SMART CONTRACT AUDIT REPORT

for

Aave Starknet Bridge

Prepared By: Xiaomi Huang

PeckShield

October 7, 2022

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	Aave Starknet Bridge
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 7, 2022	Xuxian Jiang	Final Release
1.0-rc	June 14, 2022	Luck Hu	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 **4**
1.1 Aave Starknet Bridge 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 Redundant rayToWad() Removal 11
3.2 Accommodation Of Non-ERC20-Compliant Tokens 12
3.3 Incompatibility with Deflationary/Rebasing Tokens 14

4 Conclusion **16**

References **17**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Aave Starknet Bridge`, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 Aave Starknet Bridge

The `Aave Starknet Bridge` is an integration of `Aave <> Starknet` which allows users deposit/withdrawal on `Aave Ethereum` by exclusively transacting on `Starknet`. The L1 `Bridge` contract is the `Ethereum` counterpart of the `Aave` reward collection mechanism. It has three purposes: (i) allow users deposit and withdraw of `aTokens`, (ii) update the state of the corresponding `staticATokens` on `Starknet`, and (iii) let users retrieve their rewards on `Ethereum`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Aave Starknet Bridge

Item	Description
Name	Aave
Website	https://aave.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 7, 2022

In the following, we show the Git repositories of reviewed file and the commit hash value used in this audit. Note this audit only covers the `l1/Bridge.sol` contract.

- https://github.com/aave-starknet-project/aave-starknet-bridge.git (e699360)

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

- https://github.com/aave-starknet-project/aave-starknet-bridge.git (517cd7a)

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 \ 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 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 classified into four categories accordingly, 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

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

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.

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Aave Starknet Bridge` 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	1	■
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. 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, the smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Aave Starknet Bridge Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Redundant rayToWad() Removal	Coding Practices	Fixed
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	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 Redundant rayToWad() Removal

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Bridge`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

The `Bridge` contract provides a `_computeRewardsDiff()` routine to check the difference between the L1/L2 rewards index and return the amount of unclaimed rewards. While examining the logic to compute the rewards amount, we notice the redundant invoking of `rayToWad()` which should be removed.

To elaborate, we show below the code snippet of the `_computeRewardsDiff()` routine. It accepts three input parameters: the first parameter `amount` indicates the amount of withdrawal; the second parameter `l2RewardsIndex` indicates the rewards index from L2; while the last parameter `l1RewardsIndex` indicates the current rewards index in L1. This routine returns the amount of unclaimed rewards as `(rayAmount.rayMulNoRounding(l1RewardsIndex - l2RewardsIndex)).rayToWad()` (line 396). However, we notice that the `rayAmount` is in `RAY` but the `l2RewardsIndex` and `l1RewardsIndex` are in `WAD`. As a result, the `rayMulNoRounding()` will return a value in `WAD`, not in `RAY`. So there's no need to invoke `rayToWad()`.

```
389    function _computeRewardsDiff(
390        uint256 amount,
391        uint256 l2RewardsIndex,
392        uint256 l1RewardsIndex
393    ) internal pure returns (uint256) {
394        uint256 rayAmount = amount.wadToRay();
395        return
396            (rayAmount.rayMulNoRounding(l1RewardsIndex - l2RewardsIndex))
397                .rayToWad();
```

```
    }
```

Listing 3.1: Bridge :: _computeRewardsDiff()

Recommendation Revisit the above mentioned routine to remove the redundant `rayToWad()` invoking.

Status The issue has been fixed by this commit: `6fdc8ef`.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bridge`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
```

```
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: `ZRX.sol`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `Bridge::deposit()` routine. If the `ZRX` token is supported as `underlyingAsset`, the unsafe version of `underlyingAsset.transferFrom(msg.sender, address(this), amount)` (line 90) may return false on failure while not revert. We may intend to replace the `transferFrom()` with `safeTransferFrom()`.

```
73      function deposit(
74          address l1AToken,
75          uint256 l2Recipient,
76          uint256 amount,
77          uint16 referralCode,
78          bool fromUnderlyingAsset
79      ) external override onlyValidL2Address(l2Recipient) returns (uint256) {
80          IERC20 underlyingAsset = _aTokenData[l1AToken].underlyingAsset;
81          ILendingPool lendingPool = _aTokenData[l1AToken].lendingPool;
82          require(
83              underlyingAsset != IERC20(address(0)),
84              Errors.B_ATOKEN_NOT_APPROVED
85          );
86          require(amount > 0, Errors.B_INSUFFICIENT_AMOUNT);
87          // deposit aToken or underlying asset
88
89          if (fromUnderlyingAsset) {
90              underlyingAsset.transferFrom(msg.sender, address(this), amount);
91              lendingPool.deposit(
92                  address(underlyingAsset),
93                  amount,
94                  address(this),
95                  referralCode
96              );
97          } else {
98              IERC20(l1AToken).transferFrom(msg.sender, address(this), amount);
99          }
100         ...
101     }
```

Listing 3.3: `Bridge::deposit()`

Note another routine, i.e., `Bridge::_transferRewards()`, can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been fixed by this commit: `6fdc8ef`.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bridge`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Bridge` contract provide one entry routine, i.e., `deposit()`, via which users can either bridge their `aToken` or deposit the underlying asset. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the code snippet of the `deposit()` routine.

```
    function deposit(
        address l1AToken,
        uint256 l2Recipient,
        uint256 amount,
        uint16 referralCode,
        bool fromUnderlyingAsset
    ) external override onlyValidL2Address(l2Recipient) returns (uint256) {
        IERC20 underlyingAsset = _aTokenData[l1AToken].underlyingAsset;
        ILendingPool lendingPool = _aTokenData[l1AToken].lendingPool;
        require(
            underlyingAsset != IERC20(address(0)),
            Errors.B_ATOKEN_NOT_APPROVED
        );
        require(amount > 0, Errors.B_INSUFFICIENT_AMOUNT);
        // deposit aToken or underlying asset

        if (fromUnderlyingAsset) {
            underlyingAsset.transferFrom(msg.sender, address(this), amount);
            lendingPool.deposit(
                address(underlyingAsset),
                amount,
                address(this),
```

```
                referralCode
            );
        } else {
            IERC20(l1AToken).transferFrom(msg.sender, address(this), amount);
        }
        ...
    }
```

Listing 3.4: `Bridge::deposit()`

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

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

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

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

Status This issue has been mitigated as the team confirmed that only `aTokens` are used which are assumed to be normal `ERC20`. And the team also added the assumption to the README in commit `6fdc8ef`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the L1 `Bridge` contact in the `Aave Starknet Bridge`, which is the `Ethereum` counterpart of the `Aave` reward collection mechanism. It has three purposes: (i) let users deposit and withdraw of `aTokens`, (ii) update the state of the corresponding `staticATokens` on `Starknet`, and (iii) let users retrieve their rewards on `Ethereum`. The current code base is well organized and those identified issues are promptly confirmed and addressed.

Meanwhile, 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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

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