SMART CONTRACT AUDIT REPORT

for

Aave V3.0.1

Prepared By: Xiaomi Huang

PeckShield

December 22, 2022

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	Aave V3.0.1
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 22, 2022	Stephen Bie	Final Release
1.0-rc	December 6, 2022	Stephen Bie	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Aave V3.0.1` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Aave V3.0.1

`Aave` is a popular decentralized non-custodial liquidity 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 V3` protocol includes new features for new usage scenarios and improves earlier versions on capital efficiency, security, decentralization, UX while at the same time providing new functionalities to leverage the capabilities of rollups and the growing ecosystem of competing L1s. The audited `Aave V3.0.1` adds a variety of improvements and new minor features that the community had identified valuable for the protocol (Please refer to `Issue Comments Link` for details).

Table 1.1: Basic Information of Aave V3.0.1

Item	Description
Name	Aave
Website	https://aave.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 22, 2022

In the following, we show the specific pull request and the commit hash value used in this audit.

- https://github.com/aave/aave-v3-core/pull/701 (f8825f8)

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

- https://github.com/aave/aave-v3-core/pull/701 (428e258)

1.2 About PeckShield

PeckShield Inc. [5] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [4]:

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

Table 1.3: The Full Audit Checklist

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

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [3], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Aave V3.0.1` smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	1	■
Informational	1	■
Total	2	

We have previously audited the main `Aave V3` protocol. In this report, we exclusively focus on the specific pull request `PR701`. We examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions 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 issue(s) (shown in Table 2.1), including 1 low-severity vulnerability and 1 informational recommendation.

Table 2.1: Key Aave V3.0.1 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Logic of LiquidationLogic::executeLiquidationCall()	Coding Practices	Fixed
PVE-002	Low	Improved Event Generation in ScaledBalanceTokenBase::_transfer()	Coding Practices	Fixed

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

3 | Detailed Results

3.1 Improved Logic of LiquidationLogic::executeLiquidationCall()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LiquidationLogic`
- Category: Coding Practices [2]
- CWE subcategory: CWE-628 [1]

Description

In the `Aave V3` protocol, the `LiquidationLogic` library implements the liquidation-related logic. In particular, one entry routine, i.e., `executeLiquidationCall()`, is designed to liquidate a position if its `Health Factor` drops below `1`. The caller (i.e., `liquidator`) can gain the borrower's collateral assets plus a bonus to cover the market risk via repaying the borrowed assets on behalf of the borrower. While examining its logic, we observe the current implementation can be improved.

To elaborate, we show below the related code snippet of the `executeLiquidationCall()` routine. By design, if all the collateral assets of the borrower are liquidated, the borrower will not use this kind of assets as collateral anymore. Inside the `executeLiquidationCall()` routine, there are two implementations that meet the requirement (lines 106 and 133). The former does not consider the liquidation protocol fee (i.e., `vars.liquidationProtocolFeeAmount`) while the latter does. If we assume the `vars.liquidationProtocolFeeAmount` variable is `0` and the `vars.actualCollateralToLiquidate` variable is equal to `vars.userCollateralBalance`, the statement of `userConfig.setUsingAsCollateral(collateralReserve.id, false)` will be executed twice. We believe the latter is necessary only when the `vars.actualCollateralToLiquidate` variable is not `0`.

```
96      function executeLiquidationCall(
97          mapping(address => DataTypes.ReserveData) storage reservesData,
98          mapping(uint256 => address) storage reservesList,
99          mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
```

```
        mapping(uint8 => DataTypes.EModeCategory) storage eModeCategories,
        DataTypes.ExecuteLiquidationCallParams memory params
    ) external {
        ...
        // If the collateral being liquidated is equal to the user balance,
        // we set the currency as not being used as collateral anymore
        if (vars.actualCollateralToLiquidate == vars.userCollateralBalance) {
            userConfig.setUsingAsCollateral(collateralReserve.id, false);
            emit ReserveUsedAsCollateralDisabled(params.collateralAsset, params.user);
        }

        ...

        // Transfer fee to treasury if it is non-zero
        if (vars.liquidationProtocolFeeAmount != 0) {
            uint256 liquidityIndex = collateralReserve.getNormalizedIncome();
            uint256 scaledDownLiquidationProtocolFee = vars.liquidationProtocolFeeAmount.rayDiv(
                liquidityIndex
            );
            uint256 scaledDownUserBalance = vars.collateralAToken.scaledBalanceOf(params.user);
            // To avoid trying to send more aTokens than available on balance, due to 1 wei imprecision
            if (scaledDownLiquidationProtocolFee > scaledDownUserBalance) {
                vars.liquidationProtocolFeeAmount = scaledDownUserBalance.rayMul(liquidityIndex);
            }
            vars.collateralAToken.transferOnLiquidation(
                params.user,
                vars.collateralAToken.RESERVE_TREASURY_ADDRESS(),
                vars.liquidationProtocolFeeAmount
            );
        }

        // If the collateral being liquidated is equal to the user balance,
        // we set the currency as not being used as collateral anymore
        if (vars.actualCollateralToLiquidate + vars.liquidationProtocolFeeAmount == vars.userCollateralBalance) {
            userConfig.setUsingAsCollateral(collateralReserve.id, false);
            emit ReserveUsedAsCollateralDisabled(params.collateralAsset, params.user);
        }
        ...
    }
```

Listing 3.1: `LiquidationLogic::executeLiquidationCall()`

Recommendation Improve the implementation of the `executeLiquidationCall()` routine as above-mentioned for gas optimization.

Status The issue has been addressed by the following commit: `56bcf5d`.

3.2 Improved Event Generation in ScaledBalanceTokenBase::_transfer()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ScaledBalanceTokenBase`
- Category: Coding Practices [2]
- CWE subcategory: CWE-628 [1]

Description

In the `Aave V3` protocol, the balance of the depositor's `AToken` is constantly increasing as the interest accrues. To accommodate the ever-changing indexes in the lending pool, the `AToken` should internally keep the scaled balance. The `ScaledBalanceTokenBase` contract is designed to meet the requirement. In particular, one entry routine, i.e., `_transfer()`, is designed to transfer the scaled balance between the `sender` and the `recipient`. While examining its logic, we observe its current implementation can be improved.

To elaborate, we show below the related code snippet of the `ScaledBalanceTokenBase()` contract. As mentioned above, the balance of the user's `AToken` is constantly increasing as the interest accrues. Inside the `_transfer()` routine, the increased balances of the `sender` (line 145) and the `recipient` (line 149) are calculated separately, while the corresponding `Transfer` and `Mint` events are emitted according to the ERC20 specification. However, we observe there is a corner case (i.e., `sender == recipient`) where the `Transfer` and `Mint` events are emitted repeatedly. Given this, it's better to handle the corner case to avoid emitting the same events twice.

```
138     function _transfer(
139         address sender,
140         address recipient,
141         uint256 amount,
142         uint256 index
143     ) internal {
144         uint256 senderScaledBalance = super.balanceOf(sender);
145         uint256 senderBalanceIncrease = senderScaledBalance.rayMul(index) -
146         senderScaledBalance.rayMul(_userState[sender].additionalData);
147 
148         uint256 recipientScaledBalance = super.balanceOf(recipient);
149         uint256 recipientBalanceIncrease = recipientScaledBalance.rayMul(index) -
150         recipientScaledBalance.rayMul(_userState[recipient].additionalData);
151 
152         _userState[sender].additionalData = index.toUint128();
153         _userState[recipient].additionalData = index.toUint128();
154 
155         super._transfer(sender, recipient, amount.rayDiv(index).toUint128());
156 
```

```
        if (senderBalanceIncrease > 0) {
            emit Transfer(address(0), sender, senderBalanceIncrease);
            emit Mint(_msgSender(), sender, senderBalanceIncrease, senderBalanceIncrease, index);
        }

        if (recipientBalanceIncrease > 0) {
            emit Transfer(address(0), recipient, recipientBalanceIncrease);
            emit Mint(_msgSender(), recipient, recipientBalanceIncrease, recipientBalanceIncrease, index);
        }

        emit Transfer(sender, recipient, amount);
    }
```

Listing 3.2: `ScaledBalanceTokenBase::_transfer()`

Recommendation Accommodate the corner case to avoid emitting the same events repeatedly.

Status The issue has been addressed by the following commit: `4449676`.

4 Conclusion

In this audit, we have analyzed the Aave V3.0.1 implementation, which adds a variety of improvements and new minor features that the community had identified valuable for Aave V3 (Please refer to Issue Comments Link for details). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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