SMART CONTRACT AUDIT REPORT

for

Klein Protocol

Prepared By: Xiaomi Huang

PeckShield

June 12, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	June 12, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 31, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Klein` 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 Klein

`Klein` is a decentralised DAO protocol that is inspired from the `Curve`'s DAO implementation. The longer the locking period, the more voting weights one gets. Moreover, it is unique in having a built-in `NFT`-based voting mechanism and associated token emissions. In essence, the voting weights are tokenized as `veNFT`, which can then be used to vote and eventually decide how the rewards will be distributed. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Klein

Item	Description
Target	Klein
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 12, 2022

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

- https://github.com/foxdex/klein.git (0ccc5ab)

- https://github.com/foxdex/exchange.git (3719c15)

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

- https://github.com/foxdex/klein.git (e8a998d)
- https://github.com/foxdex/exchange.git (3719c15)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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 [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: 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 implementation of the Klein protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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

Table 2.1: Key Klein Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Pool Weighting in AbstractController::_reset()	Business Logic	Resolved
PVE-002	High	Timely Reward Update in AbstractController::updatePool()	Business Logic	Confirmed
PVE-003	Low	Proper SetOperatorContract Event Generation in CheckPermission	Coding Practices	Resolved
PVE-004	High	Abused AbstractController::poke() For Voting Manipulation	Security Features	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides 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 Improper Pool Weighting in AbstractController::_reset()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `AbstractController`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Klein` protocol has an `AbstractController` contract that is inherited by `GaugeController` and `SwapController` for built-in control and voting logic. While analyzing the voting-related reset logic, we notice the current implementation needs to be improved.

For elaboration, we show below the related `reset()` function. As the name indicates, this function is used to reset the previous voting by undoing the used weight (lines 72 – 73) associated with the given `tokenId`. However, it comes to our attention that the used weight also needs to be removed from the applied pool, i.e., `weights[userPool[tokenId].pool] -= _totalWeight`. An incorrect accounting of pool weights may bring detrimental effect on the developed reward dissemination.

```
59     function reset(uint256 _tokenId) external {
60         require(IVeToken(veToken).isApprovedOrOwner(msg.sender, _tokenId));
61         PoolVote storage poolVote = userPool[_tokenId];
62         require(poolVote.lastUse + duration < block.timestamp, "next duration use");
63         _reset(_tokenId);
64         IVeToken(veToken).abstain(_tokenId);
65         poolVote.lastUse = block.timestamp;
66         updatePool();
67     }
68
69     function _reset(uint256 _tokenId) internal {
70         uint256 _totalWeight = usedWeights[_tokenId];
71         emit Abstained(_tokenId, _totalWeight);
72         totalWeight -= _totalWeight;
73         usedWeights[_tokenId] = 0;
```

```
        delete userPool[_tokenId];
    }
```

Listing 3.1: `AbstractController::reset()`

Recommendation Properly maintain the accounting of `tokenId`-associated weights when it is not used for voting.

Status This issue has been fixed in this commit: `91f45e4`.

3.2 Timely Reward Update in AbstractController::updatePool()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The reward governance in `Klein` is largely supported by a core `AbstractController` contract, which regulates the voting on supported pools. While reviewing the rewards-related logic, we notice the current implementation needs to be improved.

To elaborate, we show below a representative function `updatePool()` in `AbstractController`. It implements a simplistic logic in allowing for updating the pools for reward distribution. However, it immediately makes use of the new pool weights even for a possible time period in the past. A more accurate approach requires the use of the old pool weights up to the current moment and then applies the new pool weights for the time ahead.

```
    function updatePool() public {
        if (block.timestamp < lastUpdate.add(duration)) {
            return;
        }
        for (uint256 pid = 0; pid < getPoolLength(); ++pid) {
            address pool = EnumerableSet.at(_poolInfo, pid);
            uint256 _id = IDistribute(distribute).lpOfPid(pool);
            IDistribute(distribute).set(_id, weights[pool], false);
        }
        IDistribute(distribute).massUpdatePools();
        lastUpdate = block.timestamp;
    }
```

Listing 3.2: `AbstractController::updatePool`

This issue also affects a number of other routines, including the `set()` function in both `Boost` and `SwapMining` contracts, the `setTokenPerBlock()` function in `TokenReward`, as well as the `notifyRewardAmount()` function in `Gauge`.

Recommendation Revise the above functions to timely and properly apply the reward updates.

Status This issue has been confirmed.

3.3 Proper SetOperatorContract Event Generation in CheckPermission

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CheckPermission`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `CheckPermission` contract as an example. This contract is designed to configure the new `operatable` contract. While examining the event that reflects its change, we notice the emitted event does not contain accurate information about the old `operatable` contract. Specifically, the current event accidentally uses the new `operatable` contract as the old one (line 45).

```
43     function setOperContract(address _oper) public onlyOwner {
44         require(_oper != address(0), "bad new operator");
45         address oldOperator = _oper;
46         operatable = Operatable(_oper);
47         emit SetOperatorContract(oldOperator, _oper);
48     }
```

Listing 3.3: `CheckPermission::SetOperatorContract()`

Recommendation Properly emit the `SetOperatorContract` event with accurate information.

Status This issue has been fixed in this commit: `91f45e4`.

3.4 Abused AbstractController::poke() For Voting Manipulation

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: AbstractController
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

As mentioned earlier, the reward governance in Klein is largely supported by a core AbstractController contract, which regulates the voting on supported pools. While reviewing the voting-related logic, we notice the current poke() implementation can be abused to manipulate the accounting of pool votes.

To elaborate, we show below its implementation. Note that this function is designed to update the weight associated with a given tokenId for the intended pool. It comes to our attention that this function is permissionless and both input arguments of tokenId andpool are not validated before their use. As a result, the weight of any existing pool can be simply manipulated to influence its vote and hence its reward distribution!

```
    function _vote(uint256 _tokenId, address _poolVote) internal {
        _reset(_tokenId);
        uint256 _weight = IVeToken(veToken).balanceOfNFT(_tokenId);

        weights[_poolVote] = weights[_poolVote].add(_weight);
        emit Voted(msg.sender, _tokenId, _weight);
        IVeToken(veToken).voting(_tokenId);
        totalWeight += _weight;
        usedWeights[_tokenId] = _weight;
        updatePool();
    }

    function poke(uint256 _tokenId, address _pool) external {
        _vote(_tokenId, _pool);
    }
```

Listing 3.4: AbstractController::poke()

Recommendation Strengthen the sanity checks before applying the requested updates on the pool vote. Specifically, the calling user needs to be the owner of the given tokenId or the tokenId is indeed voted for the given pool.

Status This issue has been fixed in this commit: 91f45e4.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Klein protocol, there are privileged accounts (owner and operator) that play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and mining pool adjustment). 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 account and their related privileged accesses in current contracts.

```
    function addPair(
        uint256 _allocPoint,
        address _pool,
        bool _withUpdate
    ) public onlyOperator {
        require(_pool != address(0), "_pair is the zero address");
        if (poolLength() > 0) {
            require((lpOfPid[_pool] == 0) && (address(poolInfo[0].pair) != _pool), "only
                one pair");
        }
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                pair: _pool,
                quantity: 0,
                allocPoint: _allocPoint,
                allocSwapTokenAmount: 0,
                lastRewardBlock: lastRewardBlock
            })
        );
        lpOfPid[_pool] = poolLength() - 1;
        emit AddPool(_pool, _allocPoint);
    }

    // Update the allocPoint of the pool
    function set(
        uint256 _pid,
        uint256 _allocPoint,
```

```
        bool _withUpdate
    ) public {
        require(controllers[msg.sender]  msg.sender == operator(), "no auth");
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
        if (_withUpdate) {
            massUpdatePools();
        }
        emit SetPool(poolInfo[_pid].pair, _allocPoint);
    }

    function setRouter(address newRouter) public onlyOperator {
        require(newRouter != address(0), "SwapMining: new router is the zero address");
        address oldRouter = router;
        router = newRouter;
        emit ChangeRouter(oldRouter, router);
    }
```

Listing 3.5: Example `Setters` in the `SwapMining`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated. The team decides to use multi-sig contract for the privileged `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Klein` protocol, which is a decentralised DAO protocol that is inspired from the `Curve`'s DAO. The longer the locking period, the more voting weights one gets. Moreover, it is unique in having a built-in `NFT`-based voting mechanism and associated token emissions. In essence, the voting weights are tokenized as `veNFT`, which can then be used to vote and eventually decide how the rewards will be distributed. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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