SMART CONTRACT AUDIT REPORT

for

AMPL INTEGRATION IN AAVE

Prepared By: Shuxiao Wang

PeckShield

March 11, 2021

Document Properties

Client	AMPL Integration in Aave
Title	Smart Contract Audit Report
Target	aAMPL
Version	1.0
Author	Xuxian Jiang
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 11, 2021	Xuxian Jiang	Final Release
1.0-rc	February 27, 2021	Xuxian Jiang	Release Candidate
0.3	February 23, 2021	Xuxian Jiang	Additional Findings #2
0.2	February 20, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 19, 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

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

1.1 About Aave And Ampleforth

Aave is a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Ampleforth is an elastic supply protocol that automatically increases or decreases the number of protocol tokens AMPL in user wallets based on price. The audited integration of the AMPL token (aAMPL) for the Aave protocol aims to support the lending and borrowing of AMPL, which behaves differently from fixed supply tokens. In particular, the desired behavior is for the borrowed token count to stay the same and not get affected by rebase. And the unborrowed amount will get affected by rebasing while deposited in the Aave protocol.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of aAMPL

Item	Description
Issuer	AMPL Integration in Aave
Website	https://www.ampleforth.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 11, 2021

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

- https://github.com/ampleforth/protocol-v2.git (8a4e8f6)

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

- https://github.com/ampleforth/protocol-v2.git (7daea56f)

1.2 About PeckShield

PeckShield Inc. [8] 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			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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) [6], 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 1.3: 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

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.

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 `AMPL` integration in `AaveV2`. 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	1	🟦
Medium	0	
Low	2	🟦 🟦
Informational	0	
Total	3	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Diverged AToken Inheritance	Coding Practices	Fixed
PVE-002	High	Proper _totalScaledAMPLDeposited Accounting	Numeric Errors	Fixed
PVE-003	Low	Forward-Compatibility of AToken Upgrades	Coding Practices	Mitigated

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

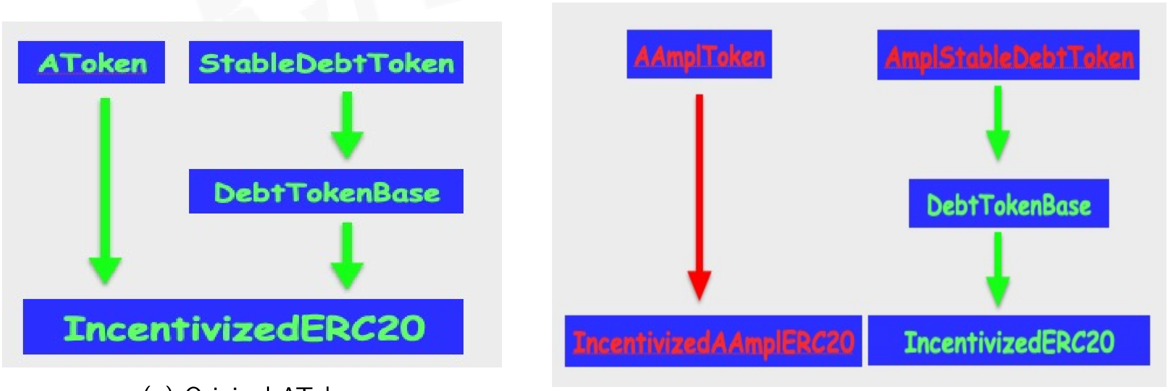
3 Detailed Results

3.1 Diverged AToken Inheritance

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [3]

Description

In `Aave`, `ATokens` are interest-bearing tokens that are minted upon deposit and burned when redeemed. By design, `ATokens` are pegged $1 : 1$ to the value of the underlying asset that is deposited in the `Aave` protocol. Once minted, `ATokens` can be freely stored, transferred, and traded. Currently, only fixed or non-rebasing tokens are supported as the underlying asset, which is loaned out to borrowers while `ATokens` accrue interest in real time.



(a) Original AToken

(b) Current aAMPL

Figure 3.1: Different Inheritance Hierarchy

The integration of `AMPL` is greatly different from the support of current underlying tokens. The reason is that AMPL is of elastic supply in nature and the balance can be dynamically increased or decreased based on current market price. In order to support AMPL as the underlying asset in ATokens, the integration needs to meet the following expectation: during the AMPL-inherent rebasing operation, the borrowed token amount stays the same while the unborrowed amount will be accordingly affected even when they are simply held in Aave. In Figure 3.1a, we show the current AToken contract inheritance hierarchy.

It comes to our attention that the proposed integration of aAMPL takes a different inheritance hierarchy as demonstrated in Figure 3.1b. The diverged inheritance hierarchy not only causes difficulty in current integration and understanding, but also brings additional overhead for maintenance and future upgrades.

```
contract AAmplToken is VersionedInitializable, IncentivizedAAmplERC20, IAToken {
  using WadRayMath for uint256;
  using SafeERC20 for IERC20;

  bytes public constant EIP712_REVIS
  ...
}
```

Listing 3.1: AAmplToken.sol

In Figure 3.1b, we show the inheritance hierarchy related to AAmplToken and AmplStableDebtToken. The same issue, however, is also applicable to AmplVariableDebtToken.

Recommendation Suggested to localize changes and maintain the same inheritance hierarchy of aAMPL with current other AToken implementation.

Status The issue has been fixed in this commit: 13c6969.

3.2 Proper _totalScaledAMPLDeposited Accounting

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High

- Target: IncentivizedAAmplERC20
- Category: Numeric Errors [5]
- CWE subcategory: CWE-190 [2]

Description

As mentioned in Section 3.1, the integration of AMPL in Aave requires proper handling of AMPL-inherent rebasing operation. Specifically, during rebasing, the borrowed token amount needs to stay the same while the unborrowed amount will be accordingly affected even when they are simply held in Aave. With that, there is a need to properly keep track of the unborrowed amount.

The unborrowed amount can be calculated by subtracting the borrowed amount from the total deposited amount. The borrowed amount stays the same on rebase while the total deposited amount will be affected by the rebase. To elaborate, we show below the `totalSupply()` routine in `IncentivizedAAmplERC20`.

```
61   /**
62    * @return The total supply of the token
63    *
64    *  totalSupply() = unborrowed + borrowed
65    *                = (totalScaledAMPLDeposited - totalScaledAMPLBorrowed)/AMPL_SCALAR +
         totalAMPLBorrowed
66    **/
67   function totalSupply() public view virtual override returns (uint256) {
68     uint256 totalAMPLBorrowed;
69     uint256 totalScaledAMPLBorrowed;
70     (totalAMPLBorrowed, totalScaledAMPLBorrowed) = getAMPLBorrowData();
71     return _totalScaledAMPLDeposited.sub(totalScaledAMPLBorrowed).div(getAMPLScalar()).
         add(totalAMPLBorrowed);
72   }
```

Listing 3.2: IncentivizedAAmplERC20::totalSupply()

As shown from the above computation (line 71), there is an internal state `_totalScaledAMPLDeposited` to keep track of current deposited amount. If we further examine the `IncentivizedAAmplERC20::_mint()/_burn()` operations, this specific state is accordingly updated (lines 254 and 290).

```
230  function _mint(address account, uint256 amount) internal virtual {
231    require(account != address(0), 'AAmplERC20: mint to the zero address');
232
233    _beforeTokenTransfer(address(0), account, amount);
234
235    uint256 oldTotalSupply = totalSupply();
236    uint256 oldAccountBalance = balanceOf(account);
237    uint256 oldRemainingAccountBalance = oldTotalSupply.sub(oldAccountBalance);
238
239    uint256 _oldAccountBalance = _balances[account];
240    uint256 _oldTotalSupply = _totalSupply;
241
242    uint256 newTotalSupply = oldTotalSupply.add(amount);
243    uint256 newAccountBalance = oldAccountBalance.add(amount);
244
245    uint256 _mintAmount = _oldTotalSupply
246      .mul(newAccountBalance)
247      .sub(_oldAccountBalance.mul(newTotalSupply))
248      .div(oldRemainingAccountBalance);
249
250    _totalSupply = _totalSupply.add(_mintAmount);
251    _balances[account] = _oldAccountBalance.add(_mintAmount);
252
253    // NOTE: this additional book keeping to keep track of 'unborrowed' AMPLs
254    _totalScaledAMPLDeposited = _totalScaledAMPLDeposited.add(amount.mul(getAMPLScalar()
         ));
```

```

    if (address(_incentivesController) != address(0)) {
      _incentivesController.handleAction(account, oldTotalSupply, oldAccountBalance);
    }
  }
```

Listing 3.3: IncentivizedAAmplERC20::_mint()

A close examination shows the update of `_totalScaledAMPLDeposited` is increased or decreased by `amount.mul(getAMPLScalar())`, where `amount` is the deposited/withdrawn amount and `getAMPLScalar()` normalizes the amount. However, due to the need of efficiently computing accrued interests, the amount passed to `IncentivizedAAmplERC20::_mint()` is already scaled – as shown in the following entry function `AAmplToken::mint()`.

```
  function mint(
    address user,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool returns (bool) {

    uint256 previousBalance = super.balanceOf(user);

    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);
    _mint(user, amountScaled);

    emit Transfer(address(0), user, amount);
    emit Mint(user, amount, index);

    return previousBalance == 0;
  }
```

Listing 3.4: AAmplToken::mint()

As a result, the current book-keeping of deposited amount is inaccurate. For correction, there is a need to use the actual deposited amount, instead of a scaled one!

Recommendation Use the real deposited number to compute the unborrowed amount.

Status This issue has been fixed in this commit: `f7f609d`.

3.3 Forward-Compatibility of AToken Upgrades

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, the `ATokens` are interest-bearing tokens that are minted upon deposit and burned when redeemed. The implementation may evolve with new additions of various features or certain customizations of existing functionality. For example, there is a recent need to add a new functionality that allows for `light deployment` of a new market with reduced deployment cost. This is proposed with the observation that 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).

The current integration of `AMPL` tokens changes the storage layout of affected contracts. To elaborate, we show below the `IncentivizedAAmplERC20` contract, the `AMPL` integration adds a new state `_totalScaledAMPLDeposited` inside existing layout. While it poses no issue to current support, it may bring inconsistency or even incompatibility for future upgrades. Using the above-mentioned `light deployment` functionality as an example, it requires the additions of several new states that were previously defined as `immutable` or `constant`. Specifically, 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.

```
// TODO: optimize external calls for AMPL scaling factor and reads
contract IncentivizedAAmplERC20 is Context, IERC20, IERC20Detailed {
  using SafeMath for uint256;

  IAaveIncentivesController internal immutable _incentivesController;

  mapping(address => uint256) private _balances;
  mapping(address => mapping(address => uint256)) private _allowances;

  uint256 internal _totalScaledAMPLDeposited;
  uint256 internal _totalSupply;

  string private _name;
  string private _symbol;
  uint8 private _decimals;
  ...

}
```

Listing 3.5: IncentivizedAAmplERC20.sol

The current integration of `AMPL` and official support for `light deployment` work on the same code base, but diverges on making their own state layout changes or modifications, which could be hard to converge in the future. From maintenance perspective, it is always better to make the code base and state layout forward-compatible.

Recommendation It is suggested to make the changes, including the introduction of new states or changes of current state layout, to be forward-compatible.

Status This issue has been mitigated by appending `AMPL`-specific states in the following commit: `f7f609d`.

4 Conclusion

In this audit, we have analyzed the aAMPL design and implementation, which aims to seamlessly integrate the AMPL token for the Aave protocol The system presents a unique enhancement to current Aave lending platform. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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