SMART CONTRACT AUDIT REPORT

for

AIA Bridge

Prepared By: Xiaomi Huang

PeckShield

December 28, 2023

Document Properties

Client	AIA Chain
Title	Smart Contract Audit Report
Target	AIA Bridge
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 28, 2023	Xuxian Jiang	Final Release
1.0-rc	December 27, 2023	Xuxian Jiang	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 `Bridge` contract in `AIA Bridge` 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 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 AIA Chain

`AIA Chain` is a distributed public blockchain designed to become an efficient, secure, and decentralized infrastructure that supports asset issuance and exchange. The `AIA Chain` is compatible with `EVM` and supports smart contracts. Anyone can join and build dApps and applications on it. The original digital asset of the `AIA Chain` is `AIA`, which uses a consensus-based `Byzantine Fault Tolerance (BFT)` consensus algorithm. The audited `AIA Bridge` provides the much-needed cross-chain bridge functionality. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of AIA Bridge

Item	Description
Name	AIA Bridge
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 28, 2023

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

- https://github.com/aiachain/bridge.git (d36da83)

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

- https://github.com/aiachain/bridge.git (60b7134)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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) [8], 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 `Bridge` contract in `AIA Chain`. 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	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

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 issues (shown in Table 2.1), including 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key AIA Bridge Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved validRequirement() Modifier in BridgeAdmin	Coding Practices	Resolved
PVE-002	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	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 Improved validRequirement() Modifier in BridgeAdmin

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BridgeAdmin
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The audited Bridge support has an admin-oriented BridgeAdmin contract. While examining this contract, we notice it has a modifier validRequirement() whose logic can be improved.

In the following, we show the implementation of this validRequirement() modifier. It has a rather straightforward logic in validating the owner amount as well as the required number. It comes to our attention that one specific requirement of ownerCount > 0 is redundant as it is guaranteed by other requirements, including _required <= ownerCount and _required > 0.

```
    modifier validRequirement(uint ownerCount, uint _required) {
        require(ownerCount <= MaxItemAdressNum
        && _required <= ownerCount
        && _required > 0
            && ownerCount > 0);
        _;
    }
```

Listing 3.1: BridgeAdmin::validRequirement()

Recommendation Revise the above modifier to remove redundant requirement.

Status The issue has been fixed by this commit: 60b7134.

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bridge`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the `Uniswap/Lendf.Me` hack [11].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `Bridge` as an example, the `withdrawNative()` function (see the code snippet below) is provided to externally call an entity (possibly a contract) to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 812) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
797     function withdrawNative(address payable to, uint value, string memory proof, bytes32
             taskHash) public
798     onlyOperator
799     whenNotPaused
800     positiveValue(value)
801     returns(bool)
802     {
803         require(address(this).balance >= value, "not enough native token");
804         require(taskHash == keccak256((abi.encodePacked(to,value,proof))),"taskHash is
                wrong");
805         uint256 status = logic.supportTask(logic.WITHDRAWTASK(), taskHash, msg.sender,
                operatorRequireNum);
806
807         if (status == logic.TASKPROCESSING()){
808             emit WithdrawingNative(to, value, proof);
809         }else if (status == logic.TASKDONE()) {
810             emit WithdrawingNative(to, value, proof);
811             emit WithdrawDoneNative(to, value, proof);
812             to.transfer(value);
```

```
            logic.removeTask(taskHash);
        }
        return true;
    }
```

Listing 3.2: `Bridge::withdrawNative()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and/or utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Note the function `withdrawToken()` can be similarly improved.

Status The issue has been fixed by this commit: `60b7134`.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

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 3.3: `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.

In the following, we show the `Bridge::transferToken()` routine. If the `USDT` token is supported as `token`, the unsafe version of `atoken.transfer(to,value)` (line 872) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace it with `safeTransfer()`.

```
870     function transferToken(address token, address to , uint256 value) onlyPauser
            external{
871         IERC20 atoken = IERC20(token);
872         bool success = atoken.transfer(to,value);
873     }
```

Listing 3.4: `Bridge::transferToken()`

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

Status The issue has been fixed by this commit: `60b7134`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BridgeAdmin`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `BridgeAdmin` contract, there is a privileged `owner` account that plays a critical role in governing and regulating the contract-wide operations (e.g., assign roles and drop tasks). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function dropAddress(string memory class, address oneAddress) public onlyOwner returns (bool){
        bytes32 classHash = getClassHash(class);
        require(classHash != STOREHASH && classHash != LOGICHASH, "wrong class");
        require(itemAddressExists(classHash, oneAddress), "no such address exist");

        if (classHash == OWNERHASH)
            require(getItemAddressCount(classHash) > ownerRequireNum, "insufficient addresses");

        bytes32 taskHash = keccak256(abi.encodePacked("dropAddress", class, oneAddress));
        addItemAddress(taskHash, msg.sender);
        if (getItemAddressCount(taskHash) >= ownerRequireNum) {
            removeOneItemAddress(classHash, oneAddress);
            emit AdminChanged("dropAddress", class, oneAddress, oneAddress);
            removeItem(taskHash);
            return true;
        }
        return false;
    }

    function addAddress(string memory class, address oneAddress) public onlyOwner returns (bool){
        bytes32 classHash = getClassHash(class);
        require(classHash != STOREHASH && classHash != LOGICHASH, "wrong class");

        bytes32 taskHash = keccak256(abi.encodePacked("addAddress", class, oneAddress));
        addItemAddress(taskHash, msg.sender);
        if (getItemAddressCount(taskHash) >= ownerRequireNum) {
            addItemAddress(classHash, oneAddress);
            emit AdminChanged("addAddress", class, oneAddress, oneAddress);
            removeItem(taskHash);
            return true;
        }
```

```
        return false;
    }

    function dropTask(bytes32 taskHash) public onlyOwner returns (bool){
        removeItem(taskHash);
        emit AdminTaskDropped(taskHash);
        return true;
    }
```

Listing 3.5: Example Privileged Operations in `BridgeAdmin`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be better if the privileged account is governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team intends to manage the admin keys with a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Bridge` contract in `AIA Chain`, which is a distributed public blockchain designed to become an efficient, secure, and decentralized infrastructure that supports asset issuance and exchange. The `AIA Chain` is compatible with `EVM` and supports smart contracts. Anyone can join and build dApps and applications on it. The original digital asset of the `AIA Chain` is `AIA`, which uses a consensus-based `Byzantine Fault Tolerance (BFT)` consensus algorithm. The audited `AIA Bridge` provides the much-needed cross-chain bridge functionality. The current code base is well 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[7] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[12] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.