SMART CONTRACT AUDIT REPORT

for

THORSwap Staking

Prepared By: Yiqun Chen

PeckShield

October 15, 2021

Document Properties

Client	THORSwap
Title	Smart Contract Audit Report
Target	THORSwap Staking
Version	1.0
Author	Jing Wang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 15, 2021	Jing Wang	Final Release
1.0-rc	September 20, 2021	Jing Wang	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

Given the opportunity to review the design document and related smart contract source code of the THORSwap Staking 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 THORSwap Staking

The THORSwap Staking is a decentralized liquidity mining platform which provides an incentive mechanism to reward the staking of supported assets with certain reward tokens. Comparing to the traditional framing protocol, THORSwap Staking define a TokenRewarder to deliver additional rewards to the users and adjust the type of reward tokens dynamically.

The basic information of the THORSwap Staking protocol is as follows:

Table 1.1: Basic Information of The THORSwap Staking Protocol

Item	Description
Issuer	THORSwap
Website	https://thorswap.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 15, 2021

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

- https://github.com/thorswap/contracts.git (362485e)

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

- https://github.com/thorswap/contracts.git (TBD)

1.2 About PeckShield

PeckShield Inc. [7] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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) [5], 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 `THORSwap Staking` 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 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	3	■■■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key THORSwap Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-002	Medium	Improved Sanity Check of Claim()	Business Logic	Fixed
PVE-003	Low	Incompatibility with Deflationary Tokens	Business Logics	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Staking
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The THORSwap Staking protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

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

```
    function set(uint256 _pid, uint256 _allocPoint, IRewarder _rewarder, bool overwrite)
         public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) { rewarder[_pid] = _rewarder; }
        emit LogSetPool(_pid, _allocPoint, overwrite ? _rewarder : rewarder[_pid],
            overwrite);
    }
```

Listing 3.1: Staking::set()

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 owner (via the `onlyOwner` modifier), which greatly alleviates the concern. Note other routine `setBlockReward()` shares the same issue.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight is being updated.

```
    function set(uint256 _pid, uint256 _allocPoint, IRewarder _rewarder, bool overwrite)
         public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) { rewarder[_pid] = _rewarder; }
        emit LogSetPool(_pid, _allocPoint, overwrite ? _rewarder : rewarder[_pid],
            overwrite);
    }
```

Listing 3.2: Staking::set()

Status The issue has been fixed by this commit: `40a6607`.

3.2 Improved Sanity Check of Claim()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TokenVesting`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `THORSwap Staking` protocol, the `TokenVesting` contract is used to management the schedule of the user vesting. It allows the `owner` to add the vesting schedule for each payee and each payee could claim available vested funds based on the schedule. While reviewing the implementation of the `claim()` routine, we found the sanity check of this routine may cause the user unable to claim the funds. To elaborate, we show below the `claim()` function in the `TokenVesting` contract.

```
function claim(uint256 _amount) external {
    require(_claimAllowed == true, "TokenVesting: claim is disabled");
    require(_totalAlloc <= _token.balanceOf(address(this)), "TokenVesting: contract does
         not have enough funds");

    address payee = msg.sender;
    VestingSchedule storage v = _vestingSchedules[payee];

    require(v.amount > 0, "TokenVesting: not vested address");
```

```
    uint256 claimableTokens = claimableAmount(payee);

    require(claimableTokens > 0, "TokenVesting: no vested funds");

    require(_amount <= claimableTokens, "TokenVesting: cannot claim larger than total
        vested amount");

    v.claimed = v.claimed.add(_amount.to128());
    _totalClaimed = _totalClaimed.add(_amount);

    // transfer vested token to payee
    _token.safeTransfer(payee, _amount);

    emit TokensClaimed(payee, _amount);
  }
```

Listing 3.3: `TokenVesting::claim()`

The check of `require(_totalAlloc <= _token.balanceOf(address(this)))` may fail. This reason is that after each claim, the `_token.balanceOf(address(this))` is supposed to be gradually decreased, while the `_totalAlloc` would remain unchanged.

Recommendation Update `_totalAlloc` during each time of `claim()` or change the sanity check to `require(_amount <= _token.balanceOf(address(this)))`.

Status The issue has been fixed by this commit: `2cde60f`.

3.3 Incompatibility with Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

In the `THORSwap Staking` protocol, the `Staking` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract makes use of the `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /// @notice Deposit LP tokens to Staking contract for Reward token allocation.
    /// @param pid The index of the pool. See `poolInfo`.
    /// @param amount LP token amount to deposit.
    /// @param to The receiver of `amount` deposit benefit.
    function deposit(uint256 pid, uint256 amount, address to) public {
        PoolInfo memory pool = updatePool(pid);
        UserInfo storage user = userInfo[pid][to];

        // Effects
        user.amount = user.amount.add(amount);
        user.rewardDebt = user.rewardDebt.add(int256(amount.mul(pool.accRewardPerShare)
            / ACC_PRECISION));

        // Interactions
        IRewarder _rewarder = rewarder[pid];
        if (address(_rewarder) != address(0)) {
            _rewarder.onTokenReward(pid, to, to, 0, user.amount);
        }

        lpToken[pid].safeTransferFrom(msg.sender, address(this), amount);

        emit Deposit(msg.sender, pid, amount, to);
    }

    /// @notice Withdraw LP tokens from Staking contract.
    /// @param pid The index of the pool. See `poolInfo`.
    /// @param amount LP token amount to withdraw.
    /// @param to Receiver of the LP tokens.
    function withdraw(uint256 pid, uint256 amount, address to) public {
        PoolInfo memory pool = updatePool(pid);
        UserInfo storage user = userInfo[pid][msg.sender];

        // Effects
        user.rewardDebt = user.rewardDebt.sub(int256(amount.mul(pool.accRewardPerShare)
            / ACC_PRECISION));
        user.amount = user.amount.sub(amount);

        // Interactions
        IRewarder _rewarder = rewarder[pid];
        if (address(_rewarder) != address(0)) {
            _rewarder.onTokenReward(pid, msg.sender, to, 0, user.amount);
        }

        lpToken[pid].safeTransfer(to, amount);

        emit Withdraw(msg.sender, pid, amount, to);
    }
```

Listing 3.4: `Staking::deposit()`and `Staking::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, 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 the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accTokenPerShare` via dividing `rewards` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 157). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may result in a huge `pool.accTokenPerShare`, which dramatically inflates the pool's reward.

```
154     function updatePool(uint256 pid) public returns (PoolInfo memory pool) {
155         pool = poolInfo[pid];
156         if (block.number > pool.lastRewardBlock) {
157             uint256 lpSupply = lpToken[pid].balanceOf(address(this));
158             if (lpSupply > 0) {
159                 uint256 blocks = block.number.sub(pool.lastRewardBlock);
160                 uint256 rewards = blocks.mul(blockReward).mul(pool.allocPoint) /
                        totalAllocPoint;
161                 pool.accRewardPerShare = pool.accRewardPerShare.add((rewards.mul(
                        ACC_PRECISION) / lpSupply).to128());
162             }
163             pool.lastRewardBlock = block.number.to64();
164             poolInfo[pid] = pool;
165             emit LogUpdatePool(pid, pool.lastRewardBlock, lpSupply, pool.
                    accRewardPerShare);
166         }
167     }
```

Listing 3.5: `Staking::updatePool()`

One 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 `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` 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 `THORSwap Staking` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status The issue has been confirmed by the team. The team clarifies that they will regulate deflationary tokens before adding them to the farming pools.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Staking`, `TokenVesting`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `THORSwap` `Staking` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., funds withdraw and reward adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `set()` and `emergencyWithdraw()` functions in the `Staking` contract. This `set()` function allows the `owner` to change two key factor, `allocPoint` and `rewarder`. The `allocPoint` greatly affects on how many shares of the LP pools could receive and the `rewarder` plays a key role when user deposits/withdraws/harvests the funds. What's more, in the `withdrawAndHarvest()` function, a bad `rewarder` who will always revert the transaction could lock the user's funds.

```
    function set(uint256 _pid, uint256 _allocPoint, IRewarder _rewarder, bool overwrite)
         public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) { rewarder[_pid] = _rewarder; }
        emit LogSetPool(_pid, _allocPoint, overwrite ? _rewarder : rewarder[_pid],
            overwrite);
    }
```

Listing 3.6: `Staking::set()`

```
    function emergencyWithdraw(uint256 pid, address to) public {
        UserInfo storage user = userInfo[pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
```

```
        IRewarder _rewarder = rewarder[pid];
        if (address(_rewarder) != address(0)) {
            _rewarder.onTokenReward(pid, msg.sender, to, 0, 0);
        }

        // Note: transfer can fail or succeed if `amount` is zero.
        lpToken[pid].safeTransfer(to, amount);
        emit EmergencyWithdraw(msg.sender, pid, amount, to);
    }
```

Listing 3.7: `Staking::emergencyWithdraw()`

Also, if we examine the the `withdraw()` and `withdrawAll()` routines in the `TokenVesting` contract. These routines allow the caller to withdraw all funds from the contracts. Note that these privileged functions are guarded with `onlyOwner`.

```
    /**
    * @notice withdraw amount of token from vesting contract to owner
    * @param _amount token amount to withdraw from contract
    */
   function withdraw(uint256 _amount) external onlyOwner {
     require(_amount < _token.balanceOf(address(this)), "TokenVesting: withdraw amount
         larger than balance");

     _token.safeTransfer(owner(), _amount);
   }

   /**
    * @notice withdraw all token from vesting contract to owner
    */
   function withdrawAll() external onlyOwner {
     _token.safeTransfer(owner(), _token.balanceOf(address(this)));
   }
```

Listing 3.8: `TokenVesting::withdraw()and TokenVesting::withdrawAll()`

We emphasize that the privilege configuration adjustment is required for proper protocol operations. However, it is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged 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. The team clarifies that it is intentional to mitigate emergency cases and future migrations.

3.5 Duplicate Pool Detection and Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `THORSwap Staking` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

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

```
    function add(uint256 allocPoint, IERC20 _lpToken, IRewarder _rewarder) public
        onlyOwner {
        uint256 lastRewardBlock = block.number;
        totalAllocPoint = totalAllocPoint.add(allocPoint);
        lpToken.push(_lpToken);
        rewarder.push(_rewarder);

        poolInfo.push(PoolInfo({
            allocPoint: allocPoint.to64(),
            lastRewardBlock: lastRewardBlock.to64(),
            accRewardPerShare: 0
        }));
        emit LogPoolAddition(lpToken.length.sub(1), allocPoint, _lpToken, _rewarder);
    }
```

Listing 3.9: `Staking::add()`

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.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(uint256 allocPoint, IERC20 _lpToken, IRewarder _rewarder) public
        onlyOwner {
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number;
        totalAllocPoint = totalAllocPoint.add(allocPoint);
        lpToken.push(_lpToken);
        rewarder.push(_rewarder);

        poolInfo.push(PoolInfo({
            allocPoint: allocPoint.to64(),
            lastRewardBlock: lastRewardBlock.to64(),
            accRewardPerShare: 0
        }));
        emit LogPoolAddition(lpToken.length.sub(1), allocPoint, _lpToken, _rewarder);
    }
```

Listing 3.10: `Staking::add()`

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

Status The issue has been fixed by this commit: `0f99e69`.

4 Conclusion

In this audit, we have analyzed the `THORSwap Staking` protocol design and implementation. The `THORSwap Staking` protocol provides a decentralized liquidity platform for `LP` providers to deposit assets into the liquidity pool and earn rewards in return. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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