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 介绍 . . . 5

1.1 关于 Aave V2 . . . 5

1.2 关于 PeckShield . . . 6

1.3 方法 . . . 6

1.4 免责声明 . . . 8

2 检测结果 . . . 10

2.1 总结 . . . 10

2.2 关键发现 . . . 11

3 检测结果详情 . . . 13

3.1 registerAddressesProvider() 的可改进的完整性检查 . . . 13

3.2 delegateBorrowAllowance() 和 borrow() 的竞争条件 . . . 15

3.3 通缩性/变基代币的不一致性 . . . 17

3.4 repay() 逻辑的简化和优化 . . . 18

3.5 验证 transferFrom() 返回值 . . . 21

3.6 先乘后除带来的精确度提升 . . . 23

3.7 改进的 STABLE_BORROWING_MASK . . . 25

3.8 AToken 中不准确的 Burn 事件 . . . 27

3.9 储备金（Reserve）和 AToken 的资产一致性 . . . 28

3.10 不精确的铸币计算 . . . 30

3.11 updateInterestRates() 在 DebtToken 改变前过早的更新 . . . 32

3.12 updateInterestRates() 在 AToken 更新后过迟的更新 . . . 34

3.13 设计文档和实现的不一致 . . . 38

3.14 移除未被使用的代码 . . . 39

3.15 通过对 LendingPool 的许可（allowance），任意基于合约的钱包可能造成经济损失 40

3.16 validateWithdraw() 中可改进的业务逻辑 . . . 42

3.17 在被索引的资产中可改进的事件（event）生成 . . . 45

3.18 _updateIndexes() 中的性能优化 . . . 46

3.19 针对 healthFactor 的边缘用例的不一致处理 . . . 49
3.20 _mintToTreasury() 中不准确的 previousStableDebt 计算 . . . 51
3.21 模式切换造成的过低 StableBorrowRate . . . 53
3.22 被绕过的 LIQUIDATION_CLOSE_FACTOR_PERCENT 限制 . . . 55

4 结论 57

5 附录 58

5.1 基础编码漏洞 . . . 58
5.1.1 构造器不匹配 . . . 58
5.1.2 夺权 . . . 58
5.1.3 多余的回退函数 . . . 58
5.1.4 上溢/下溢 . . . 58
5.1.5 重入 . . . 59
5.1.6 代币赠予 . . . 59
5.1.7 黑洞地址 . . . 59
5.1.8 未被授权的合约销毁 . . . 59
5.1.9 回滚型拒绝服务攻击 . . . 59
5.1.10 未被校验的外部调用 . . . 59
5.1.11 Gas 不足的 send 调用 . . . 60
5.1.12 使用 send 而非 transfer . . . 60
5.1.13 开销过大的循环体 . . . 60
5.1.14 使用不受信的库 . . . 60
5.1.15 使用可被预测的变量 . . . 60
5.1.16 交易顺序依赖 . . . 60
5.1.17 使用被废弃的变量 . . . 61
5.2 语义一致性检查 . . . 61
5.3 额外建议 . . . 61
5.3.1 避免使用可变字节序列 . . . 61
5.3.2 明确可见层级 . . . 61
5.3.3 明晰类型推断 . . . 61
5.3.4 严格遵守函数声明 . . . 62

References 63

PeckShield

1 介绍

通过审计 **Aave V2** 的设计文档和相关的智能合约源代码，我们（**PeckShield**）系统性地评估了智能合约实现中的潜在的安全问题，将智能合约代码和设计文档中的语义不一致暴露出来，并且相应地提供了额外的建议或者是推荐的修复方式。分析结果表明，Aave V2 当前版本中的代码在安全性和性能上仍然有改进的空间。本文档对审计结果做了分析和阐述。

1.1 关于 Aave V2

Aave 是一个去中心化的非托管货币市场协议，用户可以作为存款人或借款人参与。存款人为市场提供流动性以赚取被动收入，而借款人则可以以（永久性地）超额抵押或（单块流动性地）低额抵押的方式进行借款。Aave V2 不仅解决了 V1 中实现的一些非最优解决方案（例如，允许 `AToken` 的可升级性，简化了整体架构，便于 fuzzer 和形式化检验工具分析），还提供了额外的功能，例如，债务代币化、抵押品交易和闪电贷。

Aave V2 的基本信息如下：

表 1.1： Aave V2 的基本信息

条目	描述
发行方	Aave
官方网址	https://aave.com/
类型	以太坊智能合约
平台	Solidity
审计方法	白盒
审计完成时间	December 3, 2020

接下来，我们提供了被审计的文件的 Git 仓库链接，和被审计的分支的哈希值。注意，Aave V2 假定有一个可信的价格查询接口，能及时提供所支持的资产的市场价格，还有一个借贷价格接口，它能及时提供市场借贷利率。这两个接口不在本次审计范围内。

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

这个是修复了所有问题之后的分支哈希值：

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

1.2 关于 PeckShield

PeckShield (派盾) 是面向全球的业内顶尖区块链安全团队，以提升区块链生态整体的安全性、隐私性以及可用性为己任，通过发布行业趋势报告、实时监测生态安全风险，负责任曝光0day漏洞，以及提供相关的安全解决方案和服务等方式帮助社区抵御新兴的安全威胁。可以通过下列联系方式联络我们：Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), 或者 Email (contact@peckshield.com).

表 1.2: 漏洞危害性分类

影响力 \ 可能性	高	中	低
高	严重	高危	中危
中	高危	中危	低危
低	中危	低危	低危

PeckShield

1.3 方法

为了检测评估的标准化，我们根据 OWASP 风险评估方法 [14] 定义下列术语：

- 可能性 表示某个特定的漏洞被发现和利用的可能性；
- 影响力 度量了（利用该漏洞的）一次成功的攻击行动造成的损失；
- 危害性 显示该漏洞的危害的严重程度；

可能性和影响力各自被分为三个等级：高、中和低。危害性由可能性和影响力确定，分为四个等级：严重、高危、中危、低危，如表 1.2 所示。

为了评估风险，我们设置了一个检查项目清单，每个项目都会标明严重程度。对于每一个检查项目，如果我们的工具或分析没有发现任何问题，那么该合约对于该项目就被认为是安全的。对于任何被发现的问题，我们可能会进一步在我们的私有测试网中部署该合约，并运行测

表 1.3: 审计项目完整列表

类型	检查项目
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

试来确定结果。如果有必要的话，我们会额外构造一个 PoC 来证明漏洞利用的可能性。具体的检查项目列表如表 1.3 所示。

具体来说，我们按照以下程序进行审计：

- 基本编码错误：我们首先用我们专用的静态代码分析器静态分析给定智能合约的已知编码错误，然后手动验证（否认或确认）工具发现的所有问题。
- 语义一致性检查：然后我们手动检查已实现的智能合约的逻辑，并与白皮书中的描述进行比较。
- 针对 DeFi 业务逻辑的专门性检查：我们将进一步审查业务逻辑，检查系统操作，并对 DeFi 相关方面进行仔细检查，以发现可能的漏洞或错误。
- 附加建议：我们还从经过验证的编程实践角度提供了关于智能合约编码和开发的额外建议。

为了更好地描述我们所识别的每个问题，我们将发现通过 Common Weakness Enumeration（CWE-699） [13] 进行分类，这是一个由社区开发的软件缺陷类型列表，以更好地围绕软件开发中经常遇到的概念来划分和组织缺陷。虽然 CWE-699 中使用的一些类别可能与智能合约无关，但我们使用表 1.4 中的 CWE 类别来对我们的发现进行分类。

1.4 免责声明

请注意该审计报告并不保证能够发现 Aave V2 存在的一切安全问题，即评估结果并不能保证在未来不会发现新的安全问题。我们一向认为单次审计结果可能并不全面，因而推荐采取多个独立的审计和公开的漏洞奖赏计划相结合的方式来确保合约的安全性。最后必须要强调的是，审计结果不应构成任何投资建议。

表 1.4: 在本次审计中使用的 Common Weakness Enumeration (CWE) 分类

类别	总结
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 检测结果

2.1 总结

以下是我们分析 Aave V2 实现后的结论摘要。在审计的第一阶段，我们研究了智能合约的源代码，并在代码库上运行我们专用的静态代码分析器。这里的目的是静态识别已知的编码错误，然后手动验证（否认或确认）该工具所报告的问题。我们进一步手动审查了业务逻辑，检查了系统操作，并对 DeFi 相关方面进行仔细检查，以发现可能的漏洞和错误。

Severity	# of Findings	
严重	1	■
高危	2	■■
中危	6	■■■■■■
低危	8	■■■■■■■■
参考	5	■■■■■
总计	22	

我们已经确定了一些潜在的问题：其中一些问题涉及到以前可能没有被考虑到的边缘情况，而其他问题则是由于多个合同之间非正常的交互。因此，对于每一个问题，我们都构造了用于复现和/或验证的测试用例。经过进一步的分析和内部讨论，我们确定了几个严重程度不同的问题需要提出来并给予更多的关注，这些问题在上表中进行了分类。更多的信息可以在下一小节中找到，对每个问题的详细讨论在在第 3 章中呈现。

2.2 关键发现

总体来说，这些智能合约的设计和实现得都非常好，尽管可以通过解决所确定的问题（如表 2.1 所示）来进一步改进，总共包括了 1 个严重性漏洞、2 个高严重性漏洞、6 个中严重性漏洞、8 个低严重性漏洞和 5 个建议。

除了上述问题外，我们认为对于任何面向用户的应用和服务，建立必要的风险控制机制和制定应急预案都是非常重要的，这可能需要在主网部署前进行。风险控制机制应该在主网部署合约的那一刻开始启动。详见第 3 章。

表 2.1: 审计 Aave V2 关键发现

ID	严重度	名称	类别	状态
PVE-001	参考	registerAddressesProvider() 的可改进的完整性检查	Coding Practices	已修复
PVE-002	低危	delegateBorrowAllowance() 和 borrow() 的竞争条件	Time and State	已确认
PVE-003	低危	通缩性/变基代币的不一致性	Business Logic	已确认
PVE-004	低危	repay() 逻辑的简化和优化	Coding Practices	已修复
PVE-005	中危	验证 transferFrom() 返回值	Coding Practices	已修复
PVE-006	低危	先乘后除带来的精确度提升	Numeric Errors	已修复
PVE-007	低危	改进的 STABLE_BORROWING_MASK	Numeric Errors	已修复
PVE-008	低危	AToken 中不准确的 Burn 事件	Business Logic	已修复
PVE-009	参考	储备金（Reserve）和 AToken 的资产一致性	Time and State	已修复
PVE-010	中危	不精确的铸币计算	Business Logic	已修复
PVE-011	高危	updateInterestRates() 在 DebtToken 改变前过早的更新	Business Logic	已修复
PVE-012	高危	updateInterestRates() 在 AToken 更新后过迟的更新	Business Logic	已修复
PVE-013	参考	设计文档和实现的不一致	Coding Practices	已修复
PVE-014	参考	移除未被使用的代码	Coding Practices	已修复
PVE-015	严重	通过对 LendingPool 的许可（allowance），任意基于合约的钱包可能造成经济损失	Business Logic	已修复
PVE-016	中危	validateWithdraw() 中可改进的业务逻辑	Business Logic	已修复
PVE-017	参考	在被索引的资产中可改进的事件（event）生成	Business Logic	已修复
PVE-018	低危	_updateIndexes() 中的性能优化	Coding Practices	已修复
PVE-019	低危	针对 healthFactor 的边缘用例的不一致处理	Coding Practices	已修复
PVE-020	中危	_mintToTreasury() 中不准确的 previousStableDebt 计算	Business Logic	已修复
PVE-021	中危	模式切换造成的过低 StableBorrowRate	Time and State	已确认
PVE-022	中危	被绕过的 LIQUIDATION_CLOSE_FACTOR_PERCENT 限制	Business Logic	已修复

3 检测结果详情

3.1 registerAddressesProvider() 的可改进的完整性检查

- ID: PVE-001
- 危害性: 参考
- 可能性: N/A
- 影响力: N/A
- 目标: LendingPoolAddressesProviderRegistry
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

Aave V2 实现了一个模块化架构，使整个协议具有可扩展性和可插拔性。

为了方便模块化架构，Aave V2 有一个名为 LendingPoolAddressesProviderRegistry 的合约，其中包含了活跃的地址提供者列表。每个地址提供者都维护了一个内部映射，用来检索 Aave V2 中不同组件的当前实现。

在分析 LendingPoolAddressesProviderRegistry 的过程中，我们注意到它有一个受限制的公共函数，即 registerAddressesProvider()。这个函数只能由有特权的所有者调用，顾名思义，它允许注册一个新的地址提供者。每个注册的地址提供者都有一个相关的 id，用于识别其唯一性。

```
52  /**
53   * @dev adds a lending pool to the list of registered lending pools
54   * @param provider the pool address to be registered
55   **/
56  function registerAddressesProvider(address provider, uint256 id) external override
        onlyOwner {
57    _addressesProviders[provider] = id;
58    _addToAddressesProvidersList(provider);
59    emit AddressesProviderRegistered(provider);
60  }

62  /**
63   * @dev removes a lending pool from the list of registered lending pools
64   * @param provider the pool address to be unregistered
65   **/
66  function unregisterAddressesProvider(address provider) external override onlyOwner {
```

```
67      require(_addressesProviders[provider] > 0, Errors.PROVIDER_NOT_REGISTERED);
68      _addressesProviders[provider] = 0;
69      emit AddressesProviderUnregistered(provider);
70    }
```

Listing 3.1: LendingPoolAddressesProviderRegistry.sol

当需要取消注册地址提供者时，相应的 id 会被简单地重置为 0，有了这一点，就需要在registerAddressesProvider() 中进行额外的完整性检查，以确保相关的 id 不等于 0。

推荐方法 对于任意的已注册的地址提供者确保相关的 id > 0，例如：

```
52    /**
53     * @dev adds a lending pool to the list of registered lending pools
54     * @param provider the pool address to be registered
55     **/
56    function registerAddressesProvider(address provider, uint256 id) external override
          onlyOwner {
57      require(id !=0, Errors.PROVIDER_NOT_REGISTERED);
58      _addressesProviders[provider] = id;
59      _addToAddressesProvidersList(provider);
60      emit AddressesProviderRegistered(provider);
61    }

63    /**
64     * @dev removes a lending pool from the list of registered lending pools
65     * @param provider the pool address to be unregistered
66     **/
67    function unregisterAddressesProvider(address provider) external override onlyOwner {
68      require(_addressesProviders[provider] > 0, Errors.PROVIDER_NOT_REGISTERED);
69      _addressesProviders[provider] = 0;
70      emit AddressesProviderUnregistered(provider);
71    }
```

Listing 3.2: LendingPoolAddressesProviderRegistry.sol (revised)

状态 这个问题已经被确认并在第 82 号 merge 中被修复

3.2 delegateBorrowAllowance() 和 borrow() 的竞争条件

- ID: PVE-002
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: LendingPool
- 类别: Time and State [9]
- CWE 子类: CWE-362 [5]

描述

LendingPool 是 Aave V2 中的一个核心池合约，实现了多种创新功能。其中之一就是所谓的信用委托（credit delegation），其实质是只要用户接受其他提供抵押物的用户的委托，就可以进行无抵押贷款。该功能主要通过一对相关函数来实现，即 delegateBorrowAllowance() 和 borrow() 。为了详细说明，下面我们展示这两个函数的相关代码片段。delegateBorrowAllowance() 函数为某一特定用户地址的某类债务资产设置了预定的借款额度（197 行的 _borrowAllowance），而当用户确实从池中请求 borrow() 时，该额度将被减少。

```
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
205    * @param amount the amount to be borrowed
206    * @param interestRateMode the interest rate mode at which the user wants to borrow.
          Can be 0 (STABLE) or 1 (VARIABLE)
207    * @param referralCode a referral code for integrators
208    * @param onBehalfOf address of the user who will receive the debt
```

```
209    **/
210   function borrow(
211     address asset,
212     uint256 amount,
213     uint256 interestRateMode,
214     uint16 referralCode,
215     address onBehalfOf
216   ) external override {
217     _whenNotPaused();
218     ReserveLogic.ReserveData storage reserve = _reserves[asset];

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

223       _borrowAllowance[debtToken][onBehalfOf][msg
224         .sender] = _borrowAllowance[debtToken][onBehalfOf][msg.sender].sub(
225         amount,
226         Errors.BORROW_ALLOWANCE_ARE_NOT_ENOUGH
227       );
228     }
229     _executeBorrow(
230       ExecuteBorrowParams(
231         asset,
232         msg.sender,
233         onBehalfOf,
234         amount,
235         interestRateMode,
236         reserve.aTokenAddress,
237         referralCode,
238         true
239       )
240     );
241   }
```

Listing 3.3: LendingPool.sol

这对函数类似于 ERC20 种的 `approval()` / `transferFrom()`，并且有一个类似的已知竞争条件问题 [2]。具体来说，当用户打算将之前批准的 `_borrowAllowance` 借款额度（从比如 10 `DAI`）减少（到 1 `DAI`）。用户可能会借到之前批准的 `_borrowAllowance`（10 `DAI`），然后再额外借到刚刚批准的新额度（1 `DAI`）。这就打破了用户将借款额度限制在新额度的意图，而不是旧额度和新额度之和。 为了正确处理 `_borrowAllowance`，还存在一个已知的变通方法：用户可以利用 `increaseBorrowApproval()` 和 `decreaseBorrowApproval()` 函数，而不是传统的 `delegateBorrowAllowance()` 函数。

推荐方法 添加如建议所示的变通方法 `increaseBorrowApproval()` 和 `decreaseBorrowApproval()` 函数。然而，考虑到利用竞争条件的难度和可能带来的收益，我们也认为保持现状是合理的。

状态 这个问题得到确认。正如 `approval()` / `transferFrom()` 一样，这没有很好的解决办法。开发团队决定将确保应用开发者和使用者意识到此局限性。

3.3 通缩性/变基代币的不一致性

- ID: PVE-003
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: LendingPool
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

在 Aave V2 中，LendingPool 合约被设计为与借贷用户交互的主要入口。特别是一个入口函数，deposit()，接受资产转入，并铸造相应的 AToken 来代表存款人在借贷池中的份额。当然，该合约实现了许多帮助函数，以将资产转入或转出 Aave V2。这些资产转移函数被设计为按照标准 ERC20 代币的标准工作：即金库的内部资产余额始终与个人 ERC20 代币合约中维护的实际代币余额一致。

```
91   /**
92    * @dev deposits The underlying asset into the reserve. A corresponding amount of the
        overlying asset (aTokens)
93    * is minted.
94    * @param asset the address of the reserve
95    * @param amount the amount to be deposited
96    * @param referralCode integrators are assigned a referral code and can potentially
        receive rewards.
97    **/
98   function deposit(
99     address asset,
100    uint256 amount,
101    address onBehalfOf,
102    uint16 referralCode
103  ) external override {
104    _whenNotPaused();
105    ReserveLogic.ReserveData storage reserve = _reserves[asset];

107    ValidationLogic.validateDeposit(reserve, amount);

109    address aToken = reserve.aTokenAddress;

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

114    bool isFirstDeposit = IAToken(aToken).balanceOf(onBehalfOf) == 0;
115    if (isFirstDeposit) {
116      _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
117    }

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

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

124      emit Deposit(asset, msg.sender, onBehalfOf, amount, referralCode);
125    }
```

Listing 3.4: LendingPool.sol

然而，存在一些 ERC20 代币可能会对其 ERC20 合约进行某些定制。这些代币的一种类型是通缩性（deflanationary）代币，它对每一次 transfer() 或 transferFrom() 收取一定的费用；另一种类型是变基（rebasing）代币，如 YAM。因此，这可能不符合这些资产转移函数背后的假设。换句话说，上述操作，如 deposit()，在比较内部资产记录和外部 ERC20 代币合约时，可能会引入意外的余额不一致。 一种可能的缓解措施是在资产转移函数之前和之后计算资产变化。换句话说，我们不需要期望 transfer() 或 transferFrom() 中的金额参数总是会导致全额转移，而是需要确保在 transfer() 或 transferFrom() 前后池中增加或减少的金额是符合预期的，并与我们的操作保持一致。虽然这些额外的检查会花费额外的 gas 使用量，但我们认为它们是必要的，以处理通缩性或其他定制的代币，如果支持他们是必要的。 另一个缓解措施是规范允许进入 Aave V2 进行借贷的 ERC20 代币。事实上，Aave V2 确实能够有效地规范可以上市的代币。同时，存在某些代币可能会有控制机制，可以动态地将代币转换为通缩性。

推荐方法 如果当前代码库需要支持通缩性代币，则需要在调用 transfer() / transferFrom() 前后检查余额，以确保记账金额准确。这种支持可能会带来额外的 gas 成本。另外，请记住，某些代币可能暂时不会通缩。然而，它们可能有一个控制开关，可以通过打开开关来将它们变成通缩性代币。一个例子是被广泛采用的 USDT。

状态 这个问题已经被开发团队确认。因为可以为每个资产开发特定的 AToken，因此将使用不同的 AToken 来列出余额可以变化的代币。

3.4 repay() 逻辑的简化和优化

- ID: PVE-004
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: LendingPool
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

LendingPool 合约实现了借贷中的一些核心功能。其中之一便是 repay() 操作，可以偿还借款人的部分或全部债务。在审查 repay() 逻辑时，我们注意到其执行逻辑还可以进一步改进。 为了详细说明，我们在下面展示了 repay() 的代码片段。其执行逻辑相当简单，首先更新/验证借款

人的债务状况，然后计算部分或全部 paybackAmount 并执行预定的还款，最后更新贷款池的利率并将付款重定向到 AToken 合约。

```
251   function repay(
252     address asset,
253     uint256 amount,
254     uint256 rateMode,
255     address onBehalfOf
256   ) external override {
257     _whenNotPaused();

259     ReserveLogic.ReserveData storage reserve = _reserves[asset];

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

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

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

270     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

优化和 `paybackAmount` 的计算（270 行 — 272 行）有关。`if (amount != type(uint256).max && amount < paybackAmount)paybackAmount = amount`。`amount != type(uint256).max` 条件实际上是无用的，因此计算可以被简化为 `if (amount < paybackAmount)paybackAmount = amount`。除此之外，Aave V2 提供了一系列的帮助函数来验证一些核心操作的给定参数，包括 `deposit()`，`withdraw()`，`borrow()`，`repay()` 等函数。在 `repay()` 中，需要修改执行`validateRepay()` 的方式。具体来说，`validateRepay()` 中包含的金额不应该是函数参数 （274 行 — 281 行），而应该是 paybackAmount（266 行 — 272 行）。

推荐方法 修改 `repay()` 的逻辑如下所示：

```
251   function repay(
252     address asset,
253     uint256 amount,
254     uint256 rateMode,
255     address onBehalfOf
256   ) external override {
257     _whenNotPaused();

259     ReserveLogic.ReserveData storage reserve = _reserves[asset];

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

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

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

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

272     ValidationLogic.validateRepay(
273       reserve,
274       paybackAmount,
275       interestRateMode,
276       onBehalfOf,
277       stableDebt,
```

```
278         variableDebt
279       );

281       reserve.updateState();

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

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

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

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

303       emit Repay(asset, onBehalfOf, msg.sender, paybackAmount);
304     }
```

Listing 3.6: LendingPool.sol

状态 这个问题被开发团队确认并在 82 号 merge 中被修复。

3.5 验证 transferFrom() 返回值

- ID: PVE-005
- 严重性: 中危
- 可能性: 中
- 影响力: 中
- 目标: `LendingPool`
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

`LendingPool` 合约中实现的另一个核心功能是新的闪电贷功能。该功能允许创建各种工具，用于再融资、抵押品交换、套利和清算。它进一步解决了早期版本的局限性，允许在 Aave 协议中使用闪电贷（开发团队仔细考虑了潜在的重入问题和其他限制）。 为了详细说明，我们在下面展示了 `flashLoan()` 的代码片段。这个函数以两种不同的模式工作。第一种模式是简单地维

护一个不变量，保证贷款池的最终余额大于之前的余额，再加上为这笔闪电贷收取的利息；第二种模式是允许从贷款池中闪电贷，假设这笔贷款是被之前的抵押品抵押的。

```
574   function flashLoan(
575     address receiverAddress,
576     address asset,
577     uint256 amount,
578     uint256 mode,
579     bytes calldata params,
580     uint16 referralCode
581   ) external override {
582     _whenNotPaused();
583     ReserveLogic.ReserveData storage reserve = _reserves[asset];
584     FlashLoanLocalVars memory vars;

586     vars.aTokenAddress = reserve.aTokenAddress;

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

590     ValidationLogic.validateFlashloan(mode, vars.premium);

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

594     vars.receiver = IFlashLoanReceiver(receiverAddress);

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

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

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

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

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

611       emit FlashLoan(receiverAddress, asset, amount, vars.premium, referralCode);
612     } else {
613       // If the transfer didn't succeed, the receiver either didn't return the funds, or
            didn't approve the transfer.
614       _executeBorrow(
615         ExecuteBorrowParams(
616           asset,
617           msg.sender,
618           msg.sender,
619           vars.amountPlusPremium,
```

```
620                 mode,
621                 vars.aTokenAddress,
622                 referralCode,
623                 false
624             )
625         );
626     }
627 }
```

Listing 3.7: LendingPool.sol

在审计第一种模式时，我们注意到，当闪电贷被转移到借款人时，它被 safeTransfer()（AToken 合约中的第 245 行）正确处理，安全地验证了返回值。然而，当闪电贷与必要的利息一起返回池子时，在 transferFrom() 处理时并未验证返回值。为了更好地适应与 ERC20 代币的不同实现或定制相关的各种特殊情况，我们强烈建议用 OpenZeppelin 的安全版本的 safeTransferFrom() 替换 transferFrom()。这个问题也适用于 LendingPoolCollateralManager 中的其他两个不安全的转账操作。

推荐方法 替换所有的 transferFrom() 为 OpenZeppelin 提供的安全的 safeTransferFrom()。类似地，替换不安全的 transfer() 为 safeTransfer()。

状态 这个问题已经被开发团队确认，并且在 56 号 merge 中替换 transferFrom() 为 safeTransferFrom()。

3.6 先乘后除带来的精确度提升

- ID: PVE-006
- 严重性: 低危
- 可能性: 中
- 影响力: 低
- 目标: DefaultReserveInterestRateStrategy
- 类别: Numeric Errors [12]
- CWE 子类: CWE-190 [4]

描述

SafeMath 是一个被广泛使用的 Solidity 数学库，其设计目的是为了在处理 uint256 时，通过防止常见的向上溢出或向下溢出来支持一个更安全的算数操作。虽然它确实可以阻止常见的溢出问题，但 Solidity 中缺乏浮点数支持可能会引入另一个微妙但麻烦的问题：精度损失。在本节中，我们将研究一个可能的精度损失问题，它源于乘法（mul）和除法（div）结合时的不同顺序。 特别地，我们以 calculateInterestRates()（在 DefaultReserveInterestRateStrategy 合约中）为例。这个函数用于计算由于与贷款池相关的变化而产生的利率，无论是来自新借款还是存款。

```
119   function calculateInterestRates(
```

```
120     address reserve,
121     uint256 availableLiquidity,
122     uint256 totalStableDebt,
123     uint256 totalVariableDebt,
124     uint256 averageStableBorrowRate,
125     uint256 reserveFactor
126   )
127     external
128     override
129     view
130     returns (
131       uint256,
132       uint256,
133       uint256
134     )
135   {

137     CalcInterestRatesLocalVars memory vars;

139     vars.totalBorrows = totalStableDebt.add(totalVariableDebt);
140     vars.currentVariableBorrowRate = 0;
141     vars.currentStableBorrowRate = 0;
142     vars.currentLiquidityRate = 0;

144     uint256 utilizationRate = vars.totalBorrows == 0
145       ? 0
146       : vars.totalBorrows.rayDiv(availableLiquidity.add(vars.totalBorrows));

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

151     if (utilizationRate > OPTIMAL_UTILIZATION_RATE) {
152       uint256 excessUtilizationRateRatio = utilizationRate.sub(OPTIMAL_UTILIZATION_RATE)
            .rayDiv(
153         EXCESS_UTILIZATION_RATE
154       );

156       vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(_stableRateSlope1)
            .add(
157         _stableRateSlope2.rayMul(excessUtilizationRateRatio)
158       );

160       vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(_variableRateSlope1).
            add(
161         _variableRateSlope2.rayMul(excessUtilizationRateRatio)
162       );
163     } else {
164       vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(
165         _stableRateSlope1.rayMul(utilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE))
166       );
167       vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(
```

```
168            utilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE).rayMul(_variableRateSlope1)
169          );
170        }

172        vars.currentLiquidityRate = _getOverallBorrowRate(
173          totalStableDebt,
174          totalVariableDebt,
175          vars.currentVariableBorrowRate,
176          averageStableBorrowRate
177        )
178          .rayMul(utilizationRate)
179          .percentMul(PercentageMath.PERCENTAGE_FACTOR.sub(reserveFactor));

181        return (vars.currentLiquidityRate, vars.currentStableBorrowRate, vars.
               currentVariableBorrowRate);
182      }
```

Listing 3.8: DefaultReserveInterestRateStrategy.sol

我们注意到 `currentVariableBorrowRate` 的计算（第 167–169 行）涉及到乘法和除法的混合计算。为了提高精度，最好先计算乘法再计算除法，即`baseVariableBorrowRate.add(utilisationRate.rayMul(_variableRateSlope1).rayDiv(OPTIMAL_UTILIZATION_RATE)))`。同样地，`LendingPoolCollateralManager` 合约（第584行）中 `calculateAvailableCollateralToLiquidate()` 的计算也可以进行相应调整。注意它所造成的精度损失可能只是一个很小的数字，但在满足一定的边界条件时，它却起着至关重要的作用。我们应该尽可能地避免精度损失。

推荐方法 修改上述计算来解决可能的精度损失。

状态 这个问题已经被开发团队确认并且在 82 和 88 号 merge 中被修复。

3.7 改进的 STABLE_BORROWING_MASK

- ID: PVE-007
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: `ReserveConfiguration`
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

为了提高 gas 的使用效率和改善可扩展性，Aave V2 引入了一个位掩码来存储储备金配置（reserve configuration）。位掩码的定义如下：

具体来说，该位掩码的大小为 256 位，分为 11 段：`LTV`, `Liquidation Threshold`, `Liquidation Bonus`, `Decimal`, `isActive`, `isFreezed`, `Bororowing Enabled`, `Stable Bororowing Enabled`, `Reserved`, `Reserve`

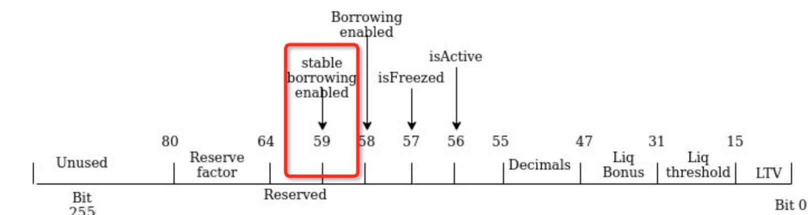

图 3.1: The Reserve Configuration Bitmask in Aave V2

Factor, 和 Unused。以上各段分别占用 16 位、16 位、16 位、8 位、1 位、1 位、1 位、1 位、5 位、16 位和 175 位。

```
15 library ReserveConfiguration {
16   uint256 constant LTV_MASK = 0xFFFFFFFFFFFFFFFF0000;
17   uint256 constant LIQUIDATION_THRESHOLD_MASK = 0xFFFFFFFFFFFF0000FFFF;
18   uint256 constant LIQUIDATION_BONUS_MASK = 0xFFFFFFF0000FFFFFFFF;
19   uint256 constant DECIMALS_MASK = 0xFFFFFF00FFFFFFFFFFFF;
20   uint256 constant ACTIVE_MASK = 0xFFFFFEFFFFFFFFFFFFFF;
21   uint256 constant FROZEN_MASK = 0xFFFFFDFFFFFFFFFFFFFF;
22   uint256 constant BORROWING_MASK = 0xFFFFFBFFFFFFFFFFFFFF;
23   uint256 constant STABLE_BORROWING_MASK = 0xFFFF07FFFFFFFFFFFFFF;
24   uint256 constant RESERVE_FACTOR_MASK = 0xFFFFFFFFFFFFFFFF;
25   ...
26 }
```

Listing 3.9: ReserveConfiguration.sol

我们检查了每个段的位掩码，并注意到 STABLE_BORROWING_MASK=0xFFFF07FFFFFFFFFFFFFF 需要5个比特，而不是 1 个。 由于 Stable Bororowing Enabled 的掩码的不正确，它通过受影响的 getStableRateBorrowingEnabled()/setStableRateBorrowingEnabled() 函数影响到了相邻的 4 个位（即使这4个位目前未被启用）[1]。

```
194   /**
195    * @dev enables or disables stable rate borrowing on the reserve
196    * @param self the reserve configuration
197    * @param enabled true if the stable rate borrowing needs to be enabled, false
        otherwise
198    **/
199   function setStableRateBorrowingEnabled(ReserveConfiguration.Map memory self, bool
       enabled)
200     internal pure
201   {
202     self.data = (self.data & STABLE_BORROWING_MASK) | (uint256(enabled ? 1 : 0) << 59);
203   }
204
205   /**
206    * @dev gets the stable rate borrowing state of the reserve
```

[1]与开发团队的后续讨论进一步 表明 LIQUIDATION_BONUS_MASK 在其掩码中漏掉了 F。换句话说。 它应该是 $0xFFFFFFFF0000FFFFFFFF$，而不是 $0xFFFFFFF0000FFFFFFFF$。

```
207    * @param self the reserve configuration
208    * @return the stable rate borrowing state
209    **/
210   function getStableRateBorrowingEnabled(ReserveConfiguration.Map storage self)
211     internal
212     view
213     returns (bool)
214   {
215     return ((self.data & ~STABLE_BORROWING_MASK) >> 59) != 0;
216   }
```

Listing 3.10: ReserveConfiguration.sol

推荐方法 将每个段计划占用的位严格对应。

状态 这个问题已经被开发团队确认并且在 63 号 merge 中被修复。

3.8 AToken 中不准确的 Burn 事件

- ID: PVE-008
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: AToken
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

在以太坊中，事件（event）是合约中不可缺少的一部分，主要用于记录各种合约运行时行为。特别是当事件被发出后，它会将传递进去的参数存储在交易日志中，这些日志可以被外部分析工具访问。事件可以在很多情况下被发出。一种特殊情况是当系统范围内的参数或设置被更改时。另一种情况是当代币被铸造、转移或销毁时。 在下文中，我们以 AToken 合约为例。该合约旨在将存入借贷池的资产进行代币化。因此，代币化的 AToken 可以被铸造、转移或销毁。在检查反映 AToken 行为的事件时，我们注意到 Burn 事件（第 111 行）包含了不正确的信息。具体来说，该事件被定义为 envet Burn(address indexed from, address indexed target, uint256 value, uint256 index)，它有许多参数：第一个参数 from 表示执行赎回或销毁操作的地址；第二个参数 target 为要赎回的金额，而最后一个参数 index 表示赎回发生时储备金（reserve）的最后索引。而这个事件包含了一个错误的 from 信息，它不应该是 msg.sender。相反，这里的 from 应该是 user，即 burn() 的第一个参数。

```
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
97     **/
```

```
98   function burn(
99     address user,
100    address receiverOfUnderlying,
101    uint256 amount,
102    uint256 index
103  ) external override onlyLendingPool {
104    _burn(user, amount.rayDiv(index));
105
106    //transfers the underlying to the target
107    IERC20(UNDERLYING_ASSET_ADDRESS).safeTransfer(receiverOfUnderlying, amount);
108
109    //transfer event to track balances
110    emit Transfer(user, address(0), amount);
111    emit Burn(msg.sender, receiverOfUnderlying, amount, index);
112  }
```

Listing 3.11: AToken.sol

推荐方法 正确地发出 Burn 事件，并且令其拥有正确的参数以及时地反映出合约状态变化。这对于外部分析工具是非常有用的。

状态 这个问题已经被开发团队确认并且在 107 号 merge 中被修复。

3.9 储备金（Reserve）和 AToken 的资产一致性

- ID: PVE-009
- 严重性: 参考
- 可能性: N/A
- 影响力: N/A
- 目标: ReserveLogic
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

如 3.2 节所述，LendingPool 是 Aave V2 中的一个核心池合约。LendingPool 的核心是储备金的概念：每个池子都持有一个特定于所支持的加密货币的储备金，以太坊中的总金额被定义为总流动性。储备金接受贷款人的存款，资产实际存储在特定资产 AToken 中。用户可以借入这些资金，前提是他们锁仓了更大价值的物品作为抵押品。 显然，支持的资产与其 AToken 之间存在一对一的映射。支持的资产和各自的储备金之间也存在一对一的映射。因此，储备金和 AToken 之间的映射也应该是一对一的。因此，强制确保一致性是很有帮助的，这样储备金就可以用共享了相同基础资产的相应 AToken 进行初始化。 为了详细说明，我们在下面展示了新储备金的初始化函数（initReserve()）。初始化函数需要五个参数，即 asset、ATokenAddress、stableDebtAddress、variableDebtAddress 和 interestRateStrategyAddress。当然，第一个参数 asset 需要与第二个参数 ATokenAddress 后面的基础资产保持一致。请注

意，ATokenAddress 有一个内部不可变的成员变量 UNDERLYING_ASSET_ADDRESS。因此，我们可以强制确保 asset 和 UNDERLYING_ASSET_ADDRESS 之间的一致性。

```
803   /**
804    * @dev initializes a reserve
805    * @param asset the address of the reserve
806    * @param aTokenAddress the address of the overlying aToken contract
807    * @param interestRateStrategyAddress the address of the interest rate strategy
          contract
808    **/
809   function initReserve(
810     address asset,
811     address aTokenAddress,
812     address stableDebtAddress,
813     address variableDebtAddress,
814     address interestRateStrategyAddress
815   ) external override {
816     _onlyLendingPoolConfigurator();
817     _reserves[asset].init(
818       aTokenAddress,
819       stableDebtAddress,
820       variableDebtAddress,
821       interestRateStrategyAddress
822     );
823     _addReserveToList(asset);
824   }
```

Listing 3.12: LendingPool.sol

推荐方法 确保储备金和资产对应的 AToken 之间的一致性

状态 这个问题已经被开发团队确认并且在 82 号 merge 中被修复。修复涉及到了 LendingPoolConfigurator，其中 initReserve() 函数现在从 AToken 中获取资产，并在 AToken、可变债务令牌和稳定债务令牌之间确保底层资产和借贷池地址的正确性。修复还涉及到对 DebtTokenBase 的一个小改动，以便在 AToken 和 DebtToken 之间为 POOL 和 UNDERLYING_ASSET_ADDRESS 提供一个通用接口。

3.10 不精确的铸币计算

- ID: PVE-010
- 严重性: 中危
- 可能性: 高
- 影响力: 低
- 目标: `AToken`
- 类别: Numeric Errors [12]
- CWE 子类: CWE-190 [4]

描述

正如在第 3.6 节中提到的，`SafeMath` 是一个广泛使用的 Solidity 数学库，它被设计用来支持安全的算数运算，防止在使用 `uint256` 时出现常见的溢出问题。虽然它确实可以阻止常见的溢出问题，但 Solidity 中缺乏浮点数支持可能会引入另一个微妙但麻烦的问题：精度损失。在本节中，我们将研究另一个与精度损失有关的问题。 具体来说，Aave V2 实现了 `WadRayMath` 库，它为 `wads`（精度为 18 位的十进制数）和 `rays`（精度为 27 位的小数）提供了乘法和除法函数。如果一个算术计算是在 `rays` 而不是 `wads` 上进行的，那么较高的精度有助于减少潜在的精度损失。作为一个例子，我们在下面展示了 `rayDiv()` 的代码片段，它将两个 `ray` 相除，结果四舍五入到最近的 `ray`。

```
116   /**
117    * @dev divides two ray, rounding half up to the nearest ray
118    * @param a ray
119    * @param b ray
120    * @return the result of a/b, in ray
121    **/
122   function rayDiv(uint256 a, uint256 b) internal pure returns (uint256) {
123     require(b != 0, Errors.DIVISION_BY_ZERO);

125     uint256 halfB = b / 2;

127     uint256 result = a * RAY;

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

131     result += halfB;

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

135     return result / b;
136   }
```

Listing 3.13: WadRayMath.sol

为了减少精度损失，Aave V2 采取了惯例，即使用 `rays` 来计算利率和指数（`index`），而使用 `wads` 来计算代币余额和金额。我们据此检查了这个惯例的执行情况，并注意到有一个函数，即 `mintToTreasury()` 违反了这个惯例。 为了详细说明，我们在下面展示了 `mintToTreasury()`

函数。为了适应借贷池中不断变化的指数，AToken 内部保留了缩放数。因此，铸币金额应该计算为 amount.rayDiv(index)，而不是 amount.div(index)。换句话说，当前的实现（第 134 行）违反了这一惯例，产生了错误的铸币金额。由于 wads 和 rays 之间的小数差异为 9，当前的铸币量变得较小，只有 1 / (10 * 9) 的预期数量。

```
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

推荐方法 替换 mintToTreasury() 中的 amount.div(index) 为 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

状态 这个问题已经被开发团队确认并且在 65 号 merge 中被修复。

3.11 updateInterestRates() 在 DebtToken 改变前过早的更新

- ID: PVE-011
- 严重性: 高危
- 可能性: 高
- 影响力: 中
- 目标: LendingPoolCollateralManager
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

LendingPool 合约提供了一些核心函数供借贷用户交互，包括 deposit(), withdraw(), borrow(), repay(), flashloan() 等。为了方便执行每个核心函数，Aave V2 会用 ValidationLogic 中相应的验证函数验证这些核心函数的给定参数，如 validateDeposit(), validateWithdraw(), validateBorrow(), validateRepay(), validateFlashloan() 等。 更重要的是，在每个核心函数中执行的所有操作都遵循一个特定的序列：

- 第一步：首先验证给定的参数和当前状态。如果当前状态不能满足预期行为所需的前提条件，交易将被回滚。
- 第二步：更新储备金状态，以重新反映最新的借贷/流动性指数（到当前的区块高度），并进一步计算新的将被铸币的金额。更新后的指数对于使储备金执行预期的行为是必要的。
- 第三步：接下来，它将「执行」预期的行为，可能需要更新用户余额和储备金余额，因为该行动可能涉及将资产转入或转出储备金。更新可能会导致铸币或销毁与当前用户的借贷头寸相关的代币。这些代币被表示为 ATokens、StableDebtTokens 或 VariableDebtTokens。
- 第四步：由于上述行为可能导致储备金发生变化，如导致借贷双方的利用率出现不同，还需要相应调整利率，以准确计提利息。
- 第五步：通过遵循 check-effects-interactions 模式，最终执行外部交互。

Aave V2 中实现的先进功能之一是借贷头寸的代币化。当用户将资产存入一个特定的储备金时，用户会收到相应数量的 AToken，以代表存入的流动资金和应计利息。当用户开立或增加借款头寸时，用户会收到相应数量的 DebtTokens（根据借款模式的不同，可以是 StableDebtTokens 或 VariableDebtTokens）来代表债务头寸，并进一步计提债务利息。 上述序列需要正确维护。我们的分析表明，在几个函数中，updateInterestRates() (步骤四)在更新与用户相关联的余额数据（步骤三）之前就已执行。 为了详细说明，我们在下面展示了处理默认用户的清算请求的 liquidationCall() 的代码片段。具体来说，updateInterestRates()（第227行）是在用户债务更新（第 234 — 251 行）之前执行的。这种顺序外的执行可能会导致计算出更高的利率，并以借款用户为代价进行计息!

```
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

推荐方法 在 updateInterestRates() 和借贷货币变化中维护一个正确的关系。上述代码片段正确的修复方式如下所示:

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

227     if (vars.userVariableDebt >= vars.actualAmountToLiquidate) {
228       IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
229         user,
230         vars.actualAmountToLiquidate,
231         principalReserve.variableBorrowIndex
232       );
233     } else {
234       IVariableDebtToken(principalReserve.variableDebtTokenAddress).burn(
235         user,
236         vars.userVariableDebt,
237         principalReserve.variableBorrowIndex
238       );

240       IStableDebtToken(principalReserve.stableDebtTokenAddress).burn(
241         user,
```

```
242             vars.actualAmountToLiquidate.sub(vars.userVariableDebt)
243           );
244         }

246         principalReserve.updateInterestRates(
247           principal,
248           principalReserve.aTokenAddress,
249           vars.actualAmountToLiquidate,
250           0
251         );
```

Listing 3.17: LendingPoolCollateralManager.sol

状态 这个问题已经被开发团队确认并且在 88 号 merge 中被修复。

3.12 updateInterestRates() 在 AToken 更新后过迟的更新

- ID: PVE-012
- 严重性: 高危
- 可能性: 高
- 影响力: 中
- 目标: LendingPoolCollateralManager
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

如第 3.11 节所述，LendingPool 合约提供了一些核心函数，供借贷用户访问其功能。在同一节中，我们还阐述了一个由于在债务头寸未被量化之前过早更新全系统利率而带来的问题。在本节中，我们将进一步分析利率计算，并报告另一个问题，该问题是由利率更新和 AToken 变化之间的无序执行引起的。 具体来说，当用户有任何转入或转出储备金时，用户的借贷头寸（由 AToken 的持有量代表）需要被正确更新。同时，我们需要维护正常的执行顺序，以便准确计算更新储备金的利率。 为了详细说明，我们在下面展示了 swapLiquidity() 函数的代码片段。

```
455     function swapLiquidity(
456       address receiverAddress,
457       address fromAsset,
458       address toAsset,
459       uint256 amountToSwap,
460       bytes calldata params
461     ) external returns (uint256, string memory) {
462       ReserveLogic.ReserveData storage fromReserve = _reserves[fromAsset];
463       ReserveLogic.ReserveData storage toReserve = _reserves[toAsset];

465       SwapLiquidityLocalVars memory vars;

467       (vars.errorCode, vars.errorMsg) = ValidationLogic.validateSwapLiquidity(
468         fromReserve,
```

```
469         toReserve,
470         fromAsset,
471         toAsset
472       );

474       if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
            .NO_ERROR) {
475         return (vars.errorCode, vars.errorMsg);
476       }

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

481       fromReserve.updateState();
482       toReserve.updateState();

484       if (vars.fromReserveAToken.balanceOf(msg.sender) == amountToSwap) {
485         _usersConfig[msg.sender].setUsingAsCollateral(fromReserve.id, false);
486       }

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

490       vars.fromReserveAToken.burn(
491         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

如果我们关注 `toReserve.updateInterestRates()`（第 518 — 523 行），在 `toAsset` 被转入相应的储备金和相关的 `ATokens` 被铸造后，利率被不适当地更新了。换句话说，`amountToReceive` 被计算了两次，导致计算出双倍的转入金额的错误，即 $2 * toReceiveAmount$。 因此，这种计算方式使得储备金在有较低的使用率的情况下，以较少的应计利息为代价出借给用户!

推荐方法 在 `updateInterestRates()` 和 `AToken` 改变间维护一个正确的顺序。上述代码片段的一个示范性修改如下所示:

```
455   function swapLiquidity(
456     address receiverAddress,
457     address fromAsset,
458     address toAsset,
459     uint256 amountToSwap,
460     bytes calldata params
461   ) external returns (uint256, string memory) {
462     ReserveLogic.ReserveData storage fromReserve = _reserves[fromAsset];
463     ReserveLogic.ReserveData storage toReserve = _reserves[toAsset];

465     SwapLiquidityLocalVars memory vars;

467     (vars.errorCode, vars.errorMsg) = ValidationLogic.validateSwapLiquidity(
468       fromReserve,
469       toReserve,
470       fromAsset,
471       toAsset
472     );

474     if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
          .NO_ERROR) {
475       return (vars.errorCode, vars.errorMsg);
476     }

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

481     fromReserve.updateState();
482     toReserve.updateState();
```

```
484     if (vars.fromReserveAToken.balanceOf(msg.sender) == amountToSwap) {
485       _usersConfig[msg.sender].setUsingAsCollateral(fromReserve.id, false);
486     }

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

490     vars.fromReserveAToken.burn(
491       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       toReserve.updateInterestRates(
508         toAsset,
509         address(vars.toReserveAToken),
510         vars.amountToReceive,
511         0
512       );
513       IERC20(toAsset).transferFrom(
514         receiverAddress,
515         address(vars.toReserveAToken),
516         vars.amountToReceive
517       );

519       if (vars.toReserveAToken.balanceOf(msg.sender) == 0) {
520         _usersConfig[msg.sender].setUsingAsCollateral(toReserve.id, true);
521       }

523       vars.toReserveAToken.mint(msg.sender, vars.amountToReceive, toReserve.
            liquidityIndex);
524     }

526     ...
527   }
```

Listing 3.19: LendingPoolCollateralManager.sol

状态 这个问题已经被开发团队确认并且在 87 号 merge 中被修复。并且 `transferFrom()` 被

转移到了计算利率之后。

3.13 设计文档和实现的不一致

- ID: PVE-013
- 严重性: 参考
- 可能性: N/A
- 影响力: N/A
- 目标: Multiple Contracts
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

有一些误导性的注释在 Solidity 代码中，这给理解和维护软件带来了不必要的障碍。 一些例子可以在 `LendingPool::rebalanceStableBorrowRate()` 的第 359 行，`StableDebtToken::_calculateBalanceIncrease()` 的 211 行，和 `LendingPoolCollateralManager::repayWithCollateral()` 的 301 行中被找到。以 `rebalanceStableBorrowRate()` 为例，前面的函数摘要表明，如果当前流动性利率大于用户稳定利率，用户的稳定利率将不会被重新平衡。但是，执行中（第 394 — 399 行）指出了 `usageRatio` 和 `currentLiquidityRate` 中的另外两个要求，且与用户稳定利率无关。

```
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
401    reserve.updateState();
402
403    IStableDebtToken(address(stableDebtToken)).burn(user, stableBorrowBalance);
404    IStableDebtToken(address(stableDebtToken)).mint(user, stableBorrowBalance, reserve.
           currentStableBorrowRate);
405
406    reserve.updateInterestRates(asset, aTokenAddress, 0, 0);
407
408    emit RebalanceStableBorrowRate(asset, user);
409
410  }
```

Listing 3.20: LendingPool.sol

推荐方法 确保设计文档（包括代码注释）和代码实现的一致性。

状态 这个问题已经被开发团队确认并且在 87 号 merge 中被修复。

3.14 移除未被使用的代码

- ID: PVE-014
- 严重性: 参考
- 可能性: N/A
- 影响力: N/A
- 目标: `GenericLogic`
- 类别: Coding Practices [10]
- CWE 子类: CWE-563 [6]

描述

Aave V2 很好地利用了一些引用合约，如 `ERC20`、`SafeERC20`、`SafeMath`、`VersionedInitializable` 和

Ownable，以方便其实现和组织代码。例如，到目前为止，LendingPool 智能合约已经导入了至少五个引用合约。然而，我们观察到其中包含了某些未使用的代码，或者存在不必要的冗余，而这些冗余是可以安全删除的。 例如，如果我们仔细检查 GenericLogic 合约后，发现有一个已经不用的常量： HEALTH_FACTOR_CRITICAL_THRESHOLD。这个常量显然是一个废弃的功能留下的。

```
19 library GenericLogic {
20   using ReserveLogic for ReserveLogic.ReserveData;
21   using SafeMath for uint256;
22   using WadRayMath for uint256;
23   using PercentageMath for uint256;
24   using ReserveConfiguration for ReserveConfiguration.Map;
25   using UserConfiguration for UserConfiguration.Map;
26
27   uint256 public constant HEALTH_FACTOR_LIQUIDATION_THRESHOLD = 1 ether;
28   uint256 public constant HEALTH_FACTOR_CRITICAL_THRESHOLD = 0.98 ether;
29   ...
30 }
```

Listing 3.21: GenericLogic.sol

除此之外，还有许多定义在 LendingPoolAddressesProvider 中未被使用的常量，这些常量也可以被移除。包括 WALLET_BALANCE_PROVIDER，LENDING_POOL_CORE，LENDING_POOL_FLASHLOAN_PROVIDER，和 DATA_PROVIDER。

推荐方法 考虑移除未被使用的代码和常量。

状态 这个问题已经被开发团队确认并且在 87 号 merge 中被修复。

3.15 通过对 LendingPool 的许可（allowance），任意基于合约的钱包可能造成经济损失

- ID: PVE-015
- 严重性: 严重
- 可能性: 高
- 影响力: 高
- 目标: LendingPoolCollateralManager, LendingPool
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

描述

在 LendingPool 提供的所有核心功能中，闪电贷是一个颠覆性的功能，它允许用户在单笔交易内从储备金中借款，只要用户归还借款金额加上额外的利息即可。在本节中，我们报告一个与闪电贷功能相关的问题。闪电贷功能改进了早期的版本，允许在 Aave V2 中使用借来的闪电贷（通过适当的措施来防止潜在的重入风险）。此外，它还无缝地集成了借贷功能，以避免在同

一交易中返回现金贷。 为了详细说明，我们在下面展示了该功能背后的 `flashLoan()` 的代码片段。

```
547   function flashLoan(
548     address receiverAddress,
549     address asset,
550     uint256 amount,
551     uint256 mode,
552     bytes calldata params,
553     uint16 referralCode
554   ) external override {
555     _whenNotPaused();
556     ReserveLogic.ReserveData storage reserve = _reserves[asset];
557     FlashLoanLocalVars memory vars;
558
559     vars.aTokenAddress = reserve.aTokenAddress;
560
561     vars.premium = amount.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000);
562
563     ValidationLogic.validateFlashloan(mode, vars.premium);
564
565     ReserveLogic.InterestRateMode debtMode = ReserveLogic.InterestRateMode(mode);
566
567     vars.receiver = IFlashLoanReceiver(receiverAddress);
568
569     //transfer funds to the receiver
570     IAToken(vars.aTokenAddress).transferUnderlyingTo(receiverAddress, amount);
571
572     //execute action of the receiver
573     vars.receiver.executeOperation(asset, amount, vars.premium, params);
574
575     vars.amountPlusPremium = amount.add(vars.premium);
576
577     if (debtMode == ReserveLogic.InterestRateMode.NONE) {
578       IERC20(asset).transferFrom(receiverAddress, vars.aTokenAddress, vars.
              amountPlusPremium);
579
580       reserve.updateState();
581       reserve.cumulateToLiquidityIndex(IERC20(vars.aTokenAddress).totalSupply(), vars.
              premium);
582       reserve.updateInterestRates(asset, vars.aTokenAddress, vars.premium, 0);
583
584       emit FlashLoan(receiverAddress, asset, amount, vars.premium, referralCode);
585     } else {
586       // If the transfer didn't succeed, the receiver either didn't return the funds, or
               didn't approve the transfer.
587       _executeBorrow(
588         ExecuteBorrowParams(
589           asset,
590           msg.sender,
591           msg.sender,
592           vars.amountPlusPremium,
```

```
593                 mode,
594                 vars.aTokenAddress,
595                 referralCode,
596                 false
597             )
598         );
599     }
600 }
```

Listing 3.22: LendingPool.sol

这个特殊的函数以一种简单的方式实现了闪电贷功能。它首先将资金转移到指定的接收方，然后调用指定的操作（executeOperation — 第 573 行），接下来从接收方转回资金或创建一个等价的借款。 然而，我们的分析表明，如果用户之前向 LendingPool 指定了某些许可（allowance），上述逻辑可能会被滥用并造成无辜用户的资金损失。具体来说，如果通过指定无辜用户作为 receiverAddress 参数并发起一笔闪电贷，那么 flashLoan()）的执行遵循这样的逻辑：首先将贷款金额转移到 receiverAddress 上，调用 receiver 上的 executeOperation()，然后将 receiver 上的金额加上利息（不大于允许的消费金额）转移回资金池。请注意，这笔闪电贷并不是由 receiverAddress 发起的，但是 receiverAddress 支付了与该笔闪电贷相关的利息。 同样的问题也适用于另外两个函数，即 swapLiquidity() 和 repayWithCollateral()。这个攻击可以直接盗取无辜用户的资金并为攻击者谋取利益。同时，我们需要提到的是， executeOperation() 将在给定的 receiverAddress 上被调用。编译器将在确保 receiverAddress 确实是一个合约的过程中执行一个完整性检查，因此将攻击只适用于基于合约的智能钱包。然而，当前的智能钱包可能有一个回退函数，可以允许 executeOperation() 调用继续进行而不被回滚。[2]

推荐方法 重新审视被影响的函数的设计，以避免从借贷池中启动 transferFrom() 调用。此外，重新审视的设计可以验证 executeOperation() 调用，以便成功地转回预期的资产（如果有的话）。

状态 这个问题已经被开发团队确认并且在 86 号 merge 中被修复。由于这个问题，swapLiquidity() 和 repayWithCollateral() 被移除。

3.16 validateWithdraw() 中可改进的业务逻辑

- ID: PVE-016
- 严重性: 中危
- 可能性: 中
- 影响力: 中
- 目标: ValidationLogic
- 类别: Business Logic [11]
- CWE 子类: CWE-841 [8]

[2]一个例子是那些在 InstaDApp() 的智能钱包，一个流行的门户网站用于简化 DeFi 用户的需求。

描述

Aave V2 将验证逻辑集中在 ValidationLogic 合约中，以简化 LendingPool 中各种核心功能的流程。具体来说，为了方便执行每个核心函数（如 deposit(), withdraw(), borrow(), repay(), 和 flashloan()），提供了相应的验证函数，包括 ValidationLogic、validateDeposit()、validateWithdraw()、validateBorrow()、validateRepay()、validateFlashloan()。在分析 validateWithdraw() 的验证逻辑时，我们注意到一个问题，就是在当前余额中而不是借款金额中验证借款金额。

```
45  /**
46   * @dev validates a withdraw action.
47   * @param reserveAddress the address of the reserve
48   * @param amount the amount to be withdrawn
49   * @param userBalance the balance of the user
50   */
51  function validateWithdraw(
52    address reserveAddress,
53    uint256 amount,
54    uint256 userBalance,
55    mapping(address => ReserveLogic.ReserveData) storage reservesData,
56    UserConfiguration.Map storage userConfig,
57    address[] calldata reserves,
58    address oracle
59  ) external view {
60    require(amount > 0, Errors.AMOUNT_NOT_GREATER_THAN_0);
61
62    require(amount <= userBalance, Errors.NOT_ENOUGH_AVAILABLE_USER_BALANCE);
63
64    require(
65      GenericLogic.balanceDecreaseAllowed(
66        reserveAddress,
67        msg.sender,
68        userBalance,
69        reservesData,
70        userConfig,
71        reserves,
72        oracle
73      ),
74      Errors.TRANSFER_NOT_ALLOWED
75    );
76  }
```

Listing 3.23: ValidationLogic.sol

为了详细说明，我们在上面展示了 validateWithdraw() 的代码片段。这个函数确保借款金额落在一个合适的范围内，即 $(0, userBalance]$，然后将验证委托给 GenericLogic.balanceDecreaseAllowed()。然而，委托调用是用 userBalance 转发的，而不是用实际的借款金额来验证这个数量的借款金额是否被允许。

推荐方法 修改 validateWithdraw() 逻辑，令其可以使用真正的借款金额而不是当前余额来

进行正确的校验。

状态 这个问题已经被开发团队确认并且在 69 号 merge 中被修复。

3.17 在被索引的资产中可改进的事件（event）生成

- ID: PVE-017
- 严重性: 参考
- 可能性: N/A
- 影响力: N/A
- 目标: LendingPoolConfigurator
- 类别: Time and State [9]
- CWE 子类: CWE-362 [5]

描述

在智能合约设计中，有意义的事件（event）是一个重要的部分，因为它们不仅可以极大地暴露智能合约的运行时行为，而且可以更好地理解智能合约的行为以方便链下分析。正如在 3.8 节中提到的，事件可以在很多场景下发出。一个特殊的情况是当系统范围的参数或设置被改变时。 我们检查了 Aave V2 中对系统范围参数的支持，并注意到与配置相关的 getter/setter 函数主要在 LendingPoolConfigurator 中实现。在下文中，我们列举了几个在 Aave V2 中被定义的代表性事件。

```
45  /**
46   * @dev emitted when borrowing is enabled on a reserve
47   * @param asset the address of the reserve
48   * @param stableRateEnabled true if stable rate borrowing is enabled, false otherwise
49   **/
50  event BorrowingEnabledOnReserve(address asset, bool stableRateEnabled);
51
52  /**
53   * @dev emitted when borrowing is disabled on a reserve
54   * @param asset the address of the reserve
55   **/
56  event BorrowingDisabledOnReserve(address indexed asset);
```

Listing 3.24: LendingPoolConfigurator.sol

我们注意到，BorrowingEnabledOnReserve 这个事件没有索引资产信息。请注意，每个发出的事件都被表示为一个主题（topic），这个主题通常由事件名称的签名（来自于 keccak256 哈希算法）和其参数的类型（uint256，字符串等）组成。每个有索引的类型将被视为一个额外的主题。如果一个参数没有被索引，这意味着它将作为数据（而不是一个单独的主题）被附加。考虑到资产通常会被查询，它通常会被视为一个主题，因此需要被索引。 还有一些其他事件也未对资产信息进行索引，包括 ReserveBaseLtvChanged，ReserveFactorChanged，ReserveLiquidationThresholdChanged，ReserveLiquidationBonusChanged，ReserveDecimalsChanged，ReserveInterestRateStrategyChanged，ATokenUpgraded，StableDebtTokenUpgraded，和 VariableDebtTokenUpgraded。

推荐方法 修改上述事件，令其将资产信息标记为索引的。

状态 这个问题已经被开发团队确认并且在 112 号 merge 中被修复。

3.18 _updateIndexes() 中的性能优化

- ID: PVE-018
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: ReserveLogic
- 类别: Coding Practices [10]
- CWE 子类: CWE-1041 [3]

描述

在 Aave V2 中，借款/流动性指数在计算贷款人和借款人的应计利息时起着关键作用。并且总是需要根据当前的块高来更新它们。当然，相关的辅助函数 updateIndexes() 在 LendingPool 中的每一个核心功能的执行路径中，在任何协议范围内的状态被改变之前，该函数总是需要被执行（ 3.11 节中的步骤二） 。因此，我们需要格外关注这个辅助函数，并优化其执行。 我们的分析表明，通过减少至少一个内部交易，可以更好地优化这个辅助函数。为了详细说明，我们在下面展示了 updateState() 的代码片段。

```
145   /**
146    * @dev Updates the liquidity cumulative index Ci and variable borrow cumulative index
          Bvc. Refer to the whitepaper for
147    * a formal specification.
148    * @param reserve the reserve object
149    **/
150   function updateState(ReserveData storage reserve) external {
151     address variableDebtToken = reserve.variableDebtTokenAddress;
152     uint256 previousVariableBorrowIndex = reserve.variableBorrowIndex;
153     uint256 previousLiquidityIndex = reserve.liquidityIndex;
154
155     (uint256 newLiquidityIndex, uint256 newVariableBorrowIndex) = _updateIndexes(
156       reserve,
157       variableDebtToken,
158       previousLiquidityIndex,
159       previousVariableBorrowIndex
160     );
161
162     _mintToTreasury(
163       reserve,
164       variableDebtToken,
165       previousVariableBorrowIndex,
166       newLiquidityIndex,
167       newVariableBorrowIndex
168     );
169   }
```

Listing 3.25: ReserveLogic.sol

该函数有两个主要的功能：第一个委托调用其内部函数 _updateIndexes() 进行实际的指数更新，而第二个则计算新的铸币量（第 3.20 节） 。内部函数正确地计算 currentLiquidityRate 和

cumulatedLiquidityInterest 以更新 liquidityIndex（第 390 行）。此外，它还为 variableBorrowIndex 的更新评估了当前的 cumulatedVariableBorrowInterest（第 401 行）。

```
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

newVariableBorrowIndex（第 399 行）的计算只有在相关的 variableDebtToken 的总供应量为非零时才会发生（第 394 — 402 行）。IERC20(variableDebtToken).totalSupply()>0（第 394 行）的完整性检查可以简化为 IERC20(variableDebtToken).scaledTotalSupply()>0。这个简化节省了对借贷池 getReserveNormalizedVariableDebt(UNDERLYING_ASSERT) 的额外调用（VariableDebtToken 中第 98 行）。

```
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

推荐方法 优化上面的 _updateIndexes() 逻辑，避免不必要的额外调用（作为一个内部交易），好处是降低 gas 成本。

状态 这个问题已经被开发团队确认并且在 77 号 merge 中被修复。

PeckShield

3.19 针对 healthFactor 的边缘用例的不一致处理

- ID: PVE-019
- 严重性: 低危
- 可能性: 低
- 影响力: 低
- 目标: GenericLogic, ValidationLogic
- 类别: Business Logic [11]
- CWE 子类: CWE-837 [7]

描述

Aave V2 中的借贷业务需要及时、准确地核算用户的借贷和债务状况。一个重要的指标是所谓的健康因子（healthFactor），它可以衡量用户的债务头寸的安全性。需要注意的是，一个正常的债务头寸，其健康因子需要不大于指定的风险参数 — HEALTH_FACTOR_ABOVE_THRESHOLD。

```
393   function validateRepayWithCollateral(
394     ReserveLogic.ReserveData storage collateralReserve,
395     ReserveLogic.ReserveData storage principalReserve,
396     UserConfiguration.Map storage userConfig,
397     address user,
398     uint256 userHealthFactor,
399     uint256 userStableDebt,
400     uint256 userVariableDebt
401   ) internal view returns (uint256, string memory) {
402     if (
403       !collateralReserve.configuration.getActive() || !principalReserve.configuration.
          getActive()
404     ) {
405       return (uint256(Errors.CollateralManagerErrors.NO_ACTIVE_RESERVE), Errors.
          NO_ACTIVE_RESERVE);
406     }
407
408     if (
409       msg.sender != user && userHealthFactor >= GenericLogic.
          HEALTH_FACTOR_LIQUIDATION_THRESHOLD
410     ) {
411       return (
412         uint256(Errors.CollateralManagerErrors.HEALTH_FACTOR_ABOVE_THRESHOLD),
413         Errors.HEALTH_FACTOR_NOT_BELOW_THRESHOLD
414       );
415     }
416     ...
417   }
```

Listing 3.28: ValidationLogic.sol

在分析整个协议的健康因子实施过程中，我们注意到当当前健康因子等于 HEALTH_FACTOR_ABOVE_THRESHOLD 时，在处理特定的边缘用例时存在差异。举个例子，validateRepayWithCollateral() 函数认为等额情况是健康的，而 balanceDecreaseAllowed() 函数认为等额情况是不健康的。

```
56  function balanceDecreaseAllowed(
57    address asset,
58    address user,
59    uint256 amount,
60    mapping(address => ReserveLogic.ReserveData) storage reservesData,
61    UserConfiguration.Map calldata userConfig,
62    address[] calldata reserves,
63    address oracle
64  ) external view returns (bool) {
65    if (
66      !userConfig.isBorrowingAny()
67      !userConfig.isUsingAsCollateral(reservesData[asset].id)
68    ) {
69      return true;
70    }
71
72    balanceDecreaseAllowedLocalVars memory vars;
73
74    (vars.ltv, , , vars.decimals) = reservesData[asset].configuration.getParams();
75
76    if (vars.ltv == 0) {
77      return true; //if reserve is not used as collateral, no reasons to block the
          transfer
78    }
79
80    (
81      vars.collateralBalanceETH,
82      vars.borrowBalanceETH,
83      ,
84      vars.currentLiquidationThreshold,
85
86    ) = calculateUserAccountData(user, reservesData, userConfig, reserves, oracle);
87
88    if (vars.borrowBalanceETH == 0) {
89      return true; //no borrows - no reasons to block the transfer
90    }
91
92    vars.amountToDecreaseETH = IPriceOracleGetter(oracle).getAssetPrice(asset).mul(
        amount).div(
93      10**vars.decimals
94    );
95
96    vars.collateralBalancefterDecrease = vars.collateralBalanceETH.sub(vars.
        amountToDecreaseETH);
97
98    //if there is a borrow, there can't be 0 collateral
99    if (vars.collateralBalancefterDecrease == 0) {
100     return false;
101   }
102
103   vars.liquidationThresholdAfterDecrease = vars
104     .collateralBalanceETH
```

```
105         .mul(vars.currentLiquidationThreshold)
106         .sub(vars.amountToDecreaseETH.mul(vars.reserveLiquidationThreshold))
107         .div(vars.collateralBalancefterDecrease);
108
109     uint256 healthFactorAfterDecrease = calculateHealthFactorFromBalances(
110       vars.collateralBalancefterDecrease,
111       vars.borrowBalanceETH,
112       vars.liquidationThresholdAfterDecrease
113     );
114
115     return healthFactorAfterDecrease > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD;
116   }
```

Listing 3.29: GenericLogic.sol

推荐方法 令 healthFactor 的实施过程中保持一致。

状态 这个问题已经被开发团队确认并且在 98 号 merge 中被修复。

3.20 _mintToTreasury() 中不准确的 previousStableDebt 计算

- ID: PVE-020
- 严重性: 中危
- 可能性: 中
- 影响力: 中
- 目标: ReserveLogic
- 类别: Business Logic [11]
- CWE 子类: CWE-837 [7]

描述

正如在第 3.18 节中提到的，借款/流动性指数在计算贷款人和借款人的应计利息时起着关键作用。在同一节中，我们讨论了相关的 updateIndexes() 函数，它有两个主要的功能：第一个委托调用其内部函数 _updateIndexes() 进行实际的指数更新，而第二个则计算新的铸币金额（通过 _mintToTreasury()）。新的入库金额是将部分已偿还的利息贡献给储备金库的机制。这个金额取决于两个数字：第一个是已偿还的利息，第二个是风险参数，即 reserveFactor。在分析过程中，我们注意到计算偿还利息金额的方式并没有考虑到从稳定债务中收集的利息。为了详细说明，我们在下面展示了负责计算的 _mintToTreasury() 函数的代码片段。

```
309   function _mintToTreasury(
310     ReserveData storage reserve,
311     address variableDebtToken,
312     uint256 previousVariableBorrowIndex,
313     uint256 newLiquidityIndex,
314     uint256 newVariableBorrowIndex
315   ) internal {
316     MintToTreasuryLocalVars memory vars;
317
```

```
318     vars.reserveFactor = reserve.configuration.getReserveFactor();
319
320     if (vars.reserveFactor == 0) {
321       return;
322     }
323
324     //fetching the last scaled total variable debt
325     vars.scaledVariableDebt = IVariableDebtToken(variableDebtToken).scaledTotalSupply();
326
327     //fetching the principal, total stable debt and the avg stable rate
328     (
329       vars.principalStableDebt,
330       vars.currentStableDebt,
331       vars.avgStableRate,
332       vars.stableSupplyUpdatedTimestamp
333     ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();
334
335     //calculate the last principal variable debt
336     vars.previousVariableDebt = vars.scaledVariableDebt.rayMul(
            previousVariableBorrowIndex);
337
338     //calculate the new total supply after accumulation of the index
339     vars.currentVariableDebt = vars.scaledVariableDebt.rayMul(newVariableBorrowIndex);
340
341     //calculate the stable debt until the last timestamp update
342     vars.cumulatedStableInterest = MathUtils.calculateCompoundedInterest(
343       vars.avgStableRate,
344       vars.stableSupplyUpdatedTimestamp
345     );
346
347     vars.previousStableDebt = vars.principalStableDebt.rayMul(vars.
            cumulatedStableInterest);
348
349     //debt accrued is the sum of the current debt minus the sum of the debt at the last
            update
350     vars.totalDebtAccrued = vars
351       .currentVariableDebt
352       .add(vars.currentStableDebt)
353       .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

稳定债务的利息可以由 `currentVariableDeb - previousStableDebt` 得出。该 `currentVariableDeb` 数是准确地从各自的稳定债务代币的 `getSupplyData()` 调用中获得的。但是，`previousStableDebt` 数是根据平均稳定利率和更新稳定供给的最后更新时间戳，从复利中重新计算出来的（第 342

— 347 行）。重新计算后的复利的 previousStableDebt 实质上变成了和 currentStableDebt 一样，将稳定债务相关利息归零。

推荐方法 修改稳定债务利息的计算以确保正确的铸币进库的金额。

状态 这个问题已经被开发团队确认并且在 99 号 merge 中被修复。

3.21 模式切换造成的过低 StableBorrowRate

- ID: PVE-021
- 严重性: 中危
- 可能性: 中
- 影响力: 中
- 目标: LendingPool
- 类别: Business Logic [11]
- CWE 子类: CWE-837 [7]

描述

Aave V2 中实现的另一个独特功能是支持可变和稳定的借款利率。可变的借款利率紧跟市场动态，可以在每次用户交互时改变（借款、存款、提款、还款或清算）。而稳定借款利率则不会受到这些操作的影响。然而，在动态储备金池上实现一个固定的稳定借款利率模型是非常复杂的，该协议提供了利率再平衡（rate-rebalancing）支持，以绕过市场条件的动态变化或池内资金成本的增加。 在下文中，我们展示了 swapBorrowRateMode() 的代码片段，它允许用户在稳定的和可变的借款利率模式之间进行切换。它遵循同样的约定顺序，首先验证输入（步骤一），其次更新相关储备金状态（步骤二），然后切换请求的借款利率（步骤三），接下来计算最新利率（步骤四），最后进行外部交互（如果有的话）（步骤五）。

```
308   /**
309    * @dev borrowers can user this function to swap between stable and variable borrow
         rate modes.
310    * @param asset the address of the reserve on which the user borrowed
311    * @param rateMode the rate mode that the user wants to swap
312    **/
313   function swapBorrowRateMode(address asset, uint256 rateMode) external override {
314     _whenNotPaused();
315     ReserveLogic.ReserveData storage reserve = _reserves[asset];

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

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

321     ValidationLogic.validateSwapRateMode(
322       reserve,
323       _usersConfig[msg.sender],
```

```
324             stableDebt,
325             variableDebt,
326             interestRateMode
327         );

329         reserve.updateState();

331         if (interestRateMode == ReserveLogic.InterestRateMode.STABLE) {
332             //burn stable rate tokens, mint variable rate tokens
333             IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
334             IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
335                 msg.sender,
336                 stableDebt,
337                 reserve.variableBorrowIndex
338             );
339         } else {
340             //do the opposite
341             IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
342                 msg.sender,
343                 variableDebt,
344                 reserve.variableBorrowIndex
345             );
346             IStableDebtToken(reserve.stableDebtTokenAddress).mint(
347                 msg.sender,
348                 variableDebt,
349                 reserve.currentStableBorrowRate
350             );
351         }

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

355         emit Swap(asset, msg.sender);
356     }
```

Listing 3.31: LendingPool.sol

我们的分析表明，这个 `swapBorrowRateMode()` 函数可以受到闪电贷辅助的夹层攻击（sandwiching attack）的影响，从而使新的稳定借款利率成为可能的最低利率。注意，当借款利率从可变利率切换到稳定利率时，这种攻击是适用的。具体来说，为了执行该攻击，恶意行为人可以先请求将闪电贷存入储备金池，使储备的利用率接近 0，然后调用 `swapBorrowRateMode()` 执行浮动利率到借款利率的切换，并享受最低的 `currentStableBorrowRate`（由于当前储备的利用率接近 0），最后退出来归还闪电贷。类似的方法也可以应用在 `validateBorrow()` 中绕过 `maxStableLoanPercent` 的执行。

推荐方法 修改当前的 `swapBorrowRateMode()` 的执行逻辑，以防御性地检测储备金使用率的突然变化，并阻止恶意尝试。

状态 这个问题已经被开发团队确认。需要注意的是，Aave 所使用的 IR 部分缓解了这一问题，稳定利率借款的斜率与可变的相比要平坦得多，因此在破发点和准备金完全清空的情况

下，稳定利率借款的差别并不大。需要注意的是，这种潜在的滥用现象在 V1 中已经存在，但并没有借款人实际使用的迹象。

3.22 被绕过的 LIQUIDATION_CLOSE_FACTOR_PERCENT 限制

- ID: PVE-022
- 严重性: 中危
- 可能性: 中
- 影响力: 中
- 目标: LendingPoolCollateralManager
- 类别: Business Logic [11]
- CWE 子类: CWE-837 [7]

描述

Aave V2 定义了许多系统范围的风险参数。在 3.19 节中，我们讨论了一个名为 HEALTH_FACTOR_ABOVE_THRESHOLD 的风险参数，它指定了允许的健康因子的阈值。在本节中，我们将研究另一个风险参数，即 LIQUIDATION_CLOSE_FACTOR_PERCENT。这个风险参数适用于债务头寸被清算时，用于限制特定 liquidationCall() 的可清算本金。

```
139   function liquidationCall(
140     address collateral,
141     address principal,
142     address user,
143     uint256 purchaseAmount,
144     bool receiveAToken
145   ) external returns (uint256, string memory) {
146     ReserveLogic.ReserveData storage collateralReserve = _reserves[collateral];
147     ReserveLogic.ReserveData storage principalReserve = _reserves[principal];
148     UserConfiguration.Map storage userConfig = _usersConfig[user];
149 
150     LiquidationCallLocalVars memory vars;
151 
152     (, , , , vars.healthFactor) = GenericLogic.calculateUserAccountData(
153       user,
154       _reserves,
155       _usersConfig[user],
156       _reservesList,
157       _addressesProvider.getPriceOracle()
158     );
159 
160     //if the user hasn't borrowed the specific currency defined by asset, it cannot be
          liquidated
161     (vars.userStableDebt, vars.userVariableDebt) = Helpers.getUserCurrentDebt(
162       user,
163       principalReserve
164     );
```

```
165
166     (vars.errorCode, vars.errorMsg) = ValidationLogic.validateLiquidationCall(
167       collateralReserve,
168       principalReserve,
169       userConfig,
170       vars.healthFactor,
171       vars.userStableDebt,
172       vars.userVariableDebt
173     );
174
175     if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors
          .NO_ERROR) {
176       return (vars.errorCode, vars.errorMsg);
177     }
178
179     vars.collateralAtoken = IAToken(collateralReserve.aTokenAddress);
180
181     vars.userCollateralBalance = vars.collateralAtoken.balanceOf(user);
182
183     vars.maxPrincipalAmountToLiquidate = vars.userStableDebt.add(vars.userVariableDebt).
          percentMul(
184       LIQUIDATION_CLOSE_FACTOR_PERCENT
185     );
186
187     vars.actualAmountToLiquidate = purchaseAmount > vars.maxPrincipalAmountToLiquidate
```

Listing 3.32: LendingPoolCollateralManager.sol

具体来说，基于上述 `liquidationCall()` 的代码片段，最大可清算本金计算为 `(userStableDebt+userVariableDebt)*(LIQUIDATION_CLOSE_FACTOR_PERCENT)`（第 183 — 185 行）。然而，我们也注意到另一个替代函数，即 `repayWithCollateral()`，它同样可以用来清算违约债务头寸，但并没有执行这个风险参数。这就造成了其执行上的不一致。

推荐方法 强制使用 `LIQUIDATION_CLOSE_FACTOR_PERCENT` 这个系统级的风险参数来限制可清算本金。

状态 这个问题已经被开发团队确认并且在 86 号 merge 中被修复。

4 结论

在本次审计中，我们对 Aave V2 的设计和实现进行了分析。该系统提出了一个独特且稳健的去中心化非托管货币市场协议，用户可以作为存款人或借款人参与。Aave V2 改进了早期版本，提供了更多的创新功能，例如，债务代币化、抵押品交易和新的资金贷款。目前的代码库结构良好，组织整齐。那些被发现的问题已经被及时地解决和修正。

作为最后的预防措施，我们需要强调的是，智能合约作为一个整体仍处于早期但令人兴奋的发展阶段。为了改进本报告，我们非常感谢任何建设性的反馈或建议，包括对我们的方法、审计结果或范围/覆盖面的潜在差异。

5 附录

5.1 基础编码漏洞

5.1.1 构造器不匹配

- 描述: 合约名称和构造函数名称是否相同
- 结果: 未发现
- 严重程度: 严重

5.1.2 夺权

- 描述: 设置 owner 权限函数是否被保护
- 结果: 未发现
- 严重程度: 严重

5.1.3 多余的回退函数

- 描述: 合约是否有多余的回退函数
- 结果: 未发现
- 严重程度: 严重

5.1.4 上溢/下溢

- 描述: 合约是否有向上溢出或向下溢出漏洞 [15, 16, 17, 18, 19]
- 结果: 未发现
- 严重程度: 严重

5.1.5 重入

- 描述: 重入漏洞 [20] 是指代码可以再次调用该合约并修改状态
- 结果: 未发现
- 严重程度: 严重

5.1.6 代币赠予

- 描述: 合约是否给任意地址返还代币
- 结果: 未发现
- 严重程度: 高

5.1.7 黑洞地址

- 描述: 合约是否永久保留所有的代币，即只转进不转出
- 结果: 未发现
- 严重程度: 高

5.1.8 未被授权的合约销毁

- 描述: 合约是否可以被任意地址销毁
- 结果: 未发现
- 严重程度: 中

5.1.9 回滚型拒绝服务攻击

- 描述: 合约是否会受到非预期的回滚造成的拒绝服务攻击影响
- 结果: 未发现
- 严重程度: 中

5.1.10 未被校验的外部调用

- 描述: 合约是否有未检查返回值的外部调用
- 结果: 未发现
- 严重程度: 中

5.1.11 Gas 不足的 send 调用

- 描述: 合约是否收到 gas 不足的 send 调用的影响
- 结果: 未发现
- 严重程度: 中

5.1.12 使用 send 而非 transfer

- 描述: 合约是否使用了 send 而非 transfer
- 结果: 未发现
- 严重程度: 中

5.1.13 开销过大的循环体

- 描述: 合约是否有一个开销过大的循环体，这会造成 Out-Of-Gas 异常
- 结果: 未发现
- 严重程度: 中

5.1.14 使用不受信的库

- 描述: 合约是否使用了可疑的库
- 结果: 未发现
- 严重程度: 中

5.1.15 使用可被预测的变量

- 描述: 合约是否包含了随机变量，但是该变量的值可以被预测
- 结果: 未发现
- 严重程度: 中

5.1.16 交易顺序依赖

- 描述: 合约的最终状态是否取决于交易的顺序
- 结果: 未发现
- 严重程度: 中

PeckShield

5.1.17 使用被废弃的变量

- 描述: 合约是否使用了被废弃的 tx.origin 来实现鉴权
- 结果: 未发现
- 严重程度: 中

5.2 语义一致性检查

- 描述: 白皮书的语义和合约的实现是否一致
- 结果: 未发现
- 严重程度: 严重

5.3 额外建议

5.3.1 避免使用可变字节序列

- 描述: 使用固定长度的字节序列而非 byte[],因为这样做会浪费空间
- 结果: 未发现
- 严重程度: 低

5.3.2 明确可见层级

- 描述: 给函数和状态变量分配清晰的可见层级
- 结果: 未发现
- 严重程度: 低

5.3.3 明晰类型推断

- 描述: 不要使用关键词 var 来定义类型,即不要让编译器来猜测可能的类型
- 结果: 未发现
- 严重程度: 低

PeckShield

5.3.4 严格遵守函数声明

- 描述: Solidity 编译器（版本号 0.4.23）强制要求 `calls()` [1] 返回值的 ABI 长度检查。这可能会中断执行如果函数的实现不遵循它的声明
- 结果: 未发现
- 严重程度: 低

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.

PeckShield

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

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