically, to facilitate the execution of each core routine (e.g., `deposit()`, `withdraw()`, `borrow()`, `repay()`, and `flashloan()`), corresponding validation routines have been provided, including `ValidationLogic`, `validateDeposit()`, `validateWithdraw()`, `validateBorrow()`, `validateRepay()`, `validateFlashloan()`.

[2] An example is those smart wallets in `InstaDApp()`, a popular portal that simplifies the needs for DeFi users.

While analyzing the validation logic of `validateWithdraw()`, we notice an issue that validates the borrow amount in the current balance, instead of the borrow amount.

```
/**
 * @dev validates a withdraw action.
 * @param reserveAddress the address of the reserve
 * @param amount the amount to be withdrawn
 * @param userBalance the balance of the user
 */
function validateWithdraw(
  address reserveAddress,
  uint256 amount,
  uint256 userBalance,
  mapping(address => ReserveLogic.ReserveData) storage reservesData,
  UserConfiguration.Map storage userConfig,
  address[] calldata reserves,
  address oracle
) external view {
  require(amount > 0, Errors.AMOUNT_NOT_GREATER_THAN_0);

  require(amount <= userBalance, Errors.NOT_ENOUGH_AVAILABLE_USER_BALANCE);

  require(
    GenericLogic.balanceDecreaseAllowed(
      reserveAddress,
      msg.sender,
      userBalance,
      reservesData,
      userConfig,
      reserves,
      oracle
    ),
    Errors.TRANSFER_NOT_ALLOWED
  );
}
```

Listing 3.23: ValidationLogic.sol

To elaborate, we show above the code snippet of `validateWithdraw()`. This routine ensures the borrow amount falls in an appropriate range, i.e., $(0, userBalance]$, and then delegates the validation to `GenericLogic.balanceDecreaseAllowed()`. However, the delegated call is forwarded with `userBalance`, not the actual borrow `amount` to validate whether this specific borrow amount is allowed.

Recommendation Revise the `validateWithdraw()` logic to properly validate using the actual borrow amount, not current balance.

Status The issue has been confirmed and accordingly fixed by this merge request: `69`.

3.17 Improved Event Generation With Indexed Assets

- ID: PVE-017
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LendingPoolConfigurator
- Category: Time and State [9]
- CWE subcategory: CWE-362 [5]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. As mentioned in Section 3.8, events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

We have examined the support of system-wide parameters in Aave V2 and notice that configuration-related getter/setter routines are mainly implemented in LendingPoolConfigurator. In the following, we list a few representative events that have been defined in Aave V2.

```
/**
 * @dev emitted when borrowing is enabled on a reserve
 * @param asset the address of the reserve
 * @param stableRateEnabled true if stable rate borrowing is enabled, false otherwise
 **/
event BorrowingEnabledOnReserve(address asset, bool stableRateEnabled);

/**
 * @dev emitted when borrowing is disabled on a reserve
 * @param asset the address of the reserve
 **/
event BorrowingDisabledOnReserve(address indexed asset);
```

Listing 3.24: LendingPoolConfigurator.sol

It comes to our attention that the event BorrowingEnabledOnReserve has not indexed the asset information. Note that each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, which means it will be attached as data (instead of a separate topic). Considering that the asset is typically queried, it is typically treated as a topic, hence the need of being indexed.

There are a few other events that also do not index the asset information, including ReserveBaseLtvChanged, ReserveFactorChanged, ReserveLiquidationThresholdChanged, ReserveLiquidationBonusChanged, ReserveDecimalsChanged, ReserveInterestRateStrategyChanged, ATokenUpgraded, StableDebtTokenUpgraded, and VariableDebtTokenUpgraded.

Recommendation Revise the above events by properly indexing the emitted asset information.

Status The issue has been confirmed and accordingly fixed by this merge request: `112`.

3.18 Performance Optimization in _updateIndexes()

- ID: PVE-018
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ReserveLogic`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [3]

Description

In Aave V2, the borrow/liquidity indexes play a critical role in calculating the accrued interests for both lenders and borrows. There is always a need to keep them updated with the current block height. Naturally, the related helper routine `updateIndexes()` is in the execution path of every single core functionality in `LendingPool` and always needs to be executed (Step II in Section 3.11) before any protocol-wide state can be changed. Therefore, we need to pay extra attention to this helper routine and optimize its execution.

Our analysis shows that this helper routine can be better optimized by reducing at least one internal transaction. To elaborate, we show below the code snippet of `updateState()`.

```
/**
 * @dev Updates the liquidity cumulative index Ci and variable borrow cumulative index
     Bvc. Refer to the whitepaper for
 * a formal specification.
 * @param reserve the reserve object
 **/
function updateState(ReserveData storage reserve) external {
  address variableDebtToken = reserve.variableDebtTokenAddress;
  uint256 previousVariableBorrowIndex = reserve.variableBorrowIndex;
  uint256 previousLiquidityIndex = reserve.liquidityIndex;

  (uint256 newLiquidityIndex, uint256 newVariableBorrowIndex) = _updateIndexes(
    reserve,
    variableDebtToken,
    previousLiquidityIndex,
    previousVariableBorrowIndex
  );

  _mintToTreasury(
    reserve,
    variableDebtToken,
    previousVariableBorrowIndex,
    newLiquidityIndex,
```

```
      newVariableBorrowIndex
    );
  }
```

Listing 3.25: ReserveLogic.sol

The routine has two main functionalities: the first one delegates the call to its internal routine `_updateIndexes()` for actual index updates while the second one calculates the new amount minted to the treasury (Section 3.20). The internal routine properly calculates `currentLiquidityRate` and `cumulatedLiquidityInterest` for the `liquidityIndex` update (line 390). In addition, it evaluates current `cumulatedVariableBorrowInterest` for the `variableBorrowIndex` update (line 401).

```
361   /**
362    * @dev updates the reserve indexes and the timestamp of the update
363    * @param reserve the reserve reserve to be updated
364    * @param variableDebtToken the debt token address
365    * @param liquidityIndex the last stored liquidity index
366    * @param variableBorrowIndex the last stored variable borrow index
367    **/
368   function _updateIndexes(
369     ReserveData storage reserve,
370     address variableDebtToken,
371     uint256 liquidityIndex,
372     uint256 variableBorrowIndex
373   ) internal returns (uint256, uint256) {
374     uint40 timestamp = reserve.lastUpdateTimestamp;
375 
376     uint256 currentLiquidityRate = reserve.currentLiquidityRate;
377 
378     uint256 newLiquidityIndex = liquidityIndex;
379     uint256 newVariableBorrowIndex = variableBorrowIndex;
380 
381     //only cumulating if there is any income being produced
382     if (currentLiquidityRate > 0) {
383       uint256 cumulatedLiquidityInterest = MathUtils.calculateLinearInterest(
384         currentLiquidityRate,
385         timestamp
386       );
387       newLiquidityIndex = cumulatedLiquidityInterest.rayMul(liquidityIndex);
388       require(newLiquidityIndex < (1 << 128), Errors.LIQUIDITY_INDEX_OVERFLOW);
389 
390       reserve.liquidityIndex = uint128(newLiquidityIndex);
391 
392       //as the liquidity rate might come only from stable rate loans, we need to ensure
393       //that there is actual variable debt before accumulating
394       if (IERC20(variableDebtToken).totalSupply() > 0) {
395         uint256 cumulatedVariableBorrowInterest = MathUtils.calculateCompoundedInterest(
396           reserve.currentVariableBorrowRate,
397           timestamp
398         );
399         newVariableBorrowIndex = cumulatedVariableBorrowInterest.rayMul(
            variableBorrowIndex);
```

```
400         require(newVariableBorrowIndex < (1 << 128), Errors.
                VARIABLE_BORROW_INDEX_OVERFLOW);
401         reserve.variableBorrowIndex = uint128(newVariableBorrowIndex);
402       }
403     }
404
405     //solium-disable-next-line
406     reserve.lastUpdateTimestamp = uint40(block.timestamp);
407     return (newLiquidityIndex, newVariableBorrowIndex);
408   }
409 }
```

Listing 3.26: ReserveLogic.sol

The evaluation of `newVariableBorrowIndex` (line 399) only occurs when there is a non-zero total supply of the related `variableDebtToken` (lines 394–402). The sanity check on `IERC20(variableDebtToken).totalSupply()> 0` (line 394) can be simplified as `IERC20(variableDebtToken).scaledTotalSupply()> 0`. This simplification saves the extra call to the lending pool for `getReserveNormalizedVariableDebt(UNDERLYING_ASSET)` (line 98 in `VariableDebtToken`).

```
93   /**
94    * @dev Returns the total supply of the variable debt token. Represents the total debt
          accrued by the users
95    * @return the total supply
96    **/
97   function totalSupply() public virtual override view returns (uint256) {
98     return super.totalSupply().rayMul(POOL.getReserveNormalizedVariableDebt(
         UNDERLYING_ASSET));
99   }
```

Listing 3.27: VariableDebtToken.sol

Recommendation Optimize the above `_updateIndexes()` logic by avoiding an unnecessary extra call (as an internal transaction) with the benefit of reduced gas cost.

Status The issue has been confirmed and accordingly fixed by this merge request: 77.

3.19 Inconsistent Handling of healthFactor Corner Cases

- ID: PVE-019
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GenericLogic`, `ValidationLogic`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

The borrowing and lending operations in Aave V2 require timely and accurate accounting of users' lending and debt positions. An important metric is the so-called `healthFactor` that measures the safety of a user's debt position. Note that a normal debt position needs to have its `healthFactor` no larger than the configured risk parameter — `HEALTH_FACTOR_ABOVE_THRESHOLD`.

```
function validateRepayWithCollateral(
  ReserveLogic.ReserveData storage collateralReserve,
  ReserveLogic.ReserveData storage principalReserve,
  UserConfiguration.Map storage userConfig,
  address user,
  uint256 userHealthFactor,
  uint256 userStableDebt,
  uint256 userVariableDebt
) internal view returns (uint256, string memory) {
  if (
    !collateralReserve.configuration.getActive() || !principalReserve.configuration.
        getActive()
  ) {
    return (uint256(Errors.CollateralManagerErrors.NO_ACTIVE_RESERVE), Errors.
        NO_ACTIVE_RESERVE);
  }

  if (
    msg.sender != user && userHealthFactor >= GenericLogic.
        HEALTH_FACTOR_LIQUIDATION_THRESHOLD
  ) {
    return (
      uint256(Errors.CollateralManagerErrors.HEALTH_FACTOR_ABOVE_THRESHOLD),
      Errors.HEALTH_FACTOR_NOT_BELOW_THRESHOLD
    );
  }
  ...
}
```

Listing 3.28: ValidationLogic.sol

During the analysis of the `healthFactor` enforcement through the entire protocol, we notice the discrepancy in the handling of a specific corner case when current `healthFactor` is equal to

HEALTH_FACTOR_ABOVE_THRESHOLD. As an example, the validateRepayWithCollateral() routine considers that the equal case is healthy while the balanceDecreaseAllowed() routine considers that the equal case is unhealthy.

```
function balanceDecreaseAllowed(
  address asset,
  address user,
  uint256 amount,
  mapping(address => ReserveLogic.ReserveData) storage reservesData,
  UserConfiguration.Map calldata userConfig,
  address[] calldata reserves,
  address oracle
) external view returns (bool) {
  if (
    !userConfig.isBorrowingAny()
    !userConfig.isUsingAsCollateral(reservesData[asset].id)
  ) {
    return true;
  }

  balanceDecreaseAllowedLocalVars memory vars;

  (vars.ltv, , , vars.decimals) = reservesData[asset].configuration.getParams();

  if (vars.ltv == 0) {
    return true; //if reserve is not used as collateral, no reasons to block the
        transfer
  }

  (
    vars.collateralBalanceETH,
    vars.borrowBalanceETH,
    ,
    vars.currentLiquidationThreshold,

  ) = calculateUserAccountData(user, reservesData, userConfig, reserves, oracle);

  if (vars.borrowBalanceETH == 0) {
    return true; //no borrows - no reasons to block the transfer
  }

  vars.amountToDecreaseETH = IPriceOracleGetter(oracle).getAssetPrice(asset).mul(
      amount).div(
    10**vars.decimals
  );

  vars.collateralBalancefterDecrease = vars.collateralBalanceETH.sub(vars.
      amountToDecreaseETH);

  //if there is a borrow, there can't be 0 collateral
  if (vars.collateralBalancefterDecrease == 0) {
    return false;
```

```
    }

    vars.liquidationThresholdAfterDecrease = vars
      .collateralBalanceETH
      .mul(vars.currentLiquidationThreshold)
      .sub(vars.amountToDecreaseETH.mul(vars.reserveLiquidationThreshold))
      .div(vars.collateralBalancefterDecrease);

    uint256 healthFactorAfterDecrease = calculateHealthFactorFromBalances(
      vars.collateralBalancefterDecrease,
      vars.borrowBalanceETH,
      vars.liquidationThresholdAfterDecrease
    );

    return healthFactorAfterDecrease > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD;
  }
```

Listing 3.29: GenericLogic.sol

Recommendation Make a consistent enforcement of the `healthFactor` metric.

Status The issue has been confirmed and accordingly fixed by this merge request: `98`.

3.20 Inaccurate previousStableDebt Calculation in _mintToTreasury()

- ID: PVE-020
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ReserveLogic`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

As mentioned in Section 3.18, the borrow/liquidity indexes play a critical role in calculating the accrued interests for both lenders and borrows. In the same section, we discuss the related `updateIndexes()` routine with two main functionalities: the first one delegates the call to its internal routine `_updateIndexes()` for actual index updates while the second one calculates the new amount minted to the treasury (via `_mintToTreasury()`).

The new amount to the treasury is a mechanism to contribute part of the repaid interests to the reserve treasury. The amount depends on two numbers: the first one is the repaid interests and the second one is the risk parameter, i.e., `reserveFactor`.

During our analysis, we notice that the way to calculate the amount of repaid interests does not take into account the interests collected from stable debts. To elaborate, we show below the code snippet of the `_mintToTreasury()` routine that is responsible for the calculation.

```
function _mintToTreasury(
  ReserveData storage reserve,
  address variableDebtToken,
  uint256 previousVariableBorrowIndex,
  uint256 newLiquidityIndex,
  uint256 newVariableBorrowIndex
) internal {
  MintToTreasuryLocalVars memory vars;

  vars.reserveFactor = reserve.configuration.getReserveFactor();

  if (vars.reserveFactor == 0) {
    return;
  }

  //fetching the last scaled total variable debt
  vars.scaledVariableDebt = IVariableDebtToken(variableDebtToken).scaledTotalSupply();

  //fetching the principal, total stable debt and the avg stable rate
  (
    vars.principalStableDebt,
    vars.currentStableDebt,
    vars.avgStableRate,
    vars.stableSupplyUpdatedTimestamp
  ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();

  //calculate the last principal variable debt
  vars.previousVariableDebt = vars.scaledVariableDebt.rayMul(previousVariableBorrowIndex);

  //calculate the new total supply after accumulation of the index
  vars.currentVariableDebt = vars.scaledVariableDebt.rayMul(newVariableBorrowIndex);

  //calculate the stable debt until the last timestamp update
  vars.cumulatedStableInterest = MathUtils.calculateCompoundedInterest(
    vars.avgStableRate,
    vars.stableSupplyUpdatedTimestamp
  );

  vars.previousStableDebt = vars.principalStableDebt.rayMul(vars.cumulatedStableInterest);

  //debt accrued is the sum of the current debt minus the sum of the debt at the last update
  vars.totalDebtAccrued = vars
    .currentVariableDebt
    .add(vars.currentStableDebt)
    .sub(vars.previousVariableDebt)
```

```
354         .sub(vars.previousStableDebt);
355 
356     vars.amountToMint = vars.totalDebtAccrued.percentMul(vars.reserveFactor);
357 
358     IAToken(reserve.aTokenAddress).mintToTreasury(vars.amountToMint, newLiquidityIndex);
359   }
```

Listing 3.30: ReserveLogic.sol

The interests from stable debts can be derived by `currentVariableDeb - previousStableDebt`. The `currentVariableDeb` number is accurately obtained from the `getSupplyData()` call of the respective stable debt token. However, the `previousStableDebt` number is re-computed from the compounded interests based on the average stable rate and the last update timestamp of updated stable supply (lines 342 – 347). The re-computed compounded `previousStableDebt` in essence becomes the same as `currentStableDebt`, which zeros out the stable debt-related interests.

Recommendation Revise the calculation of stable debt interests to ensure the correct amount minted to the treasury.

Status The issue has been confirmed and accordingly fixed by this merge request: `99`.

3.21 Flashloan-Lowered StableBorrowRate For Mode-Switching Users

- ID: PVE-021
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LendingPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

Another unique feature implemented in Aave V2 is the support of both variable and stable borrow rates. The variable borrow rate follows closely the market dynamics and can be changed on each user interaction (either borrow, deposit, withdraw, repayment or liquidation). The stable borrow rate instead will be unaffected by these actions. However, implementing a fixed stable borrow rate model on top of a dynamic reserve pool is complicated and the protocol provides the rate-rebalancing support to work around dynamic changes in market conditions or increased cost of money within the pool.

In the following, we show the code snippet of `swapBorrowRateMode()` which allows users to swap between stable and variable borrow rate modes. It follows the same sequence of convention by firstly validating the inputs (Step I), secondly updating relevant reserve states (Step II), then switching the requested borrow rates (Step III), next calculating the latest interest rates (Step IV), and finally performing external interactions, if any (Section V).

```
/**
 * @dev borrowers can user this function to swap between stable and variable borrow
   rate modes.
 * @param asset the address of the reserve on which the user borrowed
 * @param rateMode the rate mode that the user wants to swap
 **/
function swapBorrowRateMode(address asset, uint256 rateMode) external override {
  _whenNotPaused();
  ReserveLogic.ReserveData storage reserve = _reserves[asset];

  (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(msg.sender,
      reserve);

  ReserveLogic.InterestRateMode interestRateMode = ReserveLogic.InterestRateMode(
      rateMode);

  ValidationLogic.validateSwapRateMode(
    reserve,
    _usersConfig[msg.sender],
    stableDebt,
    variableDebt,
    interestRateMode
  );

  reserve.updateState();

  if (interestRateMode == ReserveLogic.InterestRateMode.STABLE) {
    //burn stable rate tokens, mint variable rate tokens
    IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
    IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
      msg.sender,
      stableDebt,
      reserve.variableBorrowIndex
    );
  } else {
    //do the opposite
    IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
      msg.sender,
      variableDebt,
      reserve.variableBorrowIndex
    );
    IStableDebtToken(reserve.stableDebtTokenAddress).mint(
      msg.sender,
      variableDebt,
      reserve.currentStableBorrowRate
    );
  }
```

```
    reserve.updateInterestRates(asset, reserve.aTokenAddress, 0, 0);

    emit Swap(asset, msg.sender);
  }
```

Listing 3.31: LendingPool.sol

Our analysis shows this `swapBorrowRateMode()` routine can be affected by a flashloan-assisted sandwiching attack such that the new stable borrow rate becomes the lowest possible. Note this attack is applicable when the borrow rate is switched from variable to stable rate. Specifically, to perform the attack, a malicious actor can first request a flashloan to deposit into the reserve pool so that the reserve's utilization rate is close to 0, then invoke `swapBorrowRateMode()` to perform the variable-to-borrow rate switch and enjoy the lowest `currentStableBorrowRate` (thanks to the nearly 0 utilization rate in current reserve), and finally withdraw to return the flashloan. A similar approach can also be applied to bypass `maxStableLoanPercent` enforcement in `validateBorrow()`.

Recommendation Revise current execution logic of `swapBorrowRateMode()` to defensively detect sudden changes to a reserve utilization and block malicious attempts.

Status This issue has been confirmed. Note that this issue is partially mitigated by the IR that Aave is using, where the stable rate borrowing has a much flattened slope compared to the variable, so that there is not much difference between the stable rate borrowing at the breaking point and with the reserve completely empty. Important to note that this potential abuse is already present in V1 but there are no signs of borrowers actually using it.

3.22 Bypassed Enforcement of LIQUIDATION_CLOSE_FACTOR_PERCENT

- ID: PVE-022
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LendingPoolCollateralManager`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

Aave V2 defines a number of system-wide risk parameters. In Section 3.19, we discussed a risk parameter named `HEALTH_FACTOR_ABOVE_THRESHOLD` that specifies the threshold on the permitted `healthFactor`. In this section, we examine another risk parameter, i.e., `LIQUIDATION_CLOSE_FACTOR_PERCENT`. This risk parameter is applicable when a debt position is being liquidated and used to limit the liquidable principal amount for a particular `liquidationCall()`.

```
function liquidationCall(
  address collateral,
  address principal,
  address user,
  uint256 purchaseAmount,
  bool receiveAToken
) external returns (uint256, string memory) {
  ReserveLogic.ReserveData storage collateralReserve = _reserves[collateral];
  ReserveLogic.ReserveData storage principalReserve = _reserves[principal];
  UserConfiguration.Map storage userConfig = _usersConfig[user];

  LiquidationCallLocalVars memory vars;

  (, , , , vars.healthFactor) = GenericLogic.calculateUserAccountData(
    user,
    _reserves,
    _usersConfig[user],
    _reservesList,
    _addressesProvider.getPriceOracle()
  );

  //if the user hasn't borrowed the specific currency defined by asset, it cannot be
      liquidated
  (vars.userStableDebt, vars.userVariableDebt) = Helpers.getUserCurrentDebt(
    user,
    principalReserve
  );

  (vars.errorCode, vars.errorMsg) = ValidationLogic.validateLiquidationCall(
    collateralReserve,
    principalReserve,
    userConfig,
    vars.healthFactor,
    vars.userStableDebt,
    vars.userVariableDebt
  );

  if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
      .NO_ERROR) {
    return (vars.errorCode, vars.errorMsg);
  }

  vars.collateralAtoken = IAToken(collateralReserve.aTokenAddress);

  vars.userCollateralBalance = vars.collateralAtoken.balanceOf(user);

  vars.maxPrincipalAmountToLiquidate = vars.userStableDebt.add(vars.userVariableDebt).
      percentMul(
    LIQUIDATION_CLOSE_FACTOR_PERCENT
  );

```

```
187         vars.actualAmountToLiquidate = purchaseAmount > vars.maxPrincipalAmountToLiquidate
```

Listing 3.32: LendingPoolCollateralManager.sol

Specifically, based on the above code snippet of `liquidationCall()`, the maximum liquidable principal amount is calculated as `(userStableDebt + userVariableDebt)*(LIQUIDATION_CLOSE_FACTOR_PERCENT)` (lines $183-185$). However, we also notice another alternative routine, i.e., `repayWithCollateral()`, which can be similarly used to liquidate a default debt position but does not enforce this risk parameter. This creates an inconsistency in its enforcement.

Recommendation Properly enforce `LIQUIDATION_CLOSE_FACTOR_PERCENT`, a system-wide risk parameter that regulates the maximum liquidable principal amount.

Status The issue has been confirmed and accordingly fixed by this merge request: `86`.

4 Conclusion

In this audit, we have analyzed the Aave V2 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 V2 improves early versions by providing additional innovative features, e.g., debt tokenization, collateral trading, and new flashloans. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [15, 16, 17, 18, 20].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [21] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

[4] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

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

[9] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[12] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

[15] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[16] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[17] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[18] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[20] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[21] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.