SMART CONTRACT AUDIT REPORT

for

FarmHero

Prepared By: Yiqun Chen

PeckShield

July 20, 2021

Document Properties

Client	FarmHero
Title	Smart Contract Audit Report
Target	FarmHero
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Shulin Bie
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 20, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 18, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
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 FarmHero 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 FarmHero

FarmHero is a novel protocol that mixes NFT, gaming and DeFi concepts. The team aims to build a healthy, long-run community that different people can find different fun. The community can contribute and earn in a various of ways, including but not limited to, yield farming, NFT farming, NFT trading, bug bounty, participation in the incubators and the house games, as well as new user referrals. The protocol's continuous token ecosystem is built upon renowned yield farms, with possible candidates of PancakeSwap on BSC, SushiSwap on Ethereum, QuickSwap on Polygon, MDex on BSC and Heco, SalmonSwap on Tron, etc.

The basic information of the FarmHero protocol is as follows:

Table 1.1: Basic Information of The AladdinDAO Protocol

Item	Description
Issuer	FarmHero
Website	https://www.farmhero.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 20, 2021

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

- https://github.com/farmhero/polygon-core.git (72130b4)

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

- https://github.com/farmhero/polygon-core.git (7d5f517)

1.2 About PeckShield

PeckShield Inc. [11] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 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) [9], 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 1.3: 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

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 design and implementation of the `FarmHero` 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	1	■
Low	5	■■■■■
Informational	0	
Total	8	

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 2 high-severity vulnerabilities, 1 medium-severity vulnerability, and 5 low-severity vulnerabilities.

Table 2.1: Key FarmHero Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Confirmed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-003	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-004	Medium	Proper Withdraw Fee Collection in withdraw()	Business Logic	Fixed
PVE-005	High	Proper Share Accounting in emergencyWithdrawNFT()	Business Logic	Fixed
PVE-006	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-007	High	Improved Logic in inCaseTokensGetStuck()	Business Logic	Fixed
PVE-008	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	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 [6]
- CWE subcategory: CWE-841 [4]

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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `forceExit()` routine in the `Playerbook` contract. If the `USDT` token is supported as `_token`, the unsafe version of `_token.transfer(msg.sender, _token.balanceOf(address(this)))` (line 419) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
417    function forceExit(IERC20 _token) public {
418        require(msg.sender == dev);
419        _token.transfer(msg.sender, _token.balanceOf(address(this)));
420    }
```

Listing 3.2: Playerbook::forceExit()

Note this issue is also applicable to other routines, including `withdarwReard()` and `incomingReward()` from the same `Playerbook` contract.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status This issue has been confirmed.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Playerbook`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `Playerbook` as an example, the `withdarwReard()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 398) starts before effecting the update on the internal state (line 400), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
394     function withdarwReard() external returns (bool) {
395         uint256 reward = ReferralReward[msg.sender];
396         require(rewardToken.balanceOf(address(this)) >= reward);
397 
398         rewardToken.transfer(msg.sender, reward);
399 
400         ReferralReward[msg.sender] = 0;
401 
402         return true;
403     }
```

Listing 3.3: `Playerbook::withdarwReard()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy. However, it is important to take precautions in making use of nonReentrant to block possible re-entrancy.

Recommendation Apply necessary reentrancy prevention by utilizing the nonReentrant modifier to block possible re-entrancy.

Status The issue has been addressed by the following commit: 41b430d.

3.3 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: HeroFarmV3
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The FarmHero protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's HERO allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner nonReentrant {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint[poolInfo[_pid].poolType] = totalAllocPoint[poolInfo[_pid].
            poolType].sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
        poolExistence[poolInfo[_pid].want] = _allocPoint > 0;
    }
```

Listing 3.4: HeroFarmV3::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

Status The issue has been addressed by the following commit: `41b430d`.

3.4 Proper Withdraw Fee Collection in withdraw()

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `HeroFarmV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

At the core of `FarmHero` is the `HeroFarmV3` contract that is developed on the widely-used `MasterChef` contract. Moreover, it adds the new feature of supporting NFT-based staking and unstaking. While examining the current logic, we notice the current implementation supports the collection of withdraw fee and the current fee collection logic is flawed.

To elaborate, we show below the full implementation of the `withdraw()` function. Our analysis shows that the withdraw fee collection is contingent on the following conditions `if (withdrawFee || !feeExclude[msg.sender])` (line 647). Basically, it requires two conditions: the first one is the `withdrawFee` is intended and the second one is that the user is not excluded. However, the current implementation collects the withdraw fee as long as one condition is satisfied. For correction, it requires both conditions to be met at the same time, i.e., `if (withdrawFee && !feeExclude[msg.sender])`.

```
602     // Withdraw LP tokens from MasterChef.
603     function withdraw(uint256 _pid, uint256 _wantAmt) public isEOA nonReentrant
            whenNotPaused {
604
605         updatePool(_pid);
606
607         PoolInfo storage pool = poolInfo[_pid];
608         UserInfo storage user = userInfo[_pid][msg.sender];
609
```

```
        require(pool.poolType == PoolType.ERC20, "invalid erc20");

        uint256 wantLockedTotal =
            IStrategy(poolInfo[_pid].strat).wantLockedTotal();
        uint256 sharesTotal = IStrategy(poolInfo[_pid].strat).sharesTotal();

        require(user.shares > 0, "user.shares is 0");
        require(sharesTotal > 0, "sharesTotal is 0");

        // Withdraw pending HERO
        uint256 pending =
            user.shares.mul(pool.accHEROPerShare).div(1e12).sub(
                user.rewardDebt
            );
        if (pending > 0) {
            _withdrawReward(pending);
        }

        // Withdraw want tokens
        uint256 amount = user.shares.mul(wantLockedTotal).div(sharesTotal);
        if (_wantAmt > amount) {
            _wantAmt = amount;
        }
        if (_wantAmt > 0) {
            uint256 sharesRemoved =
                IStrategy(poolInfo[_pid].strat).withdraw(msg.sender, _wantAmt);

            if (sharesRemoved > user.shares) {
                user.shares = 0;
            } else {
                user.shares = user.shares.sub(sharesRemoved);
            }

            uint256 wantBal = IERC20(pool.want).balanceOf(address(this));
            if (wantBal < _wantAmt) {
                _wantAmt = wantBal;
            }
            if(withdrawFee  !feeExclude[msg.sender]) {
                uint256 feeRate = _calcFeeRateByGracePeriod(uint256(user.gracePeriod));
                if(feeRate > 0){
                    uint256 feeAmount = _wantAmt.mul(feeRate).div(10000);
                    _wantAmt = _wantAmt.sub(feeAmount);
                    IERC20(pool.want).safeTransfer(feeAddress, feeAmount);
                }
            }
            IERC20(pool.want).safeTransfer(address(msg.sender), _wantAmt);
        }

        if(_wantAmt == 0 && rewardDistribution!=address(0)) {
            IRewardDistribution(rewardDistribution).earn(address(this));
        }

```

```
        user.rewardDebt = user.shares.mul(pool.accHEROPerShare).div(1e12);

        // If user withdraws all the LPs, then gracePeriod is cleared
        if (user.shares == 0) {
            user.gracePeriod = 0;
        }

        emit Withdraw(msg.sender, _pid, _wantAmt);
    }
```

Listing 3.5: `HeroFarmV3::withdraw()`

Recommendation Improve the above `withdraw()` function by collecting the withdraw fee when `(withdrawFee && !feeExclude[msg.sender])` is evaluated to be `true`.

Status The issue has been addressed by the following commit: `41b430d`.

3.5 Proper Share Accounting in emergencyWithdrawNFT()

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `HeroFarmV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.4, the `FarmHero` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint *100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool. With the `NFT` support, the protocol provides the `emergencyWithdrawNFT()` function to allow for emergency `NFT` withdraws.

To elaborate, we show below the full implementation of this function. This function properly transfers out the requested `NFTs` (line 776), but fails to properly record the internal states, including the user `shares`, `rewardDebt`, and `gracePeriod` (lines 782 – 784).

```
758    function emergencyWithdrawNFT(uint256 _pid, uint256[] memory _tokenIds) public isEOA
            nonReentrant {
759
760        PoolInfo storage pool = poolInfo[_pid];
761        UserInfo storage user = userInfo[_pid][msg.sender];
762        require(pool.poolType == PoolType.ERC721, "invalid erc721");
763
```

```
        if (_tokenIds.length > 0) {
            uint256 sharesRemoved =
                IStrategy(poolInfo[_pid].strat).withdraw(msg.sender, _tokenIds);

            if (sharesRemoved > user.shares) {
                user.shares = 0;
            } else {
                user.shares = user.shares.sub(sharesRemoved);
            }

            for(uint i = 0; i < _tokenIds.length; i++)
            {
                IERC721(pool.want).transferFrom(address(this), msg.sender, _tokenIds[i]);
            }

        }

        emit EmergencyWithdrawNFT(msg.sender, _pid, _tokenIds);
        user.shares = 0;
        user.rewardDebt = 0;
        user.gracePeriod = 0;
    }
```

Listing 3.6: `HeroFarmV3::emergencyWithdrawNFT()`

Recommendation Correct the above `emergencyWithdrawNFT()` function by properly recording the user states.

Status The issue has been addressed by the following commit: `41b430d`.

3.6 Improved Sanity Checks For System Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HeroFarmV3`
- 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 `FarmHero` protocol is no exception. Specifically, if we examine the `HeroFarmV3` contract, it has defined a number of protocol-wide risk parameters, such as `reservedNFTFarmingRate` and `teamRate`. In the following, we show the corresponding routines that allow for their changes.

```
    function setRates(uint256 _teamRate, uint256 _communityRate, uint256 _ecosystemRate,
        uint256 _reservedNFTFarmingRate) external onlyOwner {
        teamRate = _teamRate;
        communityRate = _communityRate;
        ecosystemRate = _ecosystemRate;
        reservedNFTFarmingRate = _reservedNFTFarmingRate;
    }
    ...
    function setEpochReduceRate(uint256 _epochReduceRate) external onlyOwner {
        epochReduceRate = _epochReduceRate;
    }

    function setTotalEpoch(uint256 _totalEpoch) external onlyOwner {
        totalEpoch = _totalEpoch;
    }
    ...
    function setNftRewardRate(uint256 _rate) external onlyOwner {
        nftRewardRate = _rate;
    }
```

Listing 3.7: `HeroFarmV3::setRates()` and `HeroFarmV3::setFeeExclude()`

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, an unlikely mis-configuration of `teamRate` may charge unreasonably high share in the `updatePool()` operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been addressed by the following commit: `41b430d`.

3.7 Improved Logic in inCaseTokensGetStuck()

- ID: PVE-007
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `HeroFarmV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.4, the `FarmHero` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint *100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool. Our analysis also reveals a privileged function inCaseTokensGetStuck() that needs to be improved to prevent user pool tokens from being taken.

To elaborate, we show below the full implementation of this function. This function is a rather straightforward one in transferring out the requested token. However, it only validates the input token is not the HERO token and fails to ensure the token should not be any staked pool token. Otherwise, the staked funds from participating users may be at risk.

```
    function inCaseTokensGetStuck(address _token, uint256 _amount)
        public
        onlyOwner
    {
        require(_token != HERO, "!safe");
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }
```

Listing 3.8: HeroFarmV3::inCaseTokensGetStuck()

Recommendation Correct the above inCaseTokensGetStuck() function by preventing the staked pool tokens from being possibly transferred out.

Status The issue has been fixed by this commit: 7d5f517.

3.8 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StratX2_HERO_QuickV2
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The StratX2_HERO_QuickV2 contract has a helper routine, i.e., _safeSwap(), that is designed to swap one token to another. It has a rather straightforward logic in computing the intended amount after conversion and then performing the actual swap via the UniswapV2 router.

```
    function _safeSwap(
        address _uniRouterAddress,
        uint256 _amountIn,
        uint256 _slippageFactor,
        address[] memory _path,
        address _to,
        uint256 _deadline
```

```
    ) internal virtual {
        uint256[] memory amounts =
            IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
        uint256 amountOut = amounts[amounts.length.sub(1)];

        IPancakeRouter02(_uniRouterAddress)
            .swapExactTokensForTokensSupportingFeeOnTransferTokens(
            _amountIn,
            amountOut.mul(_slippageFactor).div(1000),
            _path,
            _to,
            _deadline
        );
    }
```

Listing 3.9: `StratX2_HERO_QuickV2::_safeSwap()`

To elaborate, we show above the `_safeSwap()` routine. We notice the token swap is routed to `_uniRouterAddress` and the actual swap operation `swapExactTokensForTokensSupportingFeeOnTransferTokens()` essentially does not specify any effective restriction [1] on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the teams. And the team clarifies that the code is part of `PlayBook` and its impact is considered not too much. With that, the team decides to leave it as is.

[1] The current approach of specifying the slippage control via `amountOut.mul(_slippageFactor).div(1000)` is misleading and results in no slippage control at all.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `FarmHero` protocol. The audited system presents a unique addition to current DeFi offerings by offering a decentralized approach that mixes NFT, gaming and DeFi concepts. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.