SMART CONTRACT AUDIT REPORT

for

Earn Farming DAO

Prepared By: Xiaomi Huang

PeckShield

September 26, 2024

Document Properties

Client	Earn Farming
Title	Smart Contract Audit Report
Target	Earn Farming DAO
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 26, 2024	Xuxian Jiang	Final Release
1.0-rc	September 23, 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 DAO 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 Earn Farming DAO

Earn Farming DAO acts as the DAO for the Earn Farming ecosystem. It features a voting-escrowed Earn Farming DAO token – the voting power obtained by locking Farming token. It also features an advanced reward management subsystem that gives incentive for Earn Farming DAO token and other staking users. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Earn Farming DAO
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 26, 2024

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit. The given repository contains a number of contracts and this audit only covers the following contracts, i.e., CommonEarnFarmingDistributor.sol, PremiumPoolAutoCompounder.sol, EarnFarming.sol PremiumPoolDistributor.sol, ERC20LpEarnFarmingDistributor.sol, EarnFarmingVault.sol LpProxy.sol, PremiumPool. sol, PoolVault.sol and PoolStaking.sol.

- https://github.com/earnfarming/earnfarming.git (700b1b3)

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

- https://github.com/earnfarming/earnfarming.git (42fc764)

1.2 About PeckShield

PeckShield Inc. [9] 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		
Impact	*High*	*Medium*	*Low*
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 [8]:

- 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) [7], 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 Earn Farming DAO 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	1	🟦
Low	2	🟦 🟦
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, 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 2 low-severity vulnerabilities.

Table 2.1: Key EarnFarming DAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Premium Pool Registration and Apply	Business Logic	Resolved
PVE-002	Low	Improved Initialization Logic in EarnFarming And Premium Pool	Coding Practices	Resolved
PVE-003	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 Premium Pool Registration and Apply

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: PoolStaking/PremiumPool
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Earn Farming DAO protocol has the built-in incentivization mechanism to reward participating users. In particular, it allows the registration of new pools with new LP tokens as the underlying tokens. In the process of examining the new pool registration logic, we notice current implementation can be improved.

To elaborate, we show below the PremiumPool() routine that registers new staking pool. The new staking pool can be better validated, especially regarding the given rewardToken. More specifically, the given rewardToken should be the same as the rewardToken configured in the StakingVault contract. With that, we suggest to add the following requirement, i.e., require(vault.rewardToken() == rewardToken). Note the same function in another contract PoolStaking can be similarly improved.

```
    function registerPool(address lpToken, address rewardToken, address poolAddress,
        address distributor) external onlyOwner {
        require(!pools[lpToken].isActive, "Pool is active");

        pools[lpToken] = Pool({
            lpToken: lpToken,
            rewardToken: rewardToken,
            poolAddress: poolAddress,
            distributor: distributor,
            isActive: true
        });
    }
```

Listing 3.1: PoolStaking::registerPool()

Recommendation Strengthen the above-mentioned routines to properly validate the given rewardToken.

Status This issue has been fixed in the following commit: 5a8950e.

3.2 Improved Initialization Logic in Earn Farming And Premium Pool

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

Description

To facilitate possible future upgrade, a number of contracts in Earn Farming DAO is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current initialization can be improved.

In the following, we shows the initialization routine of the PoolStaking contract. We notice it has two parent contracts OwnableUpgradeable and ReentrancyGuardUpgradeable. Current initialization routine only initializes the first parent contract, but not the second parent contract. To improve, we suggest to call __ReentrancyGuard_init() to initialize the ReentrancyGuardUpgradeable parent contract. Note a number of proxy contracts can be similarly improved, including PoolVault, EarnFarmingVault, and PoolStaking.

```
    function initialize(address _owner, address _vault) public initializer {
        require(_owner != address(0), "owner cannot be zero address");
        require(_vault != address(0), "vault cannot be zero address");
        __Ownable_init();
        transferOwnership(_owner);

        vault = _vault;
    }
```

Listing 3.2: PoolStaking::initialize()

Recommendation Improve the above-mentioned initialization routines to ensure every inherited parent contract is properly initialized.

Status This issue has been fixed in the following commit: 5a8950e.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Earn Farming DAO protocol contract, there is a privileged account (with the assigned DEFAULT_ADMIN_ROLE or owner) that plays a critical role in governing and regulating the DAO-wide operations (e.g., manage reward pools, configure parameters, upgrade proxies and execute other privileged operations). In the following, we show the representative functions potentially affected by the privilege of this account.

```
    function setFeeRate(uint256 _feeRate) external onlyOwner {
        require(_feeRate <= MAX_FEE, "Fee rate is too high");
        feeRate = _feeRate;
    }

    /**
     * @dev set staking address
     * @param _staking staking address
     */
    function setStaking(address _staking) external onlyOwner {
        require(_staking != address(0), "staking cannot be zero address");
        staking = _staking;
    }

    /**
     * @dev set lp proxy address
     * @param _lpProxy lp proxy address
     */
    function setLpProxy(address _lpProxy) external onlyOwner {
        require(_lpProxy != address(0), "lpProxy cannot be zero address");
        lpProxy = _lpProxy;
    }
```

Listing 3.3: Example Privileged Operations in PoolVault

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 Earn Farming DAO protocol, which acts as the DAO for the Earn Farming ecosystem. It features a voting-escrowed Earn Farming DAO token – the voting power obtained by locking Farming token. It also features an advanced reward management subsystem that gives incentive for Earn Farming DAO token and other staking users. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

8 OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

9 PeckShield. PeckShield Inc. https://www.peckshield.com.