SMART CONTRACT AUDIT REPORT

for

KaoyaSwap

Prepared By: Xiaomi Huang

PeckShield

May 5, 2022

Document Properties

Client	KaoyaSwap
Title	Smart Contract Audit Report
Target	KaoyaSwap
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2022	Shulin Bie	Final Release
1.0-rc	April 28, 2022	Shulin Bie	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 `KaoyaSwap` 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 KaoyaSwap

`KaoyaSwap` is a decentralized exchange (`DEX`), which is an evolutional improvement of `UniswapV2`. It allows liquidity providers to earn the `kaoya` token via the staking of their `LP` tokens. By doing so, liquidity providers not only earn the pool's trading fees, but also earn the `kaoya` token as reward. Additionally, it also allows the user to stake the `kaoya` token to earn rewards. With that, `KaoyaSwap` effectively improves the user's annual percentage yield (`APY`).

Table 1.1: Basic Information of KaoyaSwap

Item	Description
Target	KaoyaSwap
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the `betting.sol` contract is out of our audit scope.

- https://github.com/kaoya1125/contracts.git (8273073)

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

- https://github.com/kaoya1125/contracts.git (93ae418)

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

Likelihood

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

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) [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 1.4: 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 `KaoyaSwap` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	0	
Total	7	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key KaoyaSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Fixed
PVE-002	High	Revisited Logic Of Staking::clearUserDepositTime()	Business Logic	Fixed
PVE-003	Low	Potential Reentrancy Risk In MasterChef::deposit()	Time and State	Confirmed
PVE-004	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-005	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-006	Low	Duplicate Pool Detection And Prevention	Business Logic	Fixed
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UniswapV2Router02
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [2]

Description

In the KaoyaSwap protocol, the addLiquidity() routine (see the code snippet below) is provided to add amountADesired amount of tokenA and amountBDesired amount of tokenB into the pool as liquidity via the UniswapV2Router02::addLiquidity() routine. To elaborate, we show below the related code snippet.

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
```

```
                require(amountBOptimal >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB, uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        _transferIn(msg.sender, pair, tokenA, amountA);
        _transferIn(msg.sender, pair, tokenB, amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Listing 3.1: `UniswapV2Router02::addLiquidity()`

It comes to our attention that the `UniswapV2Router02` contract has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two sets of arguments: `amountADesired/amountBDesired` and `amountAMin/amountBMin`. The first set `amountADesired/amountBDesired` determines the desired amount for adding liquidity to the pool and the second set `amountAMin/amountBMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for certain trades on `UniswapV2Router02` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `_addLiquidity()` function.

Status The issue has been addressed in this commit: `fe8c070`.

3.2 Revisited Logic Of Staking::clearUserDepositTime()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Staking/airdrop`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Staking` contract is one of the main entries for interaction with users, which provides an incentive mechanism that rewards the deposits of the supported `stakeToken` token with the `rewardToken` token. Meanwhile, an airdrop mechanism is introduced to reward the depositors who meet the following two criteria: the lockup period of the deposit is larger than the specified `period` in the `airdrop` contract and the deposit amount is larger than the specified `threshold` in the `Staking` contract. In particular, the `clearUserDepositTime()` routine is designed to reset the user's deposit time when the user claims the airdrop reward. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the contracts. The `clearUserDepositTime()` routine is called (line 37) inside the `getAirdrop()` routine to reset the user's deposit time. However, in the `clearUserDepositTime()` routine, we notice the `result` variable is defined as `memory` rather than `storage` (line 420), which will result in the failure of the deposit time reset. With that, the depositor has capability to claim the airdrop reward repeatedly.

```
33     function getAirdrop() external {
34         UserInfo memory userInfo = IStaking(poolAddress).getUserInfo(msg.sender);
35         require(userInfo.depositTime>0,"error");
36         uint diff = block.timestamp - userInfo.depositTime;
37         IStaking(poolAddress).clearUserDepositTime(msg.sender);
38         ...
39     }
```

Listing 3.2: `airdrop::getAirdrop()`

```
418     function clearUserDepositTime(address user) public {
419         require(msg.sender==airdropContract,"can't clear");
420         UserInfo memory result = userInfo[user];
421         result.depositTime = 0;
422     }
```

Listing 3.3: `Staking::clearUserDepositTime()`

Recommendation Correct the implementation of the above-mentioned routine.

Status The issue has been addressed in this commit: `423e2e6`.

3.3 Potential Reentrancy Risk In MasterChef::deposit()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: MasterChef/Staking
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

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].

In the `MasterChef` contract, we notice the `deposit()` routine has potential reentrancy risk. To elaborate, we show below the related code snippet of the `MasterChef::deposit()` routine. In the `deposit()` routine, we notice `pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount)` (line 203) will be called to transfer the underlying assets into the `MasterChef` contract. If the `pool.lpToken` faithfully implements the ERC777-like standard, then the `deposit()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining `GasTokens`).

In our case, the above hook can be planted in `pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount)` (line 203) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `user.rewardDebt` intact (used for the calculation of pending rewards at line 200). With a lower `user.rewardDebt`, the re-entered `deposit()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains.

```
194     // Deposit LP tokens to MasterChef for kaoya allocation.
195     function deposit(uint256 _pid, uint256 _amount) public {
```

```
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accKaoyaPerShare).div(1e12).sub(user.
                rewardDebt);
            safeKaoyaTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accKaoyaPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.4: `MasterChef::deposit()`

Note that other routines, i.e., `MasterChef::withdraw()/emergencyWithdraw()`, `Staking::deposit()/withdraw()/emergencyWithdraw()`, can also benefit from the reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been confirmed by the team. The team decides to leave it as is considering there is no need to support `ERC777`-like token.

3.4 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterChef`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `KaoyaSwap` protocol provides an incentive mechanism that rewards the staking of the supported assets with the `kaoya` token. 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 the 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.

```
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
```

```
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.5: `MasterChef::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 in this commit: `ecc8359`.

3.5 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef/Staking`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `KaoyaSwap` implementation, the `MasterChef` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts the deposits of the supported assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `MasterChef` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accKaoyaPerShare).div(1e12).sub(user.
                rewardDebt);
            safeKaoyaTransfer(msg.sender, pending);
```

```
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accKaoyaPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.6: MasterChef::deposit()

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 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 deposit(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of KaoyaSwap and affects protocol-wide operation and maintenance.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the MasterChef before and after the transfer() or transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into KaoyaSwap. In KaoyaSwap protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better 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 The issue has been confirmed by the team. The team decides to leave it as is considering there is no need to support deflationary/rebasing token.

3.6 Duplicate Pool Detection And Prevention

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterChef
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The MasterChef contract provides an incentive mechanism that rewards the staking of the supported assets with the kaoya token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*multiplier/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public
        onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accKaoyaPerShare: 0
        }));
    }
```

Listing 3.7: MasterChef::add()

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

Status The issue has been addressed by the following commit: `ecc8359`.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `KaoyaSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., transfer the locked assets out of the contract). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setVault(
        address vaultAddress
    )
        external
        override
    {
        require(msg.sender == owner,'UniswapV2Router: FORBIDDEN');
        vault = vaultAddress;
    }

    function take(address token, uint amount)
        external
        virtual
        override
    {
        require(msg.sender == vault,'UniswapV2Router: FORBIDDEN');
        TransferHelper.safeTransfer(token, vault, amount);
    }
```

Listing 3.8: `UniswapV2Router02`

```
    /**
    * @dev Creates `amount` tokens and assigns them to `msg.sender`, increasing
    * the total supply.
    *
    * Requirements
    *
```

```
    * - ‘msg.sender‘ must be the token owner
    */
    function mint(uint256 amount) public onlyOwner returns (bool) {
        _mint(_msgSender(), amount);
        return true;
    }
```

Listing 3.9: `Kaoya`

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

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the KaoyaSwap design and implementation. KaoyaSwap is a decentralized exchange (DEX), which is an evolutional improvement of UniswapV2. It allows liquidity providers to earn the kaoya token via the staking of their LP tokens. By doing so, liquidity providers not only earn the pool's trading fees, but also earn the kaoya token as reward. Additionally, it also allows the user to stake the kaoya token to earn rewards. With that, KaoyaSwap effectively improves the user's annual percentage yield (APY). 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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