SMART CONTRACT AUDIT REPORT

for

KaiDex

Prepared By: Xiaomi Huang

Hangzhou, China

August 15, 2022

Document Properties

Client	KardiaChain
Title	Smart Contract Audit Report
Target	KaiDex
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 15, 2022	Jing Wang	Final Release
1.0-rc	August 1, 2022	Jing Wang	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 KaiDex protocol design document and related smart contract source code, 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 branch of KaiDex 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 KaiDex

KaiDex is a decentralized exchange with an automated market maker for the support of liquidity provision and peer-to-peer transactions on the KardiaChain. KaiDex is the first Defi-meet-DAO DEX model, bringing profitably full decentralization back to the users. KaiDex allows users to exchange tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of KaiDex

Item	Description
Name	KardiaChain
Website	https://kaidex.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 15, 2022

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

- https://github.com/kardia-solutions/kaidex-core (59f163e)

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

- https://github.com/kardia-solutions/kaidex-core (TBD)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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

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

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) [13], 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 `KaiDex` 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	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key KaiDex Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Reentrancy Risk in KaidexMasterChef	Time and State	Fixed
PVE-002	Low	Incompatibility with Deflationary Tokens	Business Logic	Fixed
PVE-003	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Fixed
PVE-004	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Medium	Possible Costly LPs From Improper StKDX Initialization	Business Logic	Fixed

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 Reentrancy Risk in KaidexMasterChef

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KaidexMasterChef`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [4]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `KaidexMasterChef` as an example, the `emergencyWithdraw()` 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 264) starts before effecting the update on the internal state (lines 265-266), 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.

```
259 // Withdraw without caring about rewards. EMERGENCY ONLY.
260 function emergencyWithdraw(uint256 _pid) public {
261     PoolInfo storage pool = poolInfo[_pid];
262     UserInfo storage user = userInfo[_pid][msg.sender];
263
```

```
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    }
```

Listing 3.1: `KaidexMasterChef::emergencyWithdraw()`

Note that other routines share the same issue, including `deposit()` and `withdraw()` from the same contract.

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

Status The issue has been mitigated by this commit: `f6d5969`.

3.2 Incompatibility with Deflationary Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KaidexMasterChef`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

In the `KaiDex` protocol, the `KaidexMasterChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works 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.

```
    function _deposit(uint256 _pid, uint256 _amount, address userAddress) internal {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][userAddress];
        updatePool(_pid);
        uint256 pending = 0;
        if (user.amount > 0) {
            pending =
                user.amount.mul(pool.accKDXPerShare).div(1e12).sub(
                    user.rewardDebt
                );
```

```
            safeKDXTransfer(userAddress, pending);
        }
        pool.lpToken.safeTransferFrom(
            address(userAddress),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accKDXPerShare).div(1e12);
        emit Deposit(userAddress, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending =
            user.amount.mul(pool.accKDXPerShare).div(1e12).sub(
                user.rewardDebt
            );
        safeKDXTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accKDXPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.2: `KaidexMasterChef::_deposit()`and `KaidexMasterChef::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, 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 the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accKDXPerShare` via dividing `kdxReward` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 191). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accKDXPerShare` as the final result, which dramatically inflates the pool's reward.

```
185     // Update reward variables of the given pool to be up-to-date.
186     function updatePool(uint256 _pid) public {
187         PoolInfo storage pool = poolInfo[_pid];
188         if (block.number <= pool.lastRewardBlock) {
189             return;
190         }
```

```
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 kdxReward =
            multiplier.mul(kdxPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint
            );
        kdx.mint(address(this), kdxReward);
        pool.accKDXPerShare = pool.accKDXPerShare.add(
            kdxReward.mul(1e12).div(lpSupply)
        );
        pool.lastRewardBlock = block.number;
        emit LogUpdatePool(_pid, pool.lastRewardBlock, lpSupply, pool.accKDXPerShare);
    }
```

Listing 3.3: `KaidexMasterChef::updatePool()`

One 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 `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` 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 `KaiDex` protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status The issue has been mitigated by this commit: `86a2f97c`.

3.3 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: KaiDexRouter
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [3]

Description

In the KaiDexRouter contract, 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 UniswapRouterV2::addLiquidity() routine. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    )
        external
        virtual
        override
        ensure(deadline)
        returns (
            uint256 amountA,
            uint256 amountB,
            uint256 liquidity
        )
    {
        (amountA, amountB) = _addLiquidity(
            tokenA,
            tokenB,
            amountADesired,
            amountBDesired,
            amountAMin,
            amountBMin
        );
        address pair = KaiDexLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IKAIDexPair(pair).mint(to);
```

```
    }
```

Listing 3.4: `KaiDexRouter::addLiquidity()`

```
    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {
        // create the pair if it doesn't exist yet
        if (IKaiDexFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IKaiDexFactory(factory).createPair(tokenA, tokenB);
        }
        (uint256 reserveA, uint256 reserveB) = KaiDexLibrary.getReserves(
            factory,
            tokenA,
            tokenB
        );
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint256 amountBOptimal = KaiDexLibrary.quote(
                amountADesired,
                reserveA,
                reserveB
            );
            if (amountBOptimal <= amountBDesired) {
                require(
                    amountBOptimal >= amountBMin,
                    "KaiDexRouter: INSUFFICIENT_B_AMOUNT"
                );
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint256 amountAOptimal = KaiDexLibrary.quote(
                    amountBDesired,
                    reserveB,
                    reserveA
                );
                assert(amountAOptimal <= amountADesired);
                require(
                    amountAOptimal >= amountAMin,
                    "KaiDexRouter: INSUFFICIENT_A_AMOUNT"
                );
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.5: `KaiDexRouter::_addLiquidity()`

It comes to our attention that the `KaiDexRouter` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` 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 some trades on `KaiDexRouter` 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 mitigated by this commit: `f6d5969`.

3.4 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `KaidexMaker`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [5]

Description

In `KaiDex`, the `KaidexMaker` contract is designed to trade tokens collected from fees for `KDX` and serves up rewards for `LP` holders. The `KaidexMaker` contract has a helper routine, i.e., `_convert()`, that is designed to swap `token0` for `token1`. It has a rather straightforward logic in removing liquidity and transferring the funds to specific pairs to actually perform the intended token swap by `_convertStep()`.

```
    function _convert(address token0, address token1) internal {
        IKAIDexPair pair = IKAIDexPair(factory.getPair(token0, token1));
        require(address(pair) != address(0), "KaidexMaker: Invalid pair");

        IERC20(address(pair)).safeTransfer(
            address(pair),
            pair.balanceOf(address(this))
        );

        (uint256 amount0, uint256 amount1) = pair.burn(address(this));
        if (token0 != pair.token0()) {
            (amount0, amount1) = (amount1, amount0);
        }
        emit LogConvert(
```

```
            msg.sender,
            token0,
            token1,
            amount0,
            amount1,
            _convertStep(token0, token1, amount0, amount1)
        );
    }

    function _convertStep(
        address token0,
        address token1,
        uint256 amount0,
        uint256 amount1
    ) internal returns (uint256 kdxOut) {
        // Interactions
        require(bar != address(0), "bar is not set");
        if (token0 == token1) {
            uint256 amount = amount0.add(amount1);
            if (token0 == kdx) {
                IERC20(kdx).safeTransfer(bar, amount);
                kdxOut = amount;
            } else if (token0 == wkai) {
                kdxOut = _toKDX(wkai, amount);
            } else {
                address bridge = bridgeFor(token0);
                amount = _swap(token0, bridge, amount, address(this));
                kdxOut = _convertStep(bridge, bridge, amount, 0);
            }
        } else if (token0 == kdx) {
            // eg. KDX - KAI
            IERC20(kdx).safeTransfer(bar, amount0);
            kdxOut = _toKDX(token1, amount1).add(amount0);
        } else if (token1 == kdx) {
            // eg. USDT - KDX
            IERC20(kdx).safeTransfer(bar, amount1);
            kdxOut = _toKDX(token0, amount0).add(amount1);
        } else if (token0 == wkai) {
            // eg. KAI - USDC
            kdxOut = _toKDX(
                wkai,
                _swap(token1, wkai, amount1, address(this)).add(amount0)
            );
        } else if (token1 == wkai) {
            // eg. USDT - KAI
            kdxOut = _toKDX(
                wkai,
                _swap(token0, wkai, amount0, address(this)).add(amount1)
            );
        } else {
            // eg. MIC - USDT
            address bridge0 = bridgeFor(token0);
```

```
            address bridge1 = bridgeFor(token1);
            if (bridge0 == token1) {
                // eg. MIC - USDT - and bridgeFor(MIC) = USDT
                kdxOut = _convertStep(
                    bridge0,
                    token1,
                    _swap(token0, bridge0, amount0, address(this)),
                    amount1
                );
            } else if (bridge1 == token0) {
                // eg. WBTC - DSD - and bridgeFor(DSD) = WBTC
                kdxOut = _convertStep(
                    token0,
                    bridge1,
                    amount0,
                    _swap(token1, bridge1, amount1, address(this))
                );
            } else {
                kdxOut = _convertStep(
                    bridge0,
                    bridge1, // eg. USDT - DSD - and bridgeFor(DSD) = WBTC
                    _swap(token0, bridge0, amount0, address(this)),
                    _swap(token1, bridge1, amount1, address(this))
                );
            }
        }
    }
```

Listing 3.6: `KaidexMaker.sol`

To elaborate, we show above related routines. We notice the remove liquidity and token swap are routed to `pair` and the actual removal or swap operation via `pair.burn(address(this))` or `_swap(token0, bridge, amount, address(this))` essentially do not specify any restriction with output amount 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 This issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [1]

Description

In the `KaiDex` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., setting various parameters, moving funds in emergency). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contract.

To elaborate, we show the `emergencyWithdraw()` routine from the `WhitelistRaising()` contract. This function allow the `owner` account to withdraw all funds from the contract.

```
    function emergencyWithdraw(address token, address payable to)
        public
        onlyOwner
    {
        if (token == address(0)) {
            to.transfer(address(this).balance);
        } else {
            ERC20(token).safeTransfer(
                to,
                ERC20(token).balanceOf(address(this))
            );
        }
    }
```

We understand the need of the privileged functions for contract maintenance, but 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 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.

3.6 Possible Costly LPs From Improper StKDX Initialization

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `StKDX`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [2]

Description

The `StKDX` contract aims to provide incentives so that users can stake and lock their funds in a stake pool. The staking users will get their pro-rata share based on their staked amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `enter()` routine. This `enter()` routine is used for participating users to deposit the supported asset (e.g., `KDXs`) and get respective shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
35 function enter(uint256 _amount) public {
36     // Gets the amount of Kdx locked in the contract
37     uint256 totalKdx = kdx.balanceOf(address(this));
38     // Gets the amount of stKDX in existence
39     uint256 totalShares = totalSupply();
40     // If no stKDX exists, mint it 1:1 to the amount put in
41     if (totalShares == 0 || totalKdx == 0) {
42         _mint(msg.sender, _amount);
43     }
44     // Calculate and mint the amount of stKDX the Kdx is worth. The ratio will change
           overtime, as stKDX is burned/minted and Kdx deposited + gained from fees /
           withdrawn.
45     else {
46         uint256 what = _amount.mul(totalShares).div(totalKdx);
47         _mint(msg.sender, what);
48     }
49     // Lock the Kdx in the contract
50     kdx.transferFrom(msg.sender, address(this), _amount);
51 }
```

Listing 3.7: `StKDX::enter()`

Specifically, when the pool is being initialized, the share value directly takes the value of `shares = amount` (line 42), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further deposit a huge amount of `KDX` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been mitigated by this commit: `f6d5969`.

4 Conclusion

In this audit, we have analyzed the `KaiDex` design and implementation. `KaiDex` is a decentralized exchange with an automated market maker for the support of liquidity provision and peer-to-peer transactions on the `KardiaChain`. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[11] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

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

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