SMART CONTRACT AUDIT REPORT

for

Gin Finance Farm

Prepared By: Xiaomi Huang

PeckShield

June 9, 2022

Document Properties

Client	Gin Finance
Title	Smart Contract Audit Report
Target	Gin Finance Farm
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 9, 2022	Xiaotao Wu	Final Release
1.0-rc1	June 4, 2022	Xiaotao Wu	Release Candidate #1

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 Farm support of the Gin Finance 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 Gin Finance Farm

Gin Finance is an open-source protocol which aims to be the ultimate DeFi solution for asset liquidity and exchange on the BOBA network. With its efficient swap and staking model, it offers a wide range of decentralized applications to DeFi users at convenience. The audited Gin Finance feature allows users to stake LP tokens and earn GIN rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Gin Finance Farm

Item	Description
Name	Gin Finance
Website	https://www.gin.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 9, 2022

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

- https://github.com/ginfidev/GinFinance-Farm.git (73ed04b)

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

- https://github.com/ginfidev/GinFinance-Farm.git (49a5cc1)

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 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 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

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

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) [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.

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 `Gin Finance Farm` feature implementation. 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 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 Gin Finance Farm Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	Low	Revisited Logic In StakingRewards::notifyRewardAmount()	Business Logic	Resolved
PVE-003	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, 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 details.

3 Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

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

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 3.1: 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 `notifyRewardAmount()` routine in the `LockedStakingRewardsFactory` contract. If the `USDT` token is supported as `IERC20(rewardsToken)`, the unsafe version of `IERC20(rewardsToken).transfer(info.stakingRewards, rewardAmount)` (line 74) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
60      // notify reward amount for an individual staking token.
61      // this is a fallback in case the notifyRewardAmounts costs too much gas to call for all contracts
62      function notifyRewardAmount(uint256 index) public {
63          require(block.timestamp >= stakingRewardsGenesis, 'LockedStakingRewardsFactory::notifyRewardAmount: not ready');
64          require(index < stakingRewardsInfoList.length, 'LockedStakingRewardsFactory::notifyRewardAmount: index out of range');
65
66          StakingRewardsInfo storage info = stakingRewardsInfoList[index];
67          require(info.stakingRewards != address(0), 'LockedStakingRewardsFactory::notifyRewardAmount: not deployed');
68
69          if (info.rewardAmount > 0) {
70              uint rewardAmount = info.rewardAmount;
71              info.rewardAmount = 0;
72
73              require(
74                  IERC20(rewardsToken).transfer(info.stakingRewards, rewardAmount),
75                  'LockedStakingRewardsFactory::notifyRewardAmount: transfer failed'
76              );
77              LockedStakingRewards(info.stakingRewards).notifyRewardAmount(rewardAmount);
78              emit RewardNotified(index, rewardAmount);
79          }
80      }
```

Listing 3.2: LockedStakingRewardsFactory::notifyRewardAmount()

Note a number of routines in the `Gin Finance Farm` contracts can be similarly improved, including `LockedStakingRewardsFactory::extendStakingRewards()`, `StakingRewardsFactory::notifyRewardAmount()/extendStakingRewards()`, and `VestingStakingRewardsFactory::notifyRewardAmount()/extendStakingRewards()`.

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

Status This issue has been fixed in the following commit: `2a64433`.

3.2 Revisited Logic In StakingRewards::notifyRewardAmount()

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

Description

The `StakingRewards` contract provides an external `notifyRewardAmount()` function for the privileged `rewardsDistribution` account to update the values of user reward calculation related state variables, i.e., `rewardRate`, `lastUpdateTime`, and `periodFinish`. While examining the range check logic for the state variable `rewardRate`, we observe the need to revisit the current design.

To elaborate, we show below the related code snippet of the `StakingRewards` contract. Specifically, the `rewards` earned by users should be excluded from `balance` for calculating the upper limit of `rewardRate` (line 126). Moreover, the `remaining` and `leftover` also need to be taken into account when calculating the upper limit of `rewardRate`.

```
112    function notifyRewardAmount(uint256 reward) external onlyRewardsDistribution
           updateReward(address(0)) {
113        if (block.timestamp >= periodFinish) {
114            rewardRate = reward.div(rewardsDuration);
115        } else {
116            uint256 remaining = periodFinish.sub(block.timestamp);
117            uint256 leftover = remaining.mul(rewardRate);
118            rewardRate = reward.add(leftover).div(rewardsDuration.add(remaining));
119        }
120
121        // Ensure the provided reward amount is not more than the balance in the
               contract.
122        // This keeps the reward rate in the right range, preventing overflows due to
123        // very high values of rewardRate in the earned and rewardsPerToken functions;
124        // Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
```

```
        uint balance = rewardsToken.balanceOf(address(this));
        require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");

        lastUpdateTime = block.timestamp;

        if (block.timestamp >= periodFinish) {
            periodFinish = block.timestamp.add(rewardsDuration);
        } else {
            periodFinish = periodFinish.add(rewardsDuration);
        }

        emit RewardAdded(reward);
    }
```

Listing 3.3: `StakingRewards::notifyRewardAmount()`

Note a similar issue also exists in the `LockedStakingRewards` and `VestingStakingRewards` contracts.

Recommendation Revisit the above logic to correct calculate the upper limit of `rewardRate`.

Status This issue has been fixed in the following commit: `49a5cc1`.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In `Gin Finance Farm`, the `StakingRewards` contract is designed to be the main entry point for interaction with users. In particular, one entry routine, i.e., `stake()`, allows a user to transfer the supported assets (e.g., `stakingToken`) to the `StakingRewards` contract and earn rewards. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `StakingRewards` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `stake()` routine that is used to transfer `stakingToken` to the `StakingRewards` contract.

```
    function stake(uint256 amount) external nonReentrant updateReward(msg.sender) {
        require(amount > 0, "Cannot stake 0");
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
```

```
        emit Staked(msg.sender, amount);
    }
```

Listing 3.4: `StakingRewards::stake()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `stake()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Gin Finance Farm` for staking. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note a similar issue also exists in the `LockedStakingRewards`, `LockedStakingRewardsFactory`, `StakingRew-ardsFactory`, `VestingStakingRewards`, `VestingStakingRewardsFactory`, and `RewardLocker` contracts.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been resolved as the team confirms that `Gin Finance Farm` will not support deflationary/rebasing tokens.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Gin Finance Farm` feature, there is a privileged account, i.e., `owner`. The `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., deploy new staking contract and extend the staking rewards for a specified staking contract). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `LockedStakingRewardsFactory` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    // deploy a staking reward contract for the staking token, and store the reward amount
    // the reward will be distributed to the staking reward contract no sooner than the genesis
    // rewardDuration takes the time as second, 1 day = 86400
    function deploy(address stakingToken, uint rewardAmount, uint256 rewardDuration) public onlyOwner {

        StakingRewardsInfo memory info;

        info.stakingRewards = address(new LockedStakingRewards(/*_rewardsDistribution=*/ address(this), rewardsToken, stakingToken, rewardDuration));
        info.rewardAmount = rewardAmount;
        info.deployAt = block.timestamp;
        info.stakingToken = stakingToken;

        stakingRewardsInfoList.push(info);

        emit PoolDeployed(stakingRewardsInfoList.length - 1, stakingToken, info.stakingRewards, rewardAmount, rewardDuration);
    }

    function extendStakingRewards(uint256 index, uint256 rewardAmount) external onlyOwner {
        require(block.timestamp >= stakingRewardsGenesis, 'LockedStakingRewardsFactory::extendStakingRewards: not ready');
        require(index < stakingRewardsInfoList.length, 'LockedStakingRewardsFactory::extendStakingRewards: incorrect index');
        require(rewardAmount > 0, 'LockedStakingRewardsFactory::extendStakingRewards: incorrect rewardAmount');

```

```
        StakingRewardsInfo storage info = stakingRewardsInfoList[index];
        require(info.stakingRewards != address(0), 'LockedStakingRewardsFactory::
            extendStakingRewards: not deployed');
        require(info.rewardAmount == 0, 'LockedStakingRewardsFactory::
            extendStakingRewards: not started');

        require(
                IERC20(rewardsToken).transfer(info.stakingRewards, rewardAmount),
                'LockedStakingRewardsFactory::extendStakingRewards: transfer failed'
            );
        LockedStakingRewards(info.stakingRewards).notifyRewardAmount(rewardAmount);
        emit PoolExtended(index, rewardAmount);
    }
```

Listing 3.5: Example Privileged Operations in LockedStakingRewardsFactory

Note if the privileged owner extends the staking rewards for a specified staking contract, the periodFinish of this staking contract also will be extended (lines 131-135). Thus a user may be unable to withdraw his/her staked assets from the staking contract in time.

```
113     function notifyRewardAmount(uint256 reward) external onlyRewardsDistribution
            updateReward(address(0)) {
114         if (block.timestamp >= periodFinish) {
115             rewardRate = reward.div(rewardsDuration);
116         } else {
117             uint256 remaining = periodFinish.sub(block.timestamp);
118             uint256 leftover = remaining.mul(rewardRate);
119             rewardRate = reward.add(leftover).div(rewardsDuration.add(remaining));
120         }
121
122         // Ensure the provided reward amount is not more than the balance in the
                contract.
123         // This keeps the reward rate in the right range, preventing overflows due to
124         // very high values of rewardRate in the earned and rewardsPerToken functions;
125         // Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
126         uint balance = rewardsToken.balanceOf(address(this));
127         require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");
128
129         lastUpdateTime = block.timestamp;
130
131         if (block.timestamp >= periodFinish) {
132             periodFinish = block.timestamp.add(rewardsDuration);
133         } else {
134             periodFinish = periodFinish.add(rewardsDuration);
135         }
136
137         emit RewardAdded(reward);
138     }
```

Listing 3.6: LockedStakingRewards::notifyRewardAmount()

```
    function withdraw(uint256 amount) public nonReentrant updateReward(msg.sender) {
```

```
        require(amount > 0, "Cannot withdraw 0");
        require(block.timestamp >= periodFinish, "Cannot withdraw while locked");
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        stakingToken.safeTransfer(msg.sender, amount);
        emit Withdrawn(msg.sender, amount);
    }
```

Listing 3.7: `LockedStakingRewards::withdraw()`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` 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.

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 This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Gin Finance Farm` feature design and implementation. The audited `Gin Finance` feature allows users to stake `LP` tokens and earn `GIN` 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 `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-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] 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.