Security Audit Report for StakingKCS

Date: June 23, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.06.24 12:41:33 +08'00'

Contents

Report Manifest

Item	Description
Client	KCS Management Foundation
Target	StakingKCS

Version History

Version	Date	Description
1.0	June 23, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target is StakingKCS [1], a staking project which learns from Lido project and is developed based on KCCStaking. The KCCStaking project provides an interface for users to stake the KCS token (i.e., the native token of KuCoin Community Chain, aka KCC) to validators and gain profits. Generally, the original KCCStaking has the following limitations:

- A user must stake at least 1 KCS (= 1e18 minimum uints).
- The staking profits cannot achieve auto-compound.

To addresses them, the StakingKCS project provides solutions by aggregating users' stakings to multiple validators, and periodically withdrawing and re-staking the profits. Therefore, users can provide any amount of stakings, as the project will aggregate those less than 1 KCS and get their stakings auto-compounded.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the initial version (`Version 1`), as well as the new code (in the following versions) to fix issues in the audit report.

Project		Commit SHA
StakingKCS	`Version 1`	`81843bd8a14c821b071fb36094241dcb9a4975cf`
	`Version 2`	`35cbbe56c285d825f159f16a7fb8eb09f8a6a8c3`
	`Version 3`	`1bc28cde50ff66a8f5111d8354a88b24591fe499`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always

[1] https://github.com/stakingkcs/skcs

recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **five** potential issues. Besides, we also have **three** recommendations.

- High Risk: 2
- Medium Risk: 1
- Low Risk: 2
- Recommendations: 3

ID	Severity	Description	Category	Status
1	Medium	Permanently disabled validator	Software Security	Fixed
2	Low	Potential DoS due to the `revert` in a loop	DeFi Security	Fixed
3	High	Incorrect calculation of share amount	DeFi Security	Fixed
4	High	Unhandled staked amount	DeFi Security	Fixed
5	Low	Unchecked validator	DeFi Security	Fixed
6	-	Avoid accidentally locking KCS tokens	Recommendation	Fixed
7	-	Rewrite continuous division	Recommendation	Fixed
8	-	Remove redundant logic	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Permanently disabled validator

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `_tryRemoveDisabledValidator` of the `sKCS` contract, if the validator to be removed from `_disablingPool` has a zero `stakedKCS` field, the branch will not reset the validator info. In the meanwhile, the `addUnderlyingValidator` function requires that the validator is not added before by checking if the corresponding validator information entry has a zero address. In other words, once the validator is removed from the `_disablingPool` by invoking the `_tryRemoveDisabledValidator` function, there's a posibility that it can never be added again.

```
function _tryRemoveDisabledValidator() internal {

    for (uint8 i = 0; i < _disablingPool.length(); ) {
        address val = _disablingPool.at(i);
        if (VALIDATOR_CONTRACT.isWithdrawable(address(this), val)) {
            uint256 amount = _withdrawKCSFromKCCStaking(val);
            protocolParams.sumOfWeight -= _validators[val].weight;
            _validators[val] = ValidatorInfo(address(0), 0, 0, 0, 0, 0);
            kcsBalances.buffer += amount;

            _disablingPool.remove(val);

        } else if (_validators[val].stakedKCS == 0){
```

```

                protocolParams.sumOfWeight -= _validators[val].weight;
                _disablingPool.remove(val);
            }else{
                i++;
            }
        }
    }
```

Listing 2.1: sKCS.sol

```
    function addUnderlyingValidator(address _val, uint256 _weight) external onlyOwner override {

        require(_val != address(0), "invalid address");
        require(_weight > 0 && _weight <= 100, "invalid weight");
        require(activeValidators.length < MAX_NUM_VALIDATORS,"too many validators");

        if (_validators[_val].val == address(0)) {

            _validators[_val] = ValidatorInfo(_val, _weight, 0, 0, 0, 0, 0);

            _availablePool.add(_val);
            activeValidators.push(_val);

            protocolParams.sumOfWeight += _weight;

            emit AddValidator(msg.sender, _val, _weight);
        }
    }
```

Listing 2.2: sKCS.sol

Impact A disabled validator after removal may never be added again.

Suggestion Reset the validator information entry when removing a validator.

2.2 DeFi Security

2.2.1 Potential DoS due to the `revert` in a loop

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `sKCS` contract aggregates and stakes users' deposit to the validators. The staking rewards are stored in the `Validators` contract of KCC Staking project and can be claimed through a call to the `Validators` contract. In the `_compound` function (and the `disableUnderlyingValidator` function, see Listing 2.7) in the `sKCS` contract, there are operations that involve withdrawing staking rewards from the `Validators` contract.

Specifically, the `_claimAllPendingRewards` function will be invoked to withdraw the rewards. This function contains a loop, in which there exists an internal call to the `_claimPendingRewards` function (line 209). The `_claimPendingRewards` function will eventually invoke the `_claimReward` function (line 220).

```
205   function _claimAllPendingRewards() internal returns(uint256) {
206
207       uint256 amount;
208       for (uint8 i = 0; i < activeValidators.length; i++) {
209           amount += _claimPendingRewards(_validators[activeValidators[i]].val);
210       }
211
212       emit ClaimPendingRewards(msg.sender, block.number, amount);
213       return amount;
214   }
215
216   function _claimPendingRewards(address _val) internal returns (uint256) {
217       require(_val != address(0), "invalid address");
218       uint256 before = address(this).balance;
219
220       VALIDATOR_CONTRACT.claimReward(_val);
221
222       uint256 amount = address(this).balance - before;
223       accumulatedRewardKCSAmount += amount;
224
225       (uint256 fee, uint256 leftAmount) = _calculateProtocolFee(amount);
226       kcsBalances.fee += fee;
227       return leftAmount;
228   }
```

Listing 2.3: sKCS.sol

However, in the `_claimReward` function, the claiming procedure might be reverted if the pending reward amount is zero when claiming rewards (line 995).

```
991  function _claimReward(address _val) internal {
992       UserInfo storage user = userInfo[_val][msg.sender];
993
994       uint256 pending = _calculatePendingReward(_val, msg.sender);
995       require(pending > 0, "Validators: no pending reward to claim.");
996
997
998       user.rewardDebt = user
999           .amount
1000          .mul(poolInfos[_val].accRewardPerShare)
1001          .div(1e12);
1002      _safeTransfer(pending,msg.sender);
1003
1004      emit ClaimReward(msg.sender, _val, pending);
1005  }
```

Listing 2.4: Validators.sol

Therefore, to avoid potential DoS, the `sKCS` contract should check if the pending rewards are zero or not before performing claims.

Impact Potential DoS to claim all pending rewards because the claiming procedure might be reverted if one of the pending rewards is zero.

Suggestion Avoid using the `require` statement in a loop.

2.2.2 Incorrect calculation of share amount

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `sKCS` contract, users can deposit KCS to get sKCS tokens as “shares”, a proof for withdrawing the deposit funds and rewards. In functions `depositKCS`, `deposit`, `mint` of the `sKCS` contract, the returned `dem` variable of `exchangeRate()` is the total supply of sKCS token. However, the formula for calculating how many shares should be minted to the user is incorrectly calculated by dividing `msg.value` with `dem` as the divisor. While the correct divisor should be `num`, which means the share amount is based on the proportion of the user’s deposit in the total user-owned staked amount.

```
function depositKCS(address receiver)
external
payable
override
returns (uint256) {
    require(receiver != address(0), "invalid address");
    require(msg.value > 0, "invalid amount");

    (uint256 num, uint256 dem) = exchangeRate();
    uint256 shares = msg.value * num / dem;

    _depositKCS(receiver, msg.value, shares);

    return shares;
}
```

Listing 2.5: sKCS.sol

```
function exchangeRate() public view returns(uint256 num, uint256 dem) {

  if (totalSupply() == 0) {
      // initialize exchange rate
      return (1,1);
  }

  uint256 total;
  uint256 staked;
  uint256 pendingRewards;

  // all staked KCS and all yielded pending rewards
  (staked, pendingRewards) = _totalAmountOfValidators();
  total += staked;
  total += kcsBalances.buffer;
  // rewards with fee excluded.
```

```
    (,uint256 rewardsExcludingFee) = _calculateProtocolFee(pendingRewards);
    total += rewardsExcludingFee;

    uint256 boxRedeemingID = redemptionRequestBox.redeemingID;
    if (redemptionRequestBox.length > boxRedeemingID) {
        // the amount of KCS of all requested redemption
        uint256 totalRedeemingAmount = redemptionRequestBox.accAmountKCS
                    - redemptionRequestBox.requests[boxRedeemingID].accAmountKCSBefore
                    - redemptionRequestBox.requests[boxRedeemingID].partiallyRedeemedKCS;
        total -= totalRedeemingAmount;
    }

    return (total, totalSupply());
}
```

Listing 2.6: sKCS.sol

Impact Incorrect calculation may lead to losses.

Suggestion Fix the calculation logic.

2.2.3 Unhandled staked amount

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `sKCS` contract holds a list named `activeValidators` containing validators that the contract stakes to. All active validators will be either in `_availablePool` or `_redeemingPool` depending on the staking state. A validator could be disabled in the `disableUnderlyingValidator` function. However, when removing a validator from `_availablePool`, the function handles the pending rewards of the validator without considering the staked amount. As the calculation result of the `exchangeRate` function (see Listing 2.6) relies on the total staked amount of **every validator** in `activeValidators`, the unhandled amount would result in bad accounting which eventually leads to the losses of all liquidity providers.

```
function disableUnderlyingValidator(address _val) external onlyOwner override {
    // It can't to be removed if there ware only one validator
    require(activeValidators.length > 1, "not enough validator!");

    require(_val != address(0), "invalid address");

   if (_availablePool.contains(_val)) {
       _availablePool.remove(_val);
       kcsBalances.buffer += _claimPendingRewards(_val);
       if(_validators[_val].stakedKCS > 0 ){
           VALIDATOR_CONTRACT.revokeVote(_val, _validators[_val].stakedKCS);
       }
       _disablingPool.add(_val);

       removeUnderlyingValidator(_val);
       emit DisablingValidator(msg.sender, _val);
   }
```

```
}
```

Listing 2.7: sKCS.sol

Impact It may lead to the losses of liquidity providers.

Suggestion Handle the staked amounts when disabling validators.

2.2.4 Unchecked validator

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `addUnderlyingValidator` function (see Listing 2.2) of the `sKCS` contract, the validator address is checked to avoid a zero address. However, there does not exists any check to verify the validity of the validator. In fact, the `Validators` contract provides such an interface to perform the verification.

Impact A invalid validator might be added to the `sKCS` contract without verifying the validity.

Suggestion Check the validity of the validator when adding it.

2.3 Additional Recommendation

2.3.1 Avoid accidentally locking KCS tokens

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `receive` function in the `sKCS` contract does not contain any logic. As a result, if an EOA mistakenly sends KCS tokens into the `sKCS` contract, these tokens will be locked. This locking can be avoided by adding some extra logic, e.g., the `receive` function could invoke the `deposit` function for an EOA.

```
receive() external payable{
    // TODO: event
}
```

Listing 2.8: sKCS.sol

Impact KCS tokens that mistakenly sent to the contract can be locked.

Suggestion Implement some extra logic to prevent it.

2.3.2 Rewrite continuous division

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_calculateProtocolFee` function of the `sKCS` contract, the `feeAmount` is calculated with a continuous division, which may lead to a precision loss. It is suggested to rewrite the continuous division to avoid the potential precision loss problem.

```
function _calculateProtocolFee(uint256 totalAmount) internal view returns(uint256 feeAmount,
    uint256 leftAmount) {
     if(totalAmount == 0){
         return (0,0);
     }
     feeAmount = totalAmount * 1e12 * protocolParams.protocolFee / 10000 / 1e12;
     leftAmount = totalAmount - feeAmount;
 }
```

Listing 2.9: sKCS.sol

Impact Continuous division can lead to precision losses.

Suggestion Rewrite the continuous division.

2.3.3 Remove redundant logic

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The fixed logic of the `disableUnderlyingValidator` function ensures that the validators in `_disablingPool` must have a zero `userRedeeming` field and a zero `stakedKCS` field. Besides, the validator cannot be in `_disablingPool` and `activeValidators` simultaneously. As a result, there are two code segments that are redundant in `Version 2` (see Listing 2.11 and Listing 2.12). Specifically, the code at Line 453 in Listing 2.11 examines if an active validator is in the `_disablingPool`, which can not be. And the codes at Line 276 and 277 in Listing 2.12 try to add `stakedKCS` and `userRedeeming` fields, which are actually zero, to other variables.

```
function disableUnderlyingValidator(address _val) external onlyOwner override {
    // It can't to be removed if there ware only one validator
    require(activeValidators.length > 1, "not enough validator!");

    require(_val != address(0), "invalid address");

   if (_availablePool.contains(_val)) {
       _availablePool.remove(_val);
       kcsBalances.buffer += _claimPendingRewards(_val);
       if(_validators[_val].stakedKCS > 0 ){
            VALIDATOR_CONTRACT.revokeVote(_val, _validators[_val].stakedKCS);
            // @audit Fix Item 3: Unhandled staked amount
            _validators[_val].actualRedeeming = _validators[_val].stakedKCS;
            _validators[_val].userRedeeming = 0;
            _validators[_val].stakedKCS = 0;
       }
       _disablingPool.add(_val);

       _removeActiveValidator(_val);
       emit DisablingValidator(msg.sender, _val);
   }
}
```

Listing 2.10: sKCS.sol (Version 2)

```
function _getValidatorForStaking() internal returns (address) {

    (uint totalStaked, ,) = _totalAmountOfValidators();

    // If no KCS has been staked to any of the validators,
    // simply pick the first validator.
    if (totalStaked== 0) {
        return activeValidators.length == 0? address(0) : activeValidators[0];
    }

    int256 minWeight = type(int256).max;
    address available;
    for (uint8 i = 0; i < activeValidators.length; i++) {
        ValidatorInfo storage info = _validators[activeValidators[i]];
        if (_disablingPool.contains(activeValidators[i])) {
            continue;
        }
        int256 pri = int256((info.stakedKCS * 1e9 / totalStaked)) - int256(info.weight * 1e9 /
            protocolParams.sumOfWeight);
        if (pri <= minWeight) {
            minWeight = pri;
            available = activeValidators[i];
        }
    }
    return available;
}
```

Listing 2.11: sKCS.sol (Version 2)

```
function _totalAmountOfValidators() internal view returns (uint256 staked, uint256
    pendingRewards, uint256 residual) {

    for (uint8 i = 0; i < activeValidators.length; i++) {
        address val = activeValidators[i];
        // @audit Item 3: Unhandled staked amount
        staked += _validators[val].stakedKCS;
        residual += (_validators[val].actualRedeeming - _validators[val].userRedeeming);
        pendingRewards += VALIDATOR_CONTRACT.pendingReward(_validators[activeValidators[i]].val
            , address(this));
    }

    // @audit Item 3: Unhandled staked amount
    for (uint8 i = 0; i < _disablingPool.length(); i++) {
        address val = _disablingPool.at(i);
        staked += _validators[val].stakedKCS;
        residual += (_validators[val].actualRedeeming - _validators[val].userRedeeming);
    }
}
```

Listing 2.12: sKCSBase.sol (Version 2)

Impact Redundant logic may cause extra gas consumption.

Suggestion Remove the redundant logic.