SMART CONTRACT AUDIT REPORT

for

SHEEPDEX

Prepared By: Yiqun Chen

PeckShield
November 4, 2021

Document Properties

Client	SheepDEX
Title	Smart Contract Audit Report
Target	SheepDEX
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 4, 2021	Xiaotao Wu	Final Release
1.0-rc	November 2, 2021	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

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

`SheepDEX` is designed on top of the popular `UniswapV3` protocol with additional extensions to capitalize on the concentrated liquidity feature and integrate new incentive mechanisms, e.g., `liquidity mining` and `swap mining`, to engage community and user base. The protocol helps liquidity providers (LPs) and traders to maximize capital efficiency and earnings by introducing several innovative features that are not yet available on other `DEXs` in the BSC ecosystem, such as concentrated liquidity, multiple fee tiers, range orders and a triple-incentive mechanism.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of SheepDEX

Item	Description
Name	SheepDEX
Website	https://sheepdex.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 4, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Specifically, this audit mainly focuses on the extensions for `swap mining` and `liquidity` `mining`. Moreover, the contract of `RewardPool` is not in the scope of this audit (and the team is still preparing for the launch of the `staking` functionality).

- https://github.com/foxdex/spestaker (95d9a22)
- https://github.com/foxdex/spcore (4847bc4)
- https://github.com/foxdex/spperiphery (7685517)

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 accordingly classified into four categories, 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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.

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.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	1	■
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 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, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Confirmed
PVE-002	Low	Incorrect fixedQuantity Calculation In SPCTimeLock::reset()	Business Logic	Confirmed
PVE-003	Low	Duplicate Pool Detection And Prevention	Business Logic	Confirmed
PVE-004	Low	Improved Logic In PositionReward::transferDeposit()	Business Logic	Confirmed
PVE-005	Low	Improved Logic In PositionReward::_stakeToken()	Business Logic	Confirmed
PVE-006	Informational	Meaningful Events For Important State Changes	Coding Practices	Confirmed
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 Timely massUpdatePools During Pool Weight Changes

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

Description

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

```
    // Update the allocPoint of the pool
    function setPair(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOperator {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
        emit SetPool(poolInfo[_pid].pair, _allocPoint);
    }
```

Listing 3.1: `SwapMining::setPair()`

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 operator (via the `onlyOperator` modifier), which greatly alleviates the concern. Note a number of other routines can be similarly improved, including `PositionReward::set()` and `TokenReward::setTokenPerBlock()`.

In addition, we note that there is a lack of `massUpdatePools()` in the `createIncentive()` routine of the `PositionReward` contract and in the `reduceBlockReward()` modifier of the `TokenReward` contract.

```
    function createIncentive(IncentiveKey memory key, uint256 point) external
        onlyOperator {
        require(
            block.timestamp <= key.startTime,
            'PositionReward::createIncentive: start time must be now or in the future'
        );
        bytes32 incentiveId = PoolId.compute(key);
        totalAllocPoint = totalAllocPoint.add(point);
        incentives[incentiveId].allocPoint = point;
        incentives[incentiveId].lastRewardBlock = block.number;
        incentiveKeys.push(key);
        emit IncentiveCreated(key.rewardToken, key.pool, key.startTime, point);
    }
```

Listing 3.2: `PositionReward::createIncentive()`

```
    modifier reduceBlockReward() {
        if (block.number > startBlock && block.number >= periodEndBlock) {
            if (tokenPerBlock > minTokenReward) {
                tokenPerBlock = tokenPerBlock.mul(80).div(100);
            }
            if (tokenPerBlock < minTokenReward) {
                tokenPerBlock = minTokenReward;
            }
            periodEndBlock = block.number.add(period);
        }
        _;
    }
```

Listing 3.3: `TokenReward::reduceBlockReward()`

Recommendation Timely invoke `massUpdatePools()` when any pool's weight or `tokenPerBlock` has been updated.

Status This issue has been confirmed. The team acknowledges the need of always setting `_withUpdate` to true or manually calling `massUpdatePools()` if these privileged functions are being called. Regarding the `reduceBlockReward` modifier, there is still a need for the `SheepDEX` team to find a proper solution to fix the possible unfair distribution of rewards before the current `periodEndBlock`.

3.2 Incorrect fixedQuantity Calculation In SPCTimeLock::reset()

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

Description

The `SPCTimeLock` contract implements a privileged function `reset()` for the `operator` to reset the number of reward tokens distributed per cycle, i.e., `fixedQuantity`. Our analysis with this routine shows its current implementation is not correct.

To elaborate, we show below the `reset()` routine. According to the design, the whole reward distribution is divided into 48 cycles and each cycle with a period of 864000 blocks. The correct calculation for the `fixedQuantity` should be `token.balanceOf(address(this)).div(cycleTimes - cycle)` if `cycle < cycleTimes`, instead of current `token.balanceOf(address(this)).div(period)` (line 61).

```
60     function reset() onlyOperator external {
61         fixedQuantity = token.balanceOf(address(this)).div(period);
62     }
```

Listing 3.4: `SPCTimeLock::reset()`

Recommendation Revise the above `reset()` to properly compute the right reward distributing amount.

Status This issue has been confirmed.

3.3 Duplicate Pool Detection And Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `PositionReward`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `SheepDEX` protocol provides an incentive mechanism that rewards the staking of `SheepDEX ERC721` liquidity positions with the `swap` token. The rewards are carried out by designating a number of staking pools into which ERC721 liquidity positions can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards.

In current implementation, there are a number of concurrent pools that share the rewarded swap tokens and more can be scheduled for addition (via 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 createIncentive(), whose code logic is shown below. It turns out it does not perform necessary sanity checks in preventing a new pool with a duplicate incentiveId from being added. Though it is a privileged interface (protected with the modifier onlyOperator), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function createIncentive(IncentiveKey memory key, uint256 point) external
        onlyOperator {
        require(
            block.timestamp <= key.startTime,
            'PositionReward::createIncentive: start time must be now or in the future'
        );
        bytes32 incentiveId = PoolId.compute(key);
        totalAllocPoint = totalAllocPoint.add(point);
        incentives[incentiveId].allocPoint = point;
        incentives[incentiveId].lastRewardBlock = block.number;
        incentiveKeys.push(key);
        emit IncentiveCreated(key.rewardToken, key.pool, key.startTime, point);
    }
```

Listing 3.5: PositionReward::createIncentive()

We point out that if a new pool with a duplicate incentiveId can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

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 This issue has been confirmed.

3.4 Improved Logic In PositionReward::transferDeposit()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PositionReward`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PositionReward` contract of `SheepDEX` protocol provides an external `transferDeposit()` function for user to transfer a deposited `ERC721` liquidity position token to a new owner. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that when a deposited `ERC721` liquidity position token is transferred, the updating of state variables `_holderTokens[owner]` and `_holderTokens[to]` (lines 147-148) are not necessary if this position token is not staked.

```
143     function transferDeposit(uint256 tokenId, address to) external {
144         require(to != address(0), 'PositionReward::transferDeposit: invalid transfer
                recipient');
145         address owner = deposits[tokenId].owner;
146         require(owner == msg.sender, 'PositionReward::transferDeposit: can only be
                called by deposit owner');
147         _holderTokens[owner].remove(tokenId);
148         _holderTokens[to].add(tokenId);
149         deposits[tokenId].owner = to;
150         emit DepositTransferred(tokenId, owner, to);
151     }
```

Listing 3.6: `PositionReward::transferDeposit()`

Recommendation Update the state variables `_holderTokens[owner]` and `_holderTokens[to]` only if the transferred `ERC721` liquidity position token is staked.

Status This issue has been confirmed.

3.5 Improved Logic In PositionReward::_stakeToken()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PositionReward
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The SheepDEX protocol users can deposit their SheepDEX ERC721 liquidity position tokens to the PositionReward contract by directly transferring their tokens to the contract. The deposited tokens can be further staked to designated pools to earn rewards. While examining the _stakeToken routine of the PositionReward contract, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that when staking ERC721 liquidity position token to the PositionReward contract, it requires incentives[incentiveId].totalRewardUnclaimed > 0 (lines 262-265). However, there exists the possibility that incentives[incentiveId].totalRewardUnclaimed == 0 due to the calling of unstakeToken() by another user.

```
256     function _stakeToken(IncentiveKey memory key, uint256 tokenId) private {
257         require(block.timestamp >= key.startTime, 'PositionReward::stakeToken: incentive not started');
258 
259         bytes32 incentiveId = PoolId.compute(key);
260         updatePool(incentiveId);
261 
262         require(
263             incentives[incentiveId].totalRewardUnclaimed > 0,
264             'PositionReward::stakeToken: non-existent incentive'
265         );
266         require(
267             _stakes[tokenId][incentiveId].liquidityNoOverflow == 0,
268             'PositionReward::stakeToken: token already staked'
269         );
270 
271         (ISpePool pool, int24 tickLower, int24 tickUpper, uint128 liquidity) =
272         NFTPositionInfo.getPositionInfo(factory, nonfungiblePositionManager, tokenId);
273 
274         require(pool == key.pool, 'PositionReward::stakeToken: token pool is not the incentive pool');
275         require(liquidity > 0, 'PositionReward::stakeToken: cannot stake token with 0 liquidity');
276 
277         _holderTokens[deposits[tokenId].owner].add(tokenId);
278         deposits[tokenId].numberOfStakes++;
279         incentives[incentiveId].numberOfStakes++;
280 
```

```
        (, uint160 secondsPerLiquidityInsideX128,) = pool.snapshotCumulativesInside(
            tickLower, tickUpper);

        if (liquidity >= type(uint96).max) {
            _stakes[tokenId][incentiveId] = Stake({
            secondsPerLiquidityInsideInitialX128 : secondsPerLiquidityInsideX128,
            liquidityNoOverflow : type(uint96).max,
            liquidityIfOverflow : liquidity
            });
        } else {
            Stake storage stake = _stakes[tokenId][incentiveId];
            stake.secondsPerLiquidityInsideInitialX128 = secondsPerLiquidityInsideX128;
            stake.liquidityNoOverflow = uint96(liquidity);
        }

        emit TokenStaked(tokenId, incentiveId, liquidity);
    }
```

Listing 3.7: `PositionReward::_stakeToken()`

Recommendation Take into consideration the scenario where the value of `incentives[incentiveId].totalRewardUnclaimed` might be equal to 0.

Status This issue has been confirmed. The team confirms that this will not affect current business logic.

3.6 Meaningful Events For Important State Changes

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenReward`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

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 `TokenReward` contract as an example. While examining the events that reflect the `TokenReward` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `setHalvingPeriod()/setMintPeriod()/setMinTokenReward`

`()/setTokenPerBlock()` functions are being called, there are no corresponding events being emitted to reflect the occurrence of `setHalvingPeriod()/setMintPeriod()/setMinTokenReward()/setTokenPerBlock()`.

```
    function setHalvingPeriod(uint256 _block) public onlyOperator {
        period = _block;
    }

    function setMintPeriod(uint256 _block) public onlyOperator {
        mintPeriod = _block;
    }

    function setMinTokenReward(uint256 _reward) public onlyOperator {
        minTokenReward = _reward;
    }

    // Set the number of swap produced by each block
    function setTokenPerBlock(uint256 _newPerBlock, bool _withUpdate) public
        onlyOperator {
        if (_withUpdate) {
            massUpdatePools();
        }
        tokenPerBlock = _newPerBlock;
    }
```

Listing 3.8: `TokenReward::setHalvingPeriod()/setMintPeriod()/setMinTokenReward()/setTokenPerBlock()`

Recommendation Properly emit the related events when the above-mentioned functions are being invoked.

Status This issue has been confirmed.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `SheepDEX` protocol, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

Firstly, the privileged functions in the `SPCToken` contract allow for the the `operator` to add/delete minters and salvage `ERC20` tokens send to the contract by mistake.

```
    function addMinter(address _minter) public onlyOperator returns (bool) {
        require(_minter != address(0), ': _addMinter is the zero address');
        emit AddMinter(_minter);
        return EnumerableSet.add(_minters, _minter);
    }

    function delMinter(address _minter) public onlyOperator returns (bool) {
        require(_minter != address(0), ': _delMinter is the zero address');
        emit DelMinter(_minter);
        return EnumerableSet.remove(_minters, _minter);
    }

    function salvageToken(address reserve) external onlyOperator {
        uint256 amount = IERC20(reserve).balanceOf(address(this));
        TransferHelper.safeTransfer(reserve, operator(), amount);
    }
```

Listing 3.9: `SPCToken::addMinter()/delMinter()/salvageToken()`

It should be noted that the minters can mint more tokens into circulation for specified accounts (line 69).

```
65      function mint(address _to, uint256 _amount) public onlyMinter returns (bool) {
66          if (_amount.add(totalSupply()) > MAX_SUPPLY) {
67              return false;
68          }
69          _mint(_to, _amount);
70          return true;
71      }
```

Listing 3.10: `SPCToken::mint()`

Secondly, the privileged functions in the `TokenReward` contract allow for the `operator` to configure key parameters for the contract. These parameters include `period`, `mintPeriod`, `minTokenReward`, and `tokenPerBlock`.

```
    function setHalvingPeriod(uint256 _block) public onlyOperator {
        period = _block;
    }

    function setMintPeriod(uint256 _block) public onlyOperator {
        mintPeriod = _block;
    }

    function setMinTokenReward(uint256 _reward) public onlyOperator {
        minTokenReward = _reward;
    }

    // Set the number of swap produced by each block
```

```
    function setTokenPerBlock(uint256 _newPerBlock, bool _withUpdate) public
        onlyOperator {
        if (_withUpdate) {
            massUpdatePools();
        }
        tokenPerBlock = _newPerBlock;
    }
```

Listing 3.11: `TokenReward::setHalvingPeriod()/setMintPeriod()/setMinTokenReward()/setTokenPerBlock()`

Thirdly, the `createIncentive()` and `set()` functions in the `PositionReward` contract allow for the `operator` to create new incentive pools or adjust the `allocPoint` for existing incentive pools.

```
    function createIncentive(IncentiveKey memory key, uint256 point) external
        onlyOperator {
        require(
            block.timestamp <= key.startTime,
            'PositionReward::createIncentive: start time must be now or in the future'
        );
        bytes32 incentiveId = PoolId.compute(key);
        totalAllocPoint = totalAllocPoint.add(point);
        incentives[incentiveId].allocPoint = point;
        incentives[incentiveId].lastRewardBlock = block.number;
        incentiveKeys.push(key);
        emit IncentiveCreated(key.rewardToken, key.pool, key.startTime, point);
    }

    function set(
        IncentiveKey memory key,
        uint256 point,
        bool updateAll
    ) public onlyOperator {
        if (updateAll) {
            massUpdatePools();
        }
        bytes32 incentiveId = PoolId.compute(key);
        totalAllocPoint = totalAllocPoint.sub(incentives[incentiveId].allocPoint).add(
            point);
        incentives[incentiveId].allocPoint = point;
    }
```

Listing 3.12: `PositionReward::createIncentive()/set()`

Fourthly, the `addPair()`, `setPair()`, and `setRouter()` functions in the `SwapMining` contract allow for the `operator` to add a new pool, update the the `allocPoint` of a specified pool or set the `router` address for the `SwapMining` contract. Note the `swap()` function of the `SwapMining` contract can only be called by the `router`.

```
    function addPair(
        uint256 _allocPoint,
        address _pool,
        bool _withUpdate
```

```
    ) public onlyOperator {
        require(_pool != address(0), '_pair is the zero address');
        if (poolLength() > 0) {
            require((pairOfPid[_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,
        totalQuantity : 0,
        allocPoint : _allocPoint,
        allocSwapTokenAmount : 0,
        lastRewardBlock : lastRewardBlock
        })
        );
        pairOfPid[_pool] = poolLength() - 1;
        emit AddPool(_pool, _allocPoint);
    }

    // Update the allocPoint of the pool
    function setPair(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOperator {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
        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);
    }

    // swapMining only router
    function swap(
        address account,
        address pair,
```

```
        address input,
        address output,
        uint256 amountIn,
        uint256 amountOut
    ) public override onlyRouter returns (bool) {
        require(account != address(0), 'SwapMining: taker swap account is the zero address');
        require(input != address(0), 'SwapMining: taker swap input is the zero address');
        require(output != address(0), 'SwapMining: taker swap output is the zero address');
        require(pair != address(0), 'SwapMining: taker swap pair is the zero address');

        if (poolLength() == 0) {
            return false;
        }
        uint256 _pid = pairOfPid[pair];
        PoolInfo storage pool = poolInfo[_pid];
        // If it does not exist or the allocPoint is 0 then return
        if (pool.pair != pair   pool.allocPoint <= 0) {
            return false;
        }

        updatePool(_pid);
        uint256 quantity = getQuantity(pair, input, output, amountIn, amountOut);
        if (quantity == 0) {
            return false;
        }

        pool.quantity = pool.quantity.add(quantity);
        pool.totalQuantity = pool.totalQuantity.add(quantity);
        UserInfo storage user = userInfo[pairOfPid[pair]][account];
        user.quantity = user.quantity.add(quantity);
        user.blockNumber = block.number;
        emit SwapMining(account, pair, input, output, amountIn, amountOut);
        return true;
    }
```

Listing 3.13: `SwapMining::createIncentive()/set()`

Fifthly, the `reset()`, `release()`, and `salvageToken()` functions in the `SPCTimeLock` contract allow for the `operator` to reset the number of reward tokens distributed per cycle, release the reward tokens to `operator`, and salvage other `ERC20` tokens send to the contract by mistake.

```
    function reset() onlyOperator external {
        fixedQuantity = token.balanceOf(address(this)).div(period);
    }

    function release() onlyOperator external {
        uint reward = getReward();
        uint pCycle = currentCycle();
        cycle = pCycle >= cycleTimes ? cycleTimes : pCycle;
```

```
        rewarded = rewarded.add(reward);
        token.safeTransfer(operator(), reward);
        emit WithDraw(msg.sender, operator(), reward);
    }

    function salvageToken(address _asset) onlyOperator external returns (uint256 balance
        ) {
        require(_asset != address(token), 'no token');
        balance = IERC20(_asset).balanceOf(address(this));
        TransferHelper.safeTransfer(_asset, operator(), balance);
    }
```

Listing 3.14: `SPCTimeLock::reset()/release()/salvageToken`

Lastly, the `setSwapMining()` and `salvageToken()` functions in the `SwapRouter` contract allow for the `operator` to set the `swapMining` contract address and salvage `ERC20` tokens send to the contract by mistake.

```
    // address(0) means no swap mining
    function setSwapMining(address addr) public onlyOperator {
        address oldSwapMining = swapMining;
        swapMining = addr;
        emit ChangeSwapMining(oldSwapMining, swapMining);
    }

    function salvageToken(address _asset) onlyOperator external returns (uint256 balance
        ) {
        balance = IERC20(_asset).balanceOf(address(this));
        TransferHelper.safeTransfer(_asset, operator(), balance);
    }
```

Listing 3.15: `SwapRouter::setSwapMining()/salvageToken()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `operator/minter` 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 list of extra privileges granted to `operator/minter` explicit to `SheepDEX` protocol users.

Status This issue has been confirmed. The team confirms that `DAO` will be used in the future to solve this trust issue of admin keys by `multi-sig`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SheepDEX` protocol, which is developed on top of the popular `UniswapV3` protocol with additional extensions to capitalize on the concentrated liquidity feature and support unique incentive mechanisms. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.