# SMART CONTRACT AUDIT REPORT

for

# BSCStation Start Pools

Prepared By: Yiqun Chen

PeckShield

October 8, 2021

## Document Properties

| | |
|---|---|
| Client | BSCStation |
| Title | Smart Contract Audit Report |
| Target | BSCStation Start Pools |
| Version | 1.0 |
| Author | Shulin Bie |
| Auditors | Shulin Bie, Xuxian Jiang |
| Reviewed by | Yiqun Chen |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | October 8, 2021 | Shulin Bie | Final Release |
| 1.0-rc | September 30, 2021 | Shulin Bie | Release Candidate |

## 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** | **4** |
| 1.1 | About BSCStation Start Pools | 4 |
| 1.2 | About PeckShield | 5 |
| 1.3 | Methodology | 5 |
| 1.4 | Disclaimer | 7 |
| **2** | **Findings** | **9** |
| 2.1 | Summary | 9 |
| 2.2 | Key Findings | 10 |
| **3** | **Detailed Results** | **11** |
| 3.1 | Potential Failed withdraw() After Transferring LP Token | 11 |
| 3.2 | Incompatibility With Deflationary/Rebasing Tokens | 14 |
| 3.3 | Immutable States If Only Set at Constructor() | 16 |
| 3.4 | Improved Validation Of Function Arguments | 18 |
| 3.5 | Accommodation Of Non-ERC20-Compliant Tokens | 19 |
| 3.6 | Timely _updatePool() In Multiple Routines | 21 |
| 3.7 | Trust Issue Of Admin Keys | 22 |
| **4** | **Conclusion** | **24** |
| | **References** | **25** |

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the BSCStation Start Pools, 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 BSCStation Start Pools

BSCStation, built on the Binance Smart Chain (BSC), aims to build a full-stack DeFi with NFT auction and become the economic infrastructure for DeFi and NFT powered by BSC. The BSCStation Start Pools is an important feature of BSCStation, which allows users to earn rewards for staking the underlying token. It enriches the BSCStation ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of BSCStation Start Pools is as follows:

Table 1.1: Basic Information of BSCStation Start Pools

| Item | Description |
|---:|---|
| Target | BSCStation Start Pools |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | October 8, 2021 |

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

- https://github.com/BSCStationSwap/smartcontracts.git (56bc0d3)

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

- https://github.com/BSCStationSwap/smartcontracts.git (1dd2057)

## 1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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

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 |

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) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 1.4 Disclaimer

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

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logics** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 Findings

## 2.1 Summary

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

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 3 | ■■■ |
| Low | 2 | ■■ |
| Informational | 2 | ■■ |
| Undetermined | 0 | |
| 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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key BSCStation Start Pools Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Potential Failed withdraw() After Transferring LP Token | Business Logic | Fixed |
| PVE-002 | Low | Incompatibility With Deflationary/Rebasing Tokens | Business Logic | Confirmed |
| PVE-003 | Informational | Immutable States If Only Set at Constructor() | Coding Practices | Confirmed |
| PVE-004 | Informational | Improved Validation Of Function Arguments | Coding Practices | Confirmed |
| PVE-005 | Low | Accommodation Of Non-ERC20-Compliant Tokens | Coding Practices | Confirmed |
| PVE-006 | Medium | Timely _updatePool() In Multiple Routines | Business Logic | Mitigated |
| 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 Potential Failed withdraw() After Transferring LP Token

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: BSCSBaseStartPool
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

### Description

The BSCSBaseStartPool contract allows users to stake the underlying stakedToken token and get in return LP token (i.e., "BSCS BSCS Start Pool") to represent the pool shares. Meanwhile, the BSCSBaseStartPool contract supports all the standard ERC20 interfaces (including transfer()/transferFrom()) since it inherits from the standard ERC20 contract. In other words, the LP token can be transferred like the standard ERC20 token. While examining the logics of the deposit()/withdraw() routines, we notice there is a potential vulnerability that may result in the failure of the call to withdraw().

To elaborate, we show below the related code snippet of the BSCSBaseStartPool contract. In the deposit() function, the following statement is executed to record the user's deposit amount: user.amount = user.amount.add(_amount) (line 1134), and at the same time the same amount of LP token will be minted (line 1140). In the withdraw() function, the user.amount will be subtracted from the withdrawable amount of the underlying token (line 1190), and the same withdraw amount of LP token will be burned (line 1191). This is reasonable under the assumption that the vault's internal asset balances (i.e., user.amount) are always consistent with actual token balances maintained in individual ERC20 token contracts. However, we notice the transfer() interface of the BSCSBaseStartPool contract is inherited from the standard ERC20 contract, which only maintains the LP token balances. If we assume Alice transfers the LP token to Bob, both Alice and Bob cannot withdraw the deposit underlying token because of the inconsistency between the internal asset records (i.e., user.amount) and LP token balances maintained in ERC20 token contracts. We suggest to override the _transfer() interface to add the internal asset balances (i.e., user.amount) update.

```
    function deposit(uint256 _amount) external nonReentrant {
        UserInfo storage user = userInfo[msg.sender];

        require(stakingBlock <= block.number, "Staking has not started");
        require(stakingEndBlock >= block.number, "Staking has ended");

        if (hasPoolLimit) {
            uint256 stakedTokenSupply = stakedToken.balanceOf(address(this));
            require(
                _amount.add(stakedTokenSupply) <= poolCap,
                "Pool cap reached"
            );
        }

        if (hasUserLimit) {
            require(
                _amount.add(user.amount) <= poolLimitPerUser,
                "User amount above limit"
            );
        }

        _updatePool();

        if (user.amount > 0) {
            uint256 pending;
            for (uint256 i = 0; i < rewardTokens.length; i++) {
                pending = user
                .amount
                .mul(accTokenPerShare[rewardTokens[i]])
                .div(PRECISION_FACTOR[rewardTokens[i]])
                .sub(user.rewardDebt[rewardTokens[i]]);
                if (pending > 0) {
                    ERC20(rewardTokens[i]).transfer(
                        address(msg.sender),
                        pending
                    );
                }
            }
        }

        if (_amount > 0) {
            user.amount = user.amount.add(_amount);
            ERC20(stakedToken).transferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            _mint(address(msg.sender), _amount);
        }
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            user.rewardDebt[rewardTokens[i]] = user
            .amount
```

```
            .mul(accTokenPerShare[rewardTokens[i]])
            .div(PRECISION_FACTOR[rewardTokens[i]]);
        }

        user.lastStakingBlock = block.number;

        emit Deposit(msg.sender, _amount);
    }
```

Listing 3.1: `BSCSBaseStartPool::deposit()`

```
    function withdraw(uint256 _amount) external nonReentrant {
        UserInfo storage user = userInfo[msg.sender];
        require(unStakingBlock <= block.number, "Unstaking has not started");
        require(user.amount >= _amount, "Amount to withdraw too high");

        _updatePool();

        // uint256 pending = user.amount.mul(accTokenPerShare).div(PRECISION_FACTOR).sub
            (user.rewardDebt);
        uint256 pending;
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            pending = user
            .amount
            .mul(accTokenPerShare[rewardTokens[i]])
            .div(PRECISION_FACTOR[rewardTokens[i]])
            .sub(user.rewardDebt[rewardTokens[i]]);
            if (pending > 0) {
                // ERC20(rewardTokens[i]).transfer(address(msg.sender), pending);
                safeERC20Transfer(ERC20(rewardTokens[i]), address(msg.sender),pending);
            }
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            _burn(address(msg.sender), _amount);
            _amount = collectFee(_amount, user);
            ERC20(stakedToken).transfer(address(msg.sender), _amount);
            //_burn(address(msg.sender),_amount);
        }
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            user.rewardDebt[rewardTokens[i]] = user
            .amount
            .mul(accTokenPerShare[rewardTokens[i]])
            .div(PRECISION_FACTOR[rewardTokens[i]]);
        }

        emit Withdraw(msg.sender, _amount);
    }
```

Listing 3.2: `BSCSBaseStartPool::withdraw()`

Moreover, we notice there is a lack of the reward recalculation during transferring the LP token, which will introduce unexpected loss. Given this, we suggest to override the `_transfer()` interface in the `BSCSBaseStartPool` contract to add the reward recalculation mechanism.

**Recommendation** Suggest to override the `_transfer()` interface as above-mentioned.

**Status** The issue has been addressed by the following commit: `1dd2057`.

## 3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BSCSBaseStartPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

### Description

In the `BSCStation Start Pools` implementation, the `BSCSBaseStartPool` contract is designed to be the main entry for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of the `stakedToken` assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `BSCSBaseStartPool` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function deposit(uint256 _amount) external nonReentrant {
        UserInfo storage user = userInfo[msg.sender];

        require(stakingBlock <= block.number, "Staking has not started");
        require(stakingEndBlock >= block.number, "Staking has ended");

        if (hasPoolLimit) {
            uint256 stakedTokenSupply = stakedToken.balanceOf(address(this));
            require(
                _amount.add(stakedTokenSupply) <= poolCap,
                "Pool cap reached"
            );
        }

        if (hasUserLimit) {
            require(
                _amount.add(user.amount) <= poolLimitPerUser,
                "User amount above limit"
            );
        }

        _updatePool();
```

```
        if (user.amount > 0) {
            uint256 pending;
            for (uint256 i = 0; i < rewardTokens.length; i++) {
                pending = user
                .amount
                .mul(accTokenPerShare[rewardTokens[i]])
                .div(PRECISION_FACTOR[rewardTokens[i]])
                .sub(user.rewardDebt[rewardTokens[i]]);
                if (pending > 0) {
                    ERC20(rewardTokens[i]).transfer(
                        address(msg.sender),
                        pending
                    );
                }
            }
        }

        if (_amount > 0) {
            user.amount = user.amount.add(_amount);
            ERC20(stakedToken).transferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            _mint(address(msg.sender), _amount);
        }
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            user.rewardDebt[rewardTokens[i]] = user
            .amount
            .mul(accTokenPerShare[rewardTokens[i]])
            .div(PRECISION_FACTOR[rewardTokens[i]]);
        }

        user.lastStakingBlock = block.number;

        emit Deposit(msg.sender, _amount);
    }
```

Listing 3.3: `BSCSBaseStartPool::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of `BSCStation Start Pools` and affects protocol-wide operation and maintenance.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `BSCSBaseStartPool` before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `BSCStation Start Pools`. In `BSCStation Start Pools` protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

**Recommendation** If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

**Status** The issue has been confirmed by the team. There is no need to support deflationary/rebasing tokens.

## 3.3 Immutable States If Only Set at Constructor()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BSCSBaseStartPool`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

### Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `BSCSBaseStartPool` contract, we observe there is no need to dynamically update the `BSCStaion_CASTLE_FACTORY` variable. It can be declared as `immutable` for gas efficiency.

```
contract BSCSBaseStartPool is
  Ownable,
  ReentrancyGuard,
  ERC20("BSCS BSCS Start Pool", "BSCS-BSCS")
{
  using SafeMath for uint256;

  // The address of the smart chef factory
  address public BSCStaion_CASTLE_FACTORY;

  // Whether a limit is set for users
  bool public hasUserLimit;

  // Whether a limit is set for the pool
  bool public hasPoolLimit;

  // Whether it is initialized
  bool public isInitialized;

  ...
}
```

Listing 3.4: BSCSBaseStartPool

**Recommendation** Revisit the state variable definition and make good use of `immutable`/`constant` states.

**Status** The issue has been confirmed. The team decides to leave it as is since it has no impact on the service.

## 3.4 Improved Validation Of Function Arguments

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BSCSBaseStartPool
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [4]

### Description

According to the `BSCSBaseStartPool` contract design, we notice the `updateUnstakingFee()` routine is designed to update the transaction fee (i.e., `unStakingFee`) and the precision of the fee is `10000`. To elaborate, we show below the related code snippet of the `updateUnstakingFee()` routine.

In the `updateUnstakingFee()` function, we notice the input `_newFee` is directly stored into the `unStakingFee` storage variable (line 1308) without any validation. This is reasonable under the assumption that the input `_newFee` parameter is always correctly provided. However, in the unlikely situation, if the `_newFee` is improperly provided (e.g., larger than `10000`), the calling of the `withdraw()` function will be reverted.

```
1307    function updateUnstakingFee(uint256 _newFee) external onlyOwner {
1308        unStakingFee = _newFee;
1309    }
```

Listing 3.5: `BSCSBaseStartPool::updateUnstakingFee()`

Moreover, in the `updateFeeCollector()` function, we notice the input `_newCollector` is stored into the `feeCollector` storage variable (line 1313) as long as they are not equal (line 1312). However, in the unlikely situation, if `address(0)` is improperly provided, the current validation will not take effect (line 1312), which will result in the loss of the fee in the following transactions. We suggest to enhance the input `_newCollector` parameter validation.

```
1311    function updateFeeCollector(address _newCollector) external onlyOwner {
1312        require(_newCollector != feeCollector, "Already the fee collector");
1313        feeCollector = _newCollector;
1314    }
```

Listing 3.6: `BSCSBaseStartPool::updateFeeCollector()`

**Recommendation** Add necessary validation for above-mentioned routines.

**Status** The issue has been confirmed. The team decides to leave it as is.

## 3.5 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BSCSBaseStartPool`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.7: `ZRX.sol`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `safeERC20Transfer()` routine in the `BSCSBaseStartPool` contract. If the `USDT` token is supported as `erc20`, the unsafe version of `erc20.transfer(_to, balance)` (line 1159) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `erc20.transfer(_to, balance)` (line 1159) with `safeTransfer()`.

```
1154    function safeERC20Transfer(ERC20 erc20, address _to, uint256 _amount)
1155      private
1156    {
1157        uint256 balance = erc20.balanceOf(address(this));
1158        if (_amount > balance) {
1159            erc20.transfer(_to, balance);
1160        }
1161        else {
1162            erc20.transfer(_to, _amount); }
1163    }
```

Listing 3.8: `BSCSBaseStartPool::safeERC20Transfer()`

Note a number of routines can be similarly improved, including `deposit()`, `safeERC20Transfer()`, `withdraw()`, `collectFee()`, `emergencyRewardWithdraw()`, `recoverWrongTokens()`, `emergencyRemoval()` and `recoverWrongTokens()`.

**Recommendation** Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

**Status** This issue has been confirmed by the team. The team decides to leave it as non-compliant ERC20 tokens will not be used in the `BSCStation Start Pools` implementation.

## 3.6 Timely _updatePool() In Multiple Routines

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: BSCSBaseStartPool
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

### Description

The BSCStation Start Pools protocol provides an incentive mechanism that rewards the staking of the specified stakedToken assets with several kinds of ERC20 tokens specified by the rewardTokens array. The staking users are rewarded in proportional to their stakedToken assets in the pool.

The new reward token can be dynamically added via the addRewardToken() routine and the reward rate (per block) of each token in the rewardTokens array can be adjusted via the updateRewardPerBlock () routine. When analyzing these two routines, we notice the lack of timely invoking _updatePool() to update the accTokenPerShare and lastRewardBlock variables before the new reward-related configuration becomes effective.

If the call to _updatePool() is not immediately invoked before adding the new reward token or updating the reward rate, certain situations may be crafted to create an unfair reward distribution. With that, we suggest to timely invoke the _updatePool() at the beginning of these two routines.

```
    function addRewardToken(ERC20 _token, uint256 _rewardPerBlock)
        external
        onlyOwner
    {
        require(address(_token) != address(0), "Must be a real token");
        require(address(_token) != address(this), "Must be a real token");
        (bool foundToken, uint256 tokenIndex) = findElementPosition(
            _token,
            rewardTokens
        );
        require(!foundToken, "Token exists");
        rewardTokens.push(_token);

        uint256 decimalsRewardToken = uint256(_token.decimals());
        require(decimalsRewardToken < 30, "Must be inferior to 30");
        PRECISION_FACTOR[_token] = uint256(
            10**(uint256(30).sub(decimalsRewardToken))
        );
        rewardPerBlock[_token] = _rewardPerBlock;
        accTokenPerShare[_token] = 0;

        emit NewRewardToken(_token, _rewardPerBlock, PRECISION_FACTOR[_token]);
```

```
    }
```

Listing 3.9: BSCSBaseStartPool::addRewardToken()

```
    function updateRewardPerBlock(uint256 _rewardPerBlock, ERC20 _token)
        external
        onlyOwner
    {
        require(block.number < startBlock, "Pool has started");
        (bool foundToken, uint256 tokenIndex) = findElementPosition(
            _token,
            rewardTokens
        );
        require(foundToken, "Cannot find token");
        rewardPerBlock[_token] = _rewardPerBlock;
        emit NewRewardPerBlock(_rewardPerBlock, _token);
    }
```

Listing 3.10: BSCSBaseStartPool::updateRewardPerBlock()

Note the other routine, i.e., updateStartAndEndBlocks(), can also benefit from this improvement.

**Recommendation** Timely invoke _updatePool() when reward-related configuration has been updated in above-mentioned routines.

**Status** The issue has been confirmed by the team. The team decides to only add _updatePool() in the addRewardToken() function (commit hash:7d905b7).

## 3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BSCSBaseStartPool
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

### Description

In the BSCStation Start Pools implementation, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privileged account.

```
    function emergencyRewardWithdraw(uint256 _amount) external onlyOwner {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            ERC20(rewardTokens[i]).transfer(address(msg.sender), _amount);
        }
```

```
    }
```

Listing 3.11: `BSCSBaseStartPool::emergencyRewardWithdraw()`

```
    function emergencyRemoval(uint256 _amount) external onlyOwner {
        require(isRemovable, "The pool is not removable");
        require(
            stakedToken.balanceOf(address(this)) >= _amount,
            "Amount exceeds pool balance"
        );
        if (_amount > 0) {
            ERC20(stakedToken).transfer(address(msg.sender), _amount);
        }
    }
```

Listing 3.12: `BSCSBaseStartPool::emergencyRemoval()`

```
    function removeRewardToken(ERC20 _token) external onlyOwner {
        require(address(_token) != address(0), "Must be a real token");
        require(address(_token) != address(this), "Must be a real token");
        require(rewardTokens.length > 0, "List of token is empty");
        (bool foundToken, uint256 tokenIndex) = findElementPosition(
            _token,
            rewardTokens
        );
        require(foundToken, "Cannot find token");
        (bool success, ERC20[] memory newRewards) = removeElement(
            tokenIndex,
            rewardTokens
        );
        rewardTokens = newRewards;
        require(success, "Remove token unsuccessfully");
        PRECISION_FACTOR[_token] = 0;
        rewardPerBlock[_token] = 0;
        accTokenPerShare[_token] = 0;
        emit RemoveRewardToken(_token);
    }
```

Listing 3.13: `BSCSBaseStartPool::removeRewardToken()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged `owner` account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `BSCStaion` design.

**Recommendation** Promptly transfer the privileged `owner` account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** The issue has been confirmed by the team.

# 4 Conclusion

In this audit, we have analyzed the `BSCStation Start Pools` design and implementation. `BSCStation` aims to build a full-stack `DeFi` with `NFT` auction and become the economic infrastructure for `DeFi` and `NFT` powered by `BSC`. The `BSCStation Start Pools` is an important feature of `BSCStation`, which allows users to earn rewards for staking the underlying token. It enriches the `BSCStation` ecosystem and also presents a unique contribution to current DeFi ecosystem. 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-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

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

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

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

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