SMART CONTRACT AUDIT REPORT

for

AIT

Prepared By: Xiaomi Huang

PeckShield

May 29, 2025

Document Properties

Client	AIT
Title	Smart Contract Audit Report
Target	AIT
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 29, 2025	Xuxian Jiang	Final Release
1.0-rc1	May 23, 2025	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the AIT 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 AIT

AIT is a digital currency based on blockchain technology, designed to support applications in the field of artificial intelligence (AI) and blockchain. As a crypto asset, AIT coins can be used not only for trading and investment, but also for various AI-driven decentralized platforms to help developers, researchers and enterprises collaborate and innovate in areas such as data, computing resources, algorithms and smart contracts. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of AIT

Item	Description
Name	AIT
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 29, 2025

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

- https://github.com/[anonymous]/AIT.git (04ad035)

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

- https://github.com/[anonymous]/AIT.git (8798734)

1.2 About PeckShield

PeckShield Inc. [7] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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 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

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

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) [5], 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 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 AIT protocol. 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	2	■ ■
Low	1	■
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 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key AIT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Necessity of Single-Shot Address Configuration	Coding Practices	Resolved
PVE-002	Low	Improved Referral Reward Accounting in InvestAITNew	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Besides 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 Necessity of Single-Shot Address Configuration

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: InvestAITNew
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

To engage the community, AIT has a core InvestAITNew contract which contains the logic to configure the AIT token address. While reviewing the contract logic, we notice an issue that may allow the privileged owner to repeatedly initialize the AIT token address.

To elaborate, we show below the related init() routine. The current logic properly assigns AIT token address. However, it comes to our attention that this routine allows for repeated address assignment. For improved confidence, we strongly suggest to make the initialization only once, blocking any further attempt to re-initialize the token contract.

```
    function init(address _ait) public onlyOwner {
        AIT = IERC20(_ait);
        DAY_SECONDS = 86400;
    }
```

Listing 3.1: InvestAITNew::init()

Recommendation Revisit the above initialization routine to ensure it can only be called once.

Status The issue has been fixed by this commit: 8798734.

3.2 Improved Referral Reward Accounting in InvestAITNew

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `InvestAITNew`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

`AIT` has a built-in referral system that allows to reward referrals with the purpose of engaging community. While examining the referral-related reward accounting logic, we notice current implementation can be improved.

To elaborate, we show below the code snippet from the related `investAIT()` routine. To reward referrals, the logic iterates the chain of referrals and reward the inviter based on the hard-code reward rate. It comes to our attention that the validity of an inviter is based on the address comparison with `userInfo[currAccount].inviter == address(this)` (line 204), which should be revised as `currAccount ==address(0)|| userInfo[currAccount].inviter == owner()`.

```
200         uint8[11] memory percentArr = [6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6];
201         for (uint256 i = 0; i < 11; i++) {
202             //the inviter
203             if (userInfo[currAccount].inviter == address(this)) {
204                 break;
205             }
206             address currentInviter = userInfo[currAccount].inviter;
207             if (!userLastInvestFinished(currentInviter)) {
208                 InvestInfo storage inviterLastInvestInfo = userInfo[currentInviter].
                        investInfo[userInfo[currentInviter].investInfo.length - 1];
209                 uint256 rewards = Math.min(amount, inviterLastInvestInfo.investAmount).
                        mul(percentArr[i]).div(100);
210                 inviterLastInvestInfo.pendingReferRewards = inviterLastInvestInfo.
                        pendingReferRewards.add(rewards);
211             }
212             currAccount = currentInviter;
213         }
```

Listing 3.2: `InvestAITNew::investAIT()`

Recommendation Revise the above routine to properly check the inviter validity.

Status The issue has been fixed by this commit: `8798734`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: InvestAITNew
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the AIT protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings, pause the contract, and withdraw funds). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setAddresses(address _foundationAddress, address _miningAddress) public onlyOwner {
        foundationAddress = _foundationAddress;
        miningAddress = _miningAddress;
    }

    function setDaySeconds(uint256 daySeconds) public onlyOwner {
        DAY_SECONDS = daySeconds;
    }

    function setVIPInfo(
        uint256 vipLevel,
        uint256 maxAccounts, uint256 investAmount,
        uint256 maxProfitAmount, uint256 releaseAmountEachDay
    ) public onlyOwner {
        if (vipLevel >= 3) revert VIPLevelMustFrom0To3();
        vipInfo[vipLevel].maxAccounts = maxAccounts;
        vipInfo[vipLevel].currAccounts = 0;
        vipInfo[vipLevel].investAmount = investAmount;
        vipInfo[vipLevel].maxProfitAmount = maxProfitAmount;
        vipInfo[vipLevel].releaseAmountEachDay = releaseAmountEachDay;
    }

    function addVIPMaxAccounts(
        uint256 vipLevel,
        uint256 addMaxAccounts
    ) public onlyOwner {
        if (vipLevel >= 3) revert VIPLevelMustFrom0To3();
        vipInfo[vipLevel].maxAccounts = vipInfo[vipLevel].maxAccounts.add(addMaxAccounts);
    }
```

```
    function setPause(bool pause) public onlyOwner {
        if (pause) {
            _pause();
        }
        else {
            _unpause();
        }
    }
```

Listing 3.3: Example Privileged Operations in `InvestAITNew`

```
    function withdrawETH(uint256 amount) public onlyOwner {
        if (amount == 0) revert WithdrawETHZeroAmount();
        uint256 balance = address(this).balance;
        if (balance < amount) revert WithdrawETHInsufficientBalance(amount, balance);
        payable(owner()).sendValue(amount);
    }

    function withdrawETHAll() public onlyOwner {
        uint256 balance = address(this).balance;
        if (balance == 0) revert WithdrawETHZeroAmount();
        payable(owner()).sendValue(balance);
    }

    function withdrawERC20(address token, uint256 amount) public onlyOwner {
        if (amount == 0) revert WithdrawERC20ZeroAmount(token);
        uint256 balance = IERC20(token).balanceOf(address(this));
        if (balance < amount) revert WithdrawERC20InsufficientBalance(token, amount,
            balance);
        IERC20(token).safeTransfer(owner(), amount);
    }

    function withdrawERC20All(address token) public onlyOwner {
        uint256 balance = IERC20(token).balanceOf(address(this));
        if (balance == 0) revert WithdrawERC20ZeroAmount(token);
        IERC20(token).safeTransfer(owner(), balance);
    }
```

Listing 3.4: Example Privileged Operations in `EmergencyInterfaceInstance`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 resolved with the removal of non-essential admin account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the AIT protocol, which is a digital currency based on blockchain technology, designed to support applications in the field of artificial intelligence (AI) and blockchain. As a crypto asset, AIT coins can be used not only for trading and investment, but also for various AI-driven decentralized platforms to help developers, researchers and enterprises collaborate and innovate in areas such as data, computing resources, algorithms and smart contracts. 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-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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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