SMART CONTRACT AUDIT REPORT

for

Earn Farming Airdrop

Prepared By: Xiaomi Huang

PeckShield

October 22, 2024

Document Properties

Client	Earn Farming
Title	Smart Contract Audit Report
Target	Earn Farming Airdrop
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	October 22, 2024	Xuxian Jiang	Final Release
1.0-rc	October 21, 2024	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 `Earn Farming Airdrop` contract, 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 Earn Farming Airdrop

This audit covers the specific airdrop contract, i.e., `EarnFarmingAirdrop`. The airdrop works by generating a `Farm Tree` from all user addresses and rewards, and the root of the tree stored in the contract. Users can activate claim their rewards by depositing funds into participating farm wallets. Basically, the team will deposit airdrop rewards into the contract and users can activate and claim their rewards after the claiming time starts. Note that after the claiming time ends, users are unable to claim rewards. After a specified reclaim expiry time, the platform can reclaim any unclaimed rewards. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Earn Farming Airdrop
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 2024

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

- https://github.com/lEarnFarming-dao/EarnFarmingAirdrop.git (16e7f88)

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

- https://github.com/EarnFarming-dao/EarnFarmingAirdrop.git (feedc6c)

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

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

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.

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) [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 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 implementation of the `EarnFarmingAirdrop` 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 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	2	🟦 🟦
Informational	1	🟦
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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Lista Airdrop Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Owner Validation in SUSDT-Token::permit()	Coding Practices	Resolved
PVE-002	Low	Improved Parameter Validation in EarnFarmingAirdrop	Coding Practice	Resolved
PVE-003	Informational	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	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 Owner Validation in SUSDT Token::permit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SUSDTToken
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

To facilitate the user interaction, the SUSDTToken contract has the built-in support of the EIP2612-compliant functionality. In particular, the permit() function is introduced to simplify the call forwarding process.

To elaborate, we show below this permit() routine in the SUSDTToken contract. This routine ensures that the given owner is indeed the one who signs the approve request. Note that the internal implementation makes use of the ecrecover() precompile for validation (line 81). It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., owner, is not equal to address(0).

```
55     function permit(
56         address owner,
57         address spender,
58         uint256 amount,
59         uint256 deadline,
60         uint8 v,
61         bytes32 r,
62         bytes32 s
63     ) external override {
64         require(block.timestamp <= deadline, "ERC20Permit: expired deadline");
65         bytes32 digest = keccak256(
66             abi.encodePacked(
67                 "\x19\x01",
68                 DOMAIN_SEPARATOR,
69                 keccak256(
                    abi.encode(
                        PERMIT_TYPE_HASH,
                        owner,
                        spender,
                        amount,
                        _nonces[owner]++,
                        deadline
                    )
                )
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress == owner, "ERC20Permit: invalid signature");
        _approve(owner, spender, amount);
    }
```

Listing 3.1: SUSDTToken::permit()

Recommendation Strengthen the permit() routine to ensure the recoveredAddress is not equal to address(0).

Status The issue has been resolved as the token contract ensures that it cannot transfer any token into address(0).

3.2 Improved Parameter Validation in EarnFarmingAirdrop

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: EarnFarmingAirdrop
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The EarnFarmingAirdrop contract is no exception. Specifically, if we examine its implementation, this contract has defined a number of contract-wide risk parameters, such as startBlock/endBlock and reclaimPeriod. In the following, we show the corresponding routines that allow for their changes.

```
    function setStartBlock(uint256 _startBlock) external onlyOwner {
        require(_startBlock != startBlock, "Start block already set");
        startBlock = _startBlock;
    }

     function setEndBlock(uint256 _endBlock) external onlyOwner {
         require(_endBlock != endBlock, "End block already set");
        endBlock = _endBlock;
    }
```

Listing 3.2: EarnFarmingAirdrop :: setStartBlock/setStartBlock()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above routines can be improved by further validating `require(startBlock >= block.number && startBlock < endBlock)`.

In addition, the `reclaimPeriod` parameter is defined on seconds while both `startBlock` and `endBlock` are defined based on blocks. And there is an inherent requirement, i.e., the `reclaimPeriod` should not expire if the `endBlock` is not passed yet. And this inherent requirement is not enforced yet.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been fixed in the following commits: e1b276f, 0445209, and feedc6c.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SUSDTToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [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 the following, we examine the `transfer()` routine and related 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. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `reclaim()` routine in the `ListaToken` contract. If the `USDT` token is supported as `token`, the unsafe version of `require(IERC20(token).transfer(msg.sender, amount))` (line 62) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
60      function reclaim(uint256 amount) external onlyOwner {
61          require(block.timestamp > reclaimPeriod, "Tokens cannot be reclaimed");
62          require(IERC20(token).transfer(msg.sender, amount), "Transfer failed");
63      }
```

Listing 3.4: SUSDTToken::reclaim()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been resolved as the team confirms the supported token is fully ERC20-compliant.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: EarnFarmingAirdrop
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the EarnFarmingAirdrop contract, there is a privileged account owner that plays a critical role in governing and regulating the contract-wide operations (e.g., update parameters and reclaim airdrop tokens). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```
    function setFarmRoot(bytes32 _FarmRoot) external onlyOwner {
        FarmRoot = _FarmRoot;
    }

    function setStartBlock(uint256 _startBlock) external onlyOwner {
        require(_startBlock != startBlock, "Start block already set");
        startBlock = _startBlock;
    }

    function setEndBlock(uint256 _endBlock) external onlyOwner {
        require(_endBlock != endBlock, "End block already set");
        endBlock = _endBlock;
    }
    ...
    function reclaim ( uint256 amount ) external only Owner {
        require(block.timestamp > reclaimPeriod, "Tokens cannot be reclaimed");
        require(IERC20(token).transfer(msg.sender, amount), "Transfer failed");
    }
```

Listing 3.5: Example Privileged Operations in EarnFarmingAirdrop

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

In the meantime, the vault contract makes use of the proxy contract 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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the specific airdrop contract, i.e., `EarnFarmingAirdrop`. The airdrop works by generating a `Farm Tree` from all user addresses and rewards, and the root of the tree stored in the contract. Users can activate and claim their rewards by depositing funds into participating farm wallets. Basically, the team will deposit airdrop rewards into the contract and users can activate and claim their rewards after the claiming time starts. Note that after the claiming time ends, users are unable to claim rewards. After a specified reclaim expiry time, the platform can reclaim any unclaimed rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

5 MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.