Security Audit Report

DAOSquare DKPool

SECBIT

September 2, 2021

1. Introduction

The DKPool is DAOSquare's exploration of the use of NFT to prove user contributions. SECBIT Labs conducted an audit from August 16th to September 2nd, 2021, including an analysis of the contract in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the DAOSquare DKPool contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Gas Optimization	4.3.1 Avoid using redundant arithmetic security checks to save gas.	Info	Fixed
Design & Implementation	4.3.2 In `DKPool` contracts, stricter constraints need to be provided for `updatePoolMaster()` and `updatePoolMaxStake()` functions.	Info	Fixed
Design & Implementation	4.3.3 Changing the `supportToken` address in `DKPool` contract can lead to confusion in the management of funds.	Medium	Fixed
Design & Implementation	4.3.4 Illegal argument of `id` is not detected by the modifier function `poolExists()`.	Info	Fixed
Discussion	4.3.5 The intention of the `addCard()` function in the `DKPool` contract is unclear.	Info	Fixed

Design & Implementation	4.3.6 In `addCheckContract()` and `addRequireNFTs()` functions, the constraints on the input parameter card need to be added.	Info	Fixed
Design & Implementation	4.3.7 In `redeem()` function, it may failed to transfer `feeToken` by the `transferFrom()` function.	Medium	Fixed
Gas Optimization	4.3.8 Using external function instead of public function can save gas.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the DAOSquare DKPool contract is shown below:

- Project website
  - https://www.daosquare.io/
- Smart contract code
  - https://github.com/DAOSquare/DKPoolContract
  - initial review commit *fd8257f*
  - final review commit *d541d7b*

2.2 Contract List

The following content shows the contracts included in the DAOSquare DKPool project:

Name	Lines	Description
CreatorWhitelistRole.sol	32	A contract to manage the whitelist address.
DKPool.sol	752	This contract is the core part of the project and enables the exchange of NFT tokens.
Erc1155Tradable.sol	223	Auxiliary contract for minting and managing NFT tokens.
ICheck.sol	9	An interface contract with additional checks on the redemption conditions.
IErc1155.sol	51	The interface contract corresponding to the `Erc1155Tradable.sol` contract
NFTs.sol	28	A contract used to initialize the NFT token.
PoolTokenWrapper.sol	47	Ancillary contracts that enable users to access their principal.
Roles.sol	33	Library for managing addresses assigned to a Role.
TransferWhitelistRole.sol	32	A contract to manage the whitelist address.

3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in the protocol, namely Governance Account and Common Account.

- Governance Account
    - Description Contract administrator and individual pool administrator
    - Authority
        - Update basic parameters
        - Create NFT token and pool
        - Manage donated funds
    - Method of Authorization The contract administrator is the creator of the contract or authorized by the transferring of governance account. Individual pool administrator is authorized by the contract administrator or the original pool administrator.
- Common Account
    - Description Users stake or donate funds and receive corresponding DKP points. These points are redeemed for the NFT token (card).
    - Authority
        - Deposit or donate funds to obtain the corresponding DKP points.
        - Redeem points for NFT token (card).
    - Method of Authorization No authorization required

3.2 Functional Analysis

The DAOSquare DKPool is a permissionless community contribution protocol that allows users to redeem NFT tokens. We can divide the critical functions of the auction contract into several parts:

DKPool

Users deposit or donate specified funds by this contract to receive the corresponding DKP. Users can redeem NFT tokens (cards) by consuming a certain amount of DKP.

The main functions in `DKPool` are as below:

- `createPool()`

  The contract administrator uses this function to create a fund pool and also to set the initialization parameters. These pools are used to store different cards. Users can obtain DKP by depositing or donating funds into the pools. DKP is used to redeem cards in the pool.

- `createCard()`

  The administrator adds an NFT token (card) to an existing pool with this function.

- `updateCard()`

  The administrator updates the basic parameters of the existing pool.

- `addExpandPools()`

  Add an existing NFT token (card) to an existing pool.

- `stake()`

  By depositing support tokens into a specified pool , the user receives the corresponding DKP reward.

- `withdraw()`

  The user withdraws the support token deposited.

- `transfer()`

Users can transfer funds in different pools if the support token is the same.

- `redeem()`

  Users use DKP points to redeem NFT tokens (cards).

- `donate()`

  Users can receive DKP points directly by donating support tokens to a specified pool. The donated funds cannot be retrieved.

- `withdrawDonate()`

  The administrator uses this function to withdraw the support token donated by the user.

- `withdrawFee()`

  The administrator uses this function to withdraw the fee when users redeem a card.

Erc1155Tradable

This contract is used for the issuance and management of NFT tokens (cards).

The main functions in `Erc1155Tradable` are as below:

- `create()`

  This function limits the number of NFT tokens that can be supplied with the specified id.

- `mint()`

  This function mints the NFT token(card) for the specified user.

PoolTokenWrapper

This contract acts as a ledger to record the status of the user's funds. This contract is inherited from the `DKPool` contract.

The main functions in `PoolTokenWrapper` are as below:

- `stake()`

The user uses this function for staking support tokens in the specified pool.

- `withdraw()`

  The user calls this function to withdraw the support token deposited.

- `transfer()`

  With the same support token, users can transfer funds between different pools.

4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into two types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓

4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓

21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Avoid using redundant arithmetic security checks to save gas.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

In solidity 0.8.0 and later versions, security checks are applied to arithmetic operations by default; see https://blog.soliditylang.org/2020/12/16/solidity-v0.8.0-release-announcement/. The DAOSquare DKPool protocol uses a version of solidity code no lower than 0.8.0, so there is no need to use `SafeMath` library functions to perform overflow checks on arithmetic operations. It will increase additional user's gas consumption to call `SafeMath` functions. Some of the code that uses this function is listed below.

```
function earned(address account, uint256 pool)
        public
        view
        returns (uint256)
    {
        ......

        return
            balanceOf(account, pool)

.mul(blockTime.sub(lastUpdateTime).mul(p.rewardRate))
                .div(p.decimals)
                .add(p.DKP[account]);
    }
```

```
function donate(uint256 pool, uint256 amount)
        public
        whenNotPaused
        whenPoolNotPaused(pool)
        updateReward(msg.sender, pool)
    {
        ......
        uint256 DKP =
amount.mul(p.exchangeRate).div(p.decimals);
        p.donateCollected = p.donateCollected.add(amount);

        p.totalDonate[msg.sender] =
p.totalDonate[msg.sender].add(amount);
        p.DKP[msg.sender] = p.DKP[msg.sender].add(DKP);

        ......
    }
```

Status

The development team adopts our advice and removed the `SafeMath` library in commit de9cddc and b21c9f2.

4.3.2 In `DKPool` contracts, stricter constraints need to be provided for `updatePoolMaster()` and `updatePoolMaxStake()` functions.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

The current pool or contract manager can change the pool manager address with the `updatePoolMaster()` function. However, the current code does not determine the pool's existence, and the contract administrator may modify the `poolMaster` of a pool that does not currently exist. From the point of view of code logic tightness, adding the modifier function `poolExists()` is recommended. In this way, the function can only modify the `poolMaster` of a pool already created.

Similarly, the `updatePoolMaxStake()` function is used to modify the maximum number of `support tokens` that each user is allowed to deposit into the `DKPool` contract. It is also recommended to add `poolExists()` modifier function .

```
function updatePoolMaster(uint256 pool, address poolMaster)
public {
        ......
    }

function updatePoolMaxStake(uint256 pool, uint256 maxStake)
        public
        onlyOwner
    {
        ......
    }
```

Suggestion

One recommended modification is as follows:

```
function updatePoolMaster(uint256 pool, address poolMaster)
         public
         poolExists(pool)
{
        ......
}

function updatePoolMaxStake(uint256 pool, uint256 maxStake)
         public
```

```
        poolExists(pool)
        onlyOwner
    {
        ......
    }
```

Status

The development team adopts our advice and adds the modifier function in commit 6ed6950.

4.3.3 Changing the `supportToken` address in the `DKPool` contract can lead to confusion in the management of funds.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Implementation logic	Fixed

Description

The `updateSupportToken()` function is used to modify the `supportToken` parameter of a pool that currently exists. Consider the following scenario: Suppose a pool is created with a `supportToken` of DAI token, and userA deposits 100 DAI tokens into the contract via the `stake()` function. Then, The administrator executes the `updateSupportToken()` function to update the DAI token to the WETH token. Suppose there are enough WETH tokens in the pool. In that case, userA will call the `withdraw()` function to take 100 WETH tokens, which will seriously compromise the security of the funds.

```
function updateSupportToken(uint256 pool, address
tokenAddress)
        public
        poolExists(pool)
    {
        require(
            isPoolMaster(pool) || isOwner(),
```

```
            "Ownable: caller is not the owner or pool
poolMaster"
        );

        Pool storage p = pools[pool];
        p.supportToken = tokenAddress;

        _updateSupportToken(pool, tokenAddress);

        emit SupportTokenUpdated(pool, tokenAddress);
    }

function stake(uint256 pool, uint256 amount)
        public
        override
        poolExists(pool)
        updateReward(msg.sender, pool)
        whenNotPaused
        whenPoolNotPaused(pool)
    {
        Pool storage p = pools[pool];

        require(block.timestamp >= p.periodStart, "pool not
open");
        require(p.rewardRate > 0, "you can't stake when
rewardRate equals 0");
        require(
            amount.add(balanceOf(msg.sender, pool)) <=
p.maxStake,
            "stake exceeds max"
        );

        // @audit deposit funds
        super.stake(pool, amount);
        emit Staked(msg.sender, pool, amount);
    }

function withdraw(uint256 pool, uint256 amount)
        public
```

```
        override
        poolExists(pool)
        updateReward(msg.sender, pool)
    {
        require(amount > 0, "cannot withdraw 0");

        super.withdraw(pool, amount);
        emit Withdrawn(msg.sender, pool, amount);
    }
```

Status

The `updateSupportToken()` function has been removed by the development team in commit 8ef089b.

4.3.4 Illegal argument of `id` is not detected by the modifier function `poolExists()`.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

The `poolExists()` function determines if the pool is valid by comparing the value of the current argument `id` with the value of the current counter `_poolIds.current()`. The `id` of the first valid pool is one (instead of zero) because of code logic. A pool with an `id` of 0 should be treated as a non-existent pool, but it doesn't do so. Considering the contract upgrade, making the code as logical as possible is important to prevent security risks.

```
modifier poolExists(uint256 id) {
        require(id <= _poolIds.current(), "pool does not
exists");
        _;
   }
```

```
function createPool(
        uint256 periodStart,
        uint256 maxStake,
        uint256 rewardRate,
        uint256 exchangeRate,
        uint256 decimals,
        address poolMaster,
        address supportTokenAddress
    ) public onlyOwner returns (uint256 id) {
        require(supportTokenAddress != address(0), "only
support ERC20 token");

        _poolIds.increment();
        id = _poolIds.current();

        Pool storage p = pools[id];

        ......
    }
```

Suggestion

A simple modification option would be as follows.

```
function createPool(
        uint256 periodStart,
        uint256 maxStake,
        uint256 rewardRate,
        uint256 exchangeRate,
        uint256 decimals,
        address poolMaster,
        address supportTokenAddress
    ) public onlyOwner returns (uint256 id) {
        require(supportTokenAddress != address(0), "only
support ERC20 token");

        // @audit initial id starts at 0
```

```
        id = _poolIds.current();
        _poolIds.increment();

        Pool storage p = pools[id];

        ......
    }function burn_coll_want_only(uint256 n) public {
    require(rate == 1e18);
    recv_coll(n);
    send_want(n);
}
```

Status

The development team adopts our advice and updates the logic in commit b6d3cb9.

4.3.5 The intention of the `addCard()` function in the `DKPool` contract is unclear.

Risk Type	Risk Level	Impact	Status
Discussion	Info	Implementation logic	Discussed

Description

The `addCard()` function has two points that need to be clarified:

(1). the current code allows cards that are not created to be added;

(2). the arguments of `mintFee` and `feeToken` are not used;

```
function addCard(
        uint256 pool,
        uint256 id,
        uint256 DKP,
        uint256 mintFee,
        address feeToken,
```

```
        uint256 releaseTime,
        uint256 closeTime
    ) public poolExists(pool) {
        require(
            isPoolMaster(pool) || isOwner(),
            "Ownable: caller is not the owner or pool
poolMaster"
        );

        Card storage c = pools[pool].cards[id];
        c.DKP = DKP;
        c.releaseTime = releaseTime;
        c.closeTime = closeTime;

        ......
    }
```

Status

After discussion, the development team clarified the meaning of the function and updated the function name to `updateCard()`. The development introduces the modifier function `cardInPool()` to restrict the use of arbitrary cards. Meanwhile, the developer has removed the argument of `feeToken` in commit 2eb3a6d.

4.3.6 In `addCheckContract()` and `addRequireNFTs()` functions, the constraints on the input parameter card need to be added.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

The `addCheckContract()` function adds an additional check contract. It only determines whether the current card NFT token exists, but not whether it belongs to the current pool. It is recommended that the restriction be enhanced to only allow check contracts to be added for cards that currently exist in the pool.

Similarly, the problem of insufficient card constraints is also present in the `addRequireNFTs()` function.

```
function addCheckContract(
        uint256 pool,
        uint256 card,
        address checkContract
    ) public poolExists(pool) cardExists(card) {
        require(
            isPoolMaster(pool) || isOwner(),
            "Ownable: caller is not the owner or pool
poolMaster"
        );
        Card storage c = pools[pool].cards[card];
        c.checkContract = checkContract;

        emit CheckContractAdded(pool, checkContract);
    }

function addRequireNFTs(
        uint256 pool,
        uint256 card,
        address[] memory requireNFTContracts,
        uint256[] memory requireNFTIds,
        uint256[] memory requireNFTAmounts
    )
        public
        poolExists(pool)
        cardExists(card)
        nftsLengthEquals(requireNFTContracts, requireNFTIds,
requireNFTAmounts)
```

```
    {
        require(
            isPoolMaster(pool) || isOwner(),
            "Ownable: caller is not the owner or pool
poolMaster"
        );
        LockedNFT storage l = lockedNFTs[card];
        l.requireNFTContracts = requireNFTContracts;
        l.requireNFTIds = requireNFTIds;
        l.requireNFTAmounts = requireNFTAmounts;

        emit WinsRequireNFTAdded(pool, card);
    }
```

Status

The developer team has fixed this issue in commit a164fce.

4.3.7 In `redeem()` function, it may failed to transfer `feeToken` by the `transferFrom()` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Implementation logic	Fixed

Description

Due to the history of solidity upgrades, transferring token using the `IERC20(token).transferFrom()` method will fail for some specific ERC20 tokens.

The same problem exists with `donate()` , `withdrawDonate()` and `withdrawFee()` function in `DKPool` contract. Furthermore, in `PoolTokenWrapper()` contract, `stake()` and `withdraw()` function suffer from the same problem.

```
function redeem(uint256 pool, uint256 card)
```

```
        public
        payable
        nonReentrant
        poolExists(pool)
        cardExists(card)
        updateReward(msg.sender, pool)
        whenPoolRedeemNotPaused(pool)
    {
        ......

        if (c.mintFee > 0) {
            if (c.feeToken != address(0)) {
                IERC20 token = IERC20(c.feeToken);
                token.transferFrom(msg.sender, address(this),
c.mintFee);
            } else {
                require(msg.value == c.mintFee, "support us,
send fee");
            }

            p.feesCollected = p.feesCollected.add(c.mintFee);
        }

        ......
    }
```

Suggestion

The recommended modification is to use the `SafeREC20` library function instead of the `ERC20` library, see : SafeERC20.sol. The corresponding modifications are as follows.

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.so
l";

contract DKPool is
    Initializable,
    UUPSUpgradeable,
```

```
    PoolTokenWrapper,
    OwnableUpgradeable,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable
{
    using SafeERC20 for IERC20;
    function redeem(uint256 pool, uint256 card)
        public
        payable
        nonReentrant
        poolExists(pool)
        cardExists(card)
        updateReward(msg.sender, pool)
        whenPoolRedeemNotPaused(pool)
    {
        ......

        if (c.mintFee > 0) {
            if (c.feeToken != address(0)) {
                IERC20 token = IERC20(c.feeToken);
                token.safeTransferFrom(msg.sender,
address(this), c.mintFee);
            } else {
                require(msg.value == c.mintFee, "support us,
send fee");
            }

            p.feesCollected = p.feesCollected.add(c.mintFee);
        }

        ......
    }
}
```

Status

The development team adopts our advice in commit a164fce and 7081a3b.

4.3.8 Using external function instead of public function can save gas.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

The `public` function that is not called by the current contract can be declared as an `external` function, which effectively reduces the user's cost of gas.

```
function stake(uint256 pool, uint256 amount)
        public
        override
        poolExists(pool)
        updateReward(msg.sender, pool)
        whenNotPaused
        whenPoolNotPaused(pool)
    {
       ......
    }

function withdraw(uint256 pool, uint256 amount)
        public
        override
        poolExists(pool)
        updateReward(msg.sender, pool)
    {
       ......
    }

function transfer(
        uint256 fromPool,
        uint256 toPool,
```

```
        uint256 amount
    )
        public
        override
        poolExists(fromPool)
        poolExists(toPool)
        updateReward(msg.sender, fromPool)
        updateReward(msg.sender, toPool)
        whenNotPaused
        whenPoolNotPaused(fromPool)
        whenPoolNotPaused(toPool)
    {
       ......
    }
```

Status

The development team adopts our advice in commit a9c5770.

5. Conclusion

After auditing and analyzing the DAOSquare DKPool contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. The DAOSquare DKPool is a creative and permissionless community contribution protocol. It proposes a new pattern of DAO community governance that effectively filters out real community builders. SECBIT Labs holds the view that the DAOSquare DKPool contract has high code quality, concise implementation, and detailed documentation.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io