SMART CONTRACT AUDIT REPORT

for

AAVEV2 LIGHT DEPLOYMENT

Prepared By: Shuxiao Wang

PeckShield

March 16, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	March 16, 2021	Xuxian Jiang	Final Release
1.0-rc2	February 18, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	February 11, 2021	Xuxian Jiang	Release Candidate #1
0.2	February 9, 2021	Xuxian Jiang	Additional Findings
0.1	February 7, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction 4

1.1 About Light Deployment 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings 9

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results 11

3.1 External Declaration of Only-Externally-Invoked Functions 11

3.2 Storage Reservation For Further Upgrades 12

4 Conclusion 14

References 15

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of `Aave`'s functionality to allow for `light deployment`, 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 versions of smart contracts are well designed and engineered. This document outlines our audit results.

1.1 About Light Deployment

`Aave` is a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. This audit covers the new functionality to allow for `light deployment` of a new market with reduced deployment cost. This is necessary since the current deployment of a new market of the `Aave` protocol is rather burdensome and has a high deployment cost. Moreover, the update of the logic of one specific component at the same time across all markets is not possible, which creates big friction on versioning of implementations (behind proxies).

Table 1.1: Basic Information of Aave's Light Deployment

Item	Description
Issuer	Aave
Website	http://aave.com/
Audit Modules	Light Deployment
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 16, 2021

The `light deployment` functionality allocates necessary storage variables on all affected contracts and remove market or token specific dependencies. The basic information of this new functionality is shown above. In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/aave/protocol-v2/tree/feat/light-deployments (12cda09)

1.2 About PeckShield

PeckShield Inc. [6] 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

PeckShield

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

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

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

Table 1.3: The Full List of Check Items

Category	Check Item
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

To evaluate the risk, we go through a list of check 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) [4], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.4: 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 Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the Aave's governance subsystem and its new token contract. 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 logics, 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	0	
Informational	2	🟦🟦
Total	2	

PeckShield

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issue(s) as shown in Table 2.1, including 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category
PVE-001	Informational	External Declaration of Only-Externally-Invoked Functions	Coding Practices
PVE-002	Informational	Storage Reservation For Further Upgrades	Coding Practices

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

PeckShield

3 | Detailed Results

3.1 External Declaration of Only-Externally-Invoked Functions

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LendingPoolConfigurator
- Category: Coding Practices [3]
- CWE subcategory: CWE-287 [1]

Description

The Light Deployment support makes a number of changes, including the introduction of necessary interface functions that are designed to be called only for external users. Some of them are currently defined as public. In public functions, Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Note that memory allocation can be expensive, whereas reading from calldata is not. So when these functions are not used within the contract, it's always suggested to define them as external instead of public.

In the following, we show one example function that can be canonically changed from public to external (shown below). Note that this function takes an argument inputParams and the above performance optimization has been applied.

```
/**
 * @dev Initializes reserves in batch
 **/
function batchInitReserve(InitReserveInput[] calldata inputParams) public
    onlyPoolAdmin {
  ILendingPool cachedPool = pool;
  for (uint256 i = 0; i < inputParams.length; i++) {
    _initReserve(cachedPool, inputParams[i]);
  }
}
```

Listing 3.1: LendingPoolConfigurator::batchInitReserve()

Recommendation This is really optional: revise the aforementioned function from being public to external.

3.2 Storage Reservation For Further Upgrades

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [2]

Description

The various token implementations in AaveV2 take a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. This approach has the flexible support in terms of upgradeability. However, the upgradeability support comes with a few caveats. One important caveat is related to the storage layout of new contracts that are just deployed to replace old contracts.

Due to the inherent requirement of any proxy-based upgradeability system, the storage layout of new contracts should not overwrite old contracts. This means we should only append new states after those defined states in old contracts. It is important to realize that for contracts that use inheritance, the ordering of state variables is determined by the so-called C3 superclass linearization order of contracts starting with the most base-ward contract. Note that the C3 superclass linearization is an algorithm used primarily to obtain the order in which methods should be inherited in the presence of multiple inheritance.

In order to accommodate future upgrades, it is commonly suggested to reserve certain storage slots in the base contracts. These reserved storage slots can flexibly support future layout changes, including the additions of new states. The AaveV2's functionality for light deployment requires the additions of several new states that were previously defined as immutable or constant. As an example, the AToken contract has been enhanced with new states _pool, _treasury, _underlyingAsset, and _incentivesController and their respective getters/setters to support the deployment with reduced cost.

```
contract AToken is
  VersionedInitializable,
  IncentivizedERC20('ATOKEN_IMPL', 'ATOKEN_IMPL', 0),
  IAToken
{
  using WadRayMath for uint256;
  using SafeERC20 for IERC20;

bytes public constant EIP712_REVISION = bytes('1');
bytes32 internal constant EIP712_DOMAIN =
  keccak256('EIP712Domain(string name,string version,uint256 chainId,address
      verifyingContract)');
bytes32 public constant PERMIT_TYPEHASH =
  keccak256('Permit(address owner,address spender,uint256 value,uint256 nonce,uint256
      deadline)');

uint256 public constant ATOKEN_REVISION = 0x1;

/// @dev owner => next valid nonce to submit with permit()
mapping(address => uint256) public _nonces;

bytes32 public DOMAIN_SEPARATOR;

ILendingPool internal _pool;
address internal _treasury;
address internal _underlyingAsset;
IAaveIncentivesController internal _incentivesController;
...
}
```

Listing 3.2: New States in AToken

With that, we suggest to add the reservation of storage slots, i.e., `uint256[50] private ______gap;` at the end especially in related base constract, e.g., `IncentivizedERC20` for future upgrades.

Recommendation Reserve additional storage slots in base contracts for future upgrades.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AaveV2` protocol's functionality to allow for `light deployment`. The new functionality greatly reduce the deployment cost of a new market. The current code base is well organized and those identified issues are promptly confirmed and addressed.

PeckShield

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

PeckShield