SMART CONTRACT AUDIT REPORT

for

IN (OFT) Token & Staking

Prepared By: Xiaomi Huang

PeckShield

July 24, 2025

Document Properties

Client	INFINIT
Title	Smart Contract Audit Report
Target	IN OFT Token and Staking
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	Description
1.0	July 24, 2025	Xuxian Jiang	Final Release
1.0-rc	July 23, 2025	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 source code of the `IN` `(OFT)` token contract and the associated staking protocol, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC20 compliance issues or security concerns. This document outlines our audit results.

1.1 About IN OFT Token

`INFINIT` is an `AI`-powered `DeFi Intelligence` protocol, acting as an on-chain `DeFi` companion that helps users discover, evaluate, and execute `DeFi` opportunities through `AI Agents`. This audit focuses on its `OFT`-based `ERC20` token contract as well as the associated staking protocol. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of `IN` `OFT/Staking` Contracts

Item	Description
Name	INFINIT
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	July 24, 2025

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

- https://github.com/infinit-xyz/in-oft-token-contracts.git (0fd4dec)
- https://github.com/infinit-xyz/infinit-staking-contracts.git (04a4c95)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in.

- https://github.com/infinit-xyz/in-oft-token-contracts.git (0fd4dec)
- https://github.com/infinit-xyz/infinit-staking-contracts.git (19c4cdb)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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).

1.3 Methodology

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

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

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

We perform the audit according to the following procedures:

- 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.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.2: Vulnerability Severity Classification

Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

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 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.

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 1.3: 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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the IN token contract and associated staking 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 logics, examine system operations, and place ERC20-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	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. While there is no critical or high severity issue, 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 IN OFT Token and Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved MAX_ACTIVE_EPOCHS Enforcement in StakingRewarder	Business Logic	Resolved
PVE-002	Low	Timely Interest Accrual Upon Rewarder Removal in IN_Staking	Business Logic	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate the above issue(s), we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited IN token contract. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification.

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements, but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Upgradable	The token contract allows for future upgrades	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that the related token transfers are dis-allowed	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

4 Detailed Results

4.1 Improved MAX_ACTIVE_EPOCHS Enforcement in StakingRewarder

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewarder`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The audited `Staking` protocol is no exception. Specifically, if we examine the core `StakingRewarder` contract, it has defined a protocol-wide risk parameter, i.e., `MAX_ACTIVE_EPOCHS`, which indicates the maximum number of active epochs that may be queued. In the following, we show the corresponding `addNewEpoch()` routine that honors this risk parameter.

```
    function addNewEpoch(uint32 startTs, uint32 endTs, uint128 rewardsPerSec) external
        onlyOwner {
        require(startTs < endTs, StartTimeGreaterThanEndTime());
        // Check that the new epoch starts in the future
        require(startTs >= block.timestamp, StartTimeLessThanCurrentTimestamp());
        // NOTE: _epochs.length > 0 since first epoch is initialized in initialize
            function
        uint256 newEpochIndex = _epochs.length;
        // NOTE: We assume completedEpochCount is up-to-date; if not, it's only
            temporary,
        // and may make the number of active epochs appear higher than it actually is.
        // Ensure the number of active epochs does not exceed MAX_ACTIVE_EPOCHS.
        require(newEpochIndex - completedEpochCount < MAX_ACTIVE_EPOCHS,
            ActiveEpochsLimit());

        // if the new epoch is not the first epoch, check that the new epoch MUST start
            after the last epoch
```

```
123         require(newEpochIndex == 0 || startTs >= _epochs[newEpochIndex - 1].endTs,
                EpochOverlap());
124
125         Epoch memory newEpoch = Epoch({startTs: startTs, endTs: endTs, rewardsPerSec:
                rewardsPerSec});
126         // Add the new epoch
127         _epochs.push(newEpoch);
128
129         // Calculate the total reward
130         uint256 epochTotalRewards = _calculateRewards(endTs - startTs, rewardsPerSec);
131
132         // Transfer the rewards to the contract
133         if (epochTotalRewards != 0) {
134             asset.safeTransferFrom(msg.sender, address(this), uint256(epochTotalRewards)
                    );
135         }
136         emit EpochAdded(newEpochIndex, newEpoch.startTs, newEpoch.endTs, newEpoch.
                rewardsPerSec, epochTotalRewards);
137     }
```

Listing 4.1: `StakingRewarder::addNewEpoch()`

Our analysis shows that the above routine has an implicit assumption, i.e., the storage state of `completedEpochCount` is up-to-date. However, this implicit assumption can be eliminated by always calculating the latest `completedEpochCount` state. And the latest state can then be used to ensure the number of active epochs does not exceed `MAX_ACTIVE_EPOCHS` (line 120). Note the latest `completedEpochCount` state can be computed by calling the internal helper of `_calculateRewardsAndNewCompletedEpochCount()`.

Recommendation Revisit the above routine to properly enforce the `MAX_ACTIVE_EPOCHS` risk parameter.

Status The issue has been fixed by this commit: `19c4cdb`.

4.2 Timely Interest Accrual Upon Rewarder Removal in IN_Staking

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: IN_Staking
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The Staking protocol has a core IN_Staking contract that is in essence an ERC4626 staking vault, allowing users to stake tokens and earn rewards. By design, it supports multiple rewarder contracts and all withdrawals by design go through the intended unstaking schedule. While reviewing the management of multiple rewarder contracts, we notice the associated rewards can be timely accrued (especially when an existing rewarder contract is being removed).

In the following, we show the implementation of the related routine, i.e., removeRewarder(). As the name indicates, this routine is used to remove an existing rewarder contract. While it has a rather straightforward logic in simply removing the given rewarder contract from the internal _rewarders array, there is a need to ensure the rewards from the removed rewarder contract have been timely collected. In other words, we need to add the accrue() modifier to the removeRewarder() routine. This accrue() modifier also automatically updates the total managed assets with any new rewards received.

```
    /**
     * @dev Removes a rewarder contract address (owner only)
     * @param rewarder The address of the rewarder contract to remove
     */
    function removeRewarder(address rewarder) external onlyOwner {
        bool success = _rewarders.remove(rewarder);
        require(success, RewarderNotExist(rewarder));
        emit RewarderRemoved(rewarder);
    }
```

Listing 4.2: IN_Staking::removeRewarder()

Recommendation Revise the above routine to ensure the rewards are timely collected, including those from the rewarder contract when it is being removed.

Status The issue has been fixed by this commit: 19c4cdb.

4.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `IN` token contract and the associated staking protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., mint/burn tokens into/from circulation and upgrade the token contract). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contract.

```
    function mint(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
    }

    function burn(address _from, uint256 _amount) public onlyOwner {
        _burn(_from, _amount);
    }
```

Listing 4.3: Example Privileged Operations in `IN_Token`

```
    function addRewarder(address rewarder) external onlyOwner {
        require(_rewarders.length() < MAX_REWARDERS_LENGTH,
            ActiveRewardersLengthLimitExceeded());
        require(rewarder != address(0), ZeroAddress());
        bool success = _rewarders.add(rewarder);
        require(success, AlreadyAdded(rewarder));
        emit RewarderAdded(rewarder);
    }

    /**
     * @dev Removes a rewarder contract address (owner only)
     * @param rewarder The address of the rewarder contract to remove
     */
    function removeRewarder(address rewarder) external onlyOwner {
        bool success = _rewarders.remove(rewarder);
        require(success, RewarderNotExist(rewarder));
        emit RewarderRemoved(rewarder);
    }

    /**
     * @dev Sets the minimum deposit amount (owner only)
     * @param minDepositAmount_ The minimum deposit amount
    */
    function setMinDepositAmount(uint128 minDepositAmount_) external onlyOwner {
        _setMinDepositAmount(minDepositAmount_);
    }

    /**
     * @dev Sets the minimum withdraw amount (owner only)
     * @param minWithdrawAmount_ The minimum withdraw amount
     */
    function setMinWithdrawAmount(uint128 minWithdrawAmount_) external onlyOwner {
        _setMinWithdrawAmount(minWithdrawAmount_);
    }

    /**
     * @dev Sets the unstaking contract address (owner only)
     * @param unstakingContract_ The unstaking contract address
     */
    function setUnstakingContract(address unstakingContract_) external onlyOwner {
        unstakingContract = unstakingContract_;
        emit UnstakingContractSet(unstakingContract_);
    }
```

Listing 4.4: Example Privileged Operations in `IN_Staking`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it would be worrisome if the privileged account is a plain `EOA` account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`. In the meantime, a timelock-based mechanism can also be considered as mitigation.

In the meantime, the above contracts make use of the proxy pattern to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 This issue has been mitigated with the use of a multi-sig account to hold the `owner` account.

5 Conclusion

In this security audit, we have examined the design and implementation of the `IN` `(OFT)` token contract and the associated staking protocol. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, no major issue was found in these areas, and the current development follows the best practice. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

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

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

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

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

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

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