



# SMART CONTRACT AUDIT REPORT

for

## QuantoSwap



Prepared By: Xiaomi Huang

PeckShield  
May 7, 2024

## Document Properties

Client	QuantoSwap
Title	Smart Contract Audit Report
Target	QuantoSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	May 7, 2024	Xuxian Jiang	Final Release
1.0-rc	May 6, 2024	Xuxian Jiang	Release Candidate #1

## Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About QuantoSwap . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	9
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary . . . . .	10
2.2	Key Findings . . . . .	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Duplicate Pool Detection and Prevention in MasterChef . . . . .	12
3.2	Timely massUpdatePools During Pool Weight Changes . . . . .	14
3.3	Suggested Adherence Of Checks-Effects-Interactions Pattern . . . . .	16
3.4	Sybil Attacks on QuantoSwap Token Voting . . . . .	18
3.5	Implicit Assumption Enforcement In AddLiquidity() . . . . .	20
3.6	Inconsistent Fee Share Calculation in QuantoSwapV2LiquidityMathLibrary . . . . .	22
3.7	Staking Incompatibility With Deflationary Tokens in SmartChef . . . . .	23
3.8	Trust Issue of Admin Keys . . . . .	25
<b>4</b>	<b>Conclusion</b>	<b>27</b>
	<b>References</b>	<b>28</b>

# 1 | Introduction

Given the opportunity to review the design document and related source code of the QuantoSwap 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 QuantoSwap

QuantoSwap is a DEX that is forked from the popular UniswapV2 with the further integration with MasterChef to engage protocol users. Harnessing the power of multiple liquidity pools, QuantoSwap LPs are fungible and are composable for a wide variety of utility, including on-chain collateralization, farming, as well as other yield strategies. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of QuantoSwap

Item	Description
Name	QuantoSwap
Website	<a href="https://quantoswap.org/">https://quantoswap.org/</a>
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 7, 2024

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

- <https://github.com/QuantoSwap/farming.git> (db6d742)
- <https://github.com/QuantoSwap/time-lock.git> (9b1ff54)

- <https://github.com/QuantoSwap/v2-periphery.git> (6af45bb)
- <https://github.com/QuantoSwap/v2-core.git> (5db3c39)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- <https://github.com/QuantoSwap/farming.git> (7397641)
- <https://github.com/QuantoSwap/time-lock.git> (9b1ff54)
- <https://github.com/QuantoSwap/v2-periphery.git> (6ba929d)
- <https://github.com/QuantoSwap/v2-core.git> (5db3c39)

## 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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

Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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.







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



## 2 | Findings

### 2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	1	
Medium	4	
Low	2	
Undetermined	1	
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerabilities, 4 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Duplicate Pool Detection and Prevention in MasterChef	Business Logic	Resolved
PVE-002	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-003	Medium	Suggested Adherence of Checks-Effects-Interactions (SmartChef)	Business Logic	Resolved
PVE-004	High	Sybil Attacks on QuantoSwap Token Voting	Business Logic	Resolved
PVE-005	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Resolved
PVE-006	Medium	Inconsistent Fee Share Calculation in QuantoSwapV2LiquidityMathLibrary	Business Logic	Resolved
PVE-007	Undetermined	Staking Incompatibility With Deflationary Tokens in SmartChef	Business Logic	Resolved
PVE-008	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Duplicate Pool Detection and Prevention in MasterChef

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

The QuantoSwap 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 (via a proper governance procedure). 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.

```
103     function add( uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate ) public
        onlyOwner {
104         if (_withUpdate) {
105             massUpdatePools();
106         }
107         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
```

```

108     totalAllocPoint = totalAllocPoint.add(_allocPoint);
109     poolInfo.push(
110         PoolInfo({
111             lpToken: _lpToken,
112             allocPoint: _allocPoint,
113             lastRewardBlock: lastRewardBlock,
114             accQNSPerShare: 0
115         })
116     );
117 }

```

Listing 3.1: MasterChef::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.

```

103     function checkPoolDuplicate(IERC20 _lpToken) public {
104         uint256 length = poolInfo.length;
105         for (uint256 pid = 0; pid < length; ++pid) {
106             require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
107         }
108     }
109
110     function add( uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate ) public
111         onlyOwner {
112         if (_withUpdate) {
113             massUpdatePools();
114         }
115         checkPoolDuplicate(_lpToken);
116         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
117         totalAllocPoint = totalAllocPoint.add(_allocPoint);
118         poolInfo.push(
119             PoolInfo({
120                 lpToken: _lpToken,
121                 allocPoint: _allocPoint,
122                 lastRewardBlock: lastRewardBlock,
123                 accQNSPerShare: 0
124             })
125     );

```

Listing 3.2: Revised MasterChef::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** This issue has been fixed in the following commit: [7397641](#).

## 3.2 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

As mentioned earlier, the QuantoSwap 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.

```

119     function set( uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner
120     {
121         if (_withUpdate) {
122             massUpdatePools();
123         }
124         totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
125     );
126         poolInfo[_pid].allocPoint = _allocPoint;
127     }

```

Listing 3.3: MasterChef::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.

**Recommendation** Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed. Note the same issue is also applicable to other routines, including `updateMultiplier()` and `newQNSPerBlock()`.

```

119     function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
120         massUpdatePools();
121         totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
122     );

```

---

```
122     poolInfo[_pid].allocPoint = _allocPoint;
```

Listing 3.4: Revised `MasterChef::set()`

**Status** This issue has been fixed in the following commit: [7397641](#).



### 3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterChef
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

#### Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the MasterChef as an example, the emergencyWithdraw() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 261) starts before effecting the update on internal states (lines 266–267), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

258     function emergencyWithdraw(uint256 _pid) public {
259         PoolInfo storage pool = poolInfo[_pid];
260         UserInfo storage user = userInfo[_pid][msg.sender];
261         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
262         if (_pid == 0){
263             depositedQNS = depositedQNS.sub(user.amount);
264         }
265         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
266         user.amount = 0;
267         user.rewardDebt = 0;
268     }

```

Listing 3.5: MasterChef::emergencyWithdraw()

Note that other routines share the same issue, including deposit(), withdraw(), enterStaking(), and leaveStaking().



**Recommendation** Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible re-entrancy.

**Status** This issue has been fixed in the following commit: [7397641](#).



### 3.4 Sybil Attacks on QuantoSwap Token Voting

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: QuantoSwapToken
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

The `QuantoSwapToken` tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the `QuantoSwapToken` contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted via `getPriorVotes()`.

Our analysis shows that the current governance functionality is vulnerable to a new type of so-called sybil attacks. For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 QNS tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of QNS. This sybil attack can be launched as follows:

```

188     function _delegate(address delegator, address delegatee)
189     internal
190     {
191         address currentDelegate = _delegates[delegator];
192         uint256 delegatorBalance = balanceOf(delegator); // balance of underlying QNSs (
            not scaled);
193         _delegates[delegator] = delegatee;
194
195         emit DelegateChanged(delegator, currentDelegate, delegatee);
196
197         _moveDelegates(currentDelegate, delegatee, delegatorBalance);
198     }
199
200     function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
201         if (srcRep != dstRep && amount > 0) {
202             if (srcRep != address(0)) {
203                 // decrease old representative
204                 uint32 srcRepNum = numCheckpoints[srcRep];
205                 uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].
                    votes : 0;
206                 uint256 srcRepNew = srcRepOld.sub(amount);
207                 _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
208             }
209
210             if (dstRep != address(0)) {
211                 // increase new representative
212                 uint32 dstRepNum = numCheckpoints[dstRep];

```

```

213         uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].
           votes : 0;
214         uint256 dstRepNew = dstRepOld.add(amount);
215         _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
216     }
217 }
218 }

```

Listing 3.6: QuantoSwapToken::\_delegate()

1. Malice initially delegates the voting to Trudy. Right after the initial delegation, Trudy can have 100 votes if he chooses to cast the vote.
2. Malice transfers the full 100 balance to  $M_1$  who also delegates the voting to Trudy. Right after this delegation, Trudy can have 200 votes if he chooses to cast the vote. The reason is that the QuantoSwapToken contract's `transfer()` does NOT `_moveDelegates()` together. In other words, even now Malice has 0 balance, the initial delegation (of Malice) to Trudy will not be affected, therefore Trudy still retains the voting power of 100 QNS. When  $M_1$  delegates to Trudy, since  $M_1$  now has 100 QNS, Trudy will get additional 100 votes, totaling 200 votes.
3. We can repeat by transferring  $M_i$ 's 100 QNS balance to  $M_{i+1}$  who also delegates the votes to Trudy. Every iteration will essentially add 100 voting power to Trudy. In other words, we can effectively amplify the voting powers of Trudy arbitrarily with new accounts created and iterated!

**Recommendation** To mitigate, it is necessary to accompany every single `transfer()` and `transferFrom()` with the `_moveDelegates()` so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above Sybil attacks.

**Status** This issue has been fixed in the following commit: [7397641](#).

### 3.5 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: QuantoSwapV2Router02
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [2]

#### Description

In the QuantoSwapV2Router02 contract, the addLiquidity() routine (see the code snippet below) is provided to add amountADesired amount of tokenA and amountBDesired amount of tokenB into the pool as liquidity via the QuantoSwapV2Router02::addLiquidity() routine. To elaborate, we show below the related code snippet.

```

32     function _addLiquidity(
33         address tokenA,
34         address tokenB,
35         uint amountADesired,
36         uint amountBDesired,
37         uint amountAMin,
38         uint amountBMin
39     ) internal virtual returns (uint amountA, uint amountB) {
40         // create the pair if it doesn't exist yet
41         if (IQuantoSwapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
42             IQuantoSwapV2Factory(factory).createPair(tokenA, tokenB);
43         }
44         (uint reserveA, uint reserveB) = QuantoSwapV2Library.getReserves(factory, tokenA
45             , tokenB);
46         if (reserveA == 0 && reserveB == 0) {
47             (amountA, amountB) = (amountADesired, amountBDesired);
48         } else {
49             uint amountBOptimal = QuantoSwapV2Library.quote(amountADesired, reserveA,
50                 reserveB);
51             if (amountBOptimal <= amountBDesired) {
52                 require(amountBOptimal >= amountBMin, 'QuantoSwapV2Router:
53                     INSUFFICIENT_B_AMOUNT');
54                 (amountA, amountB) = (amountADesired, amountBOptimal);
55             } else {
56                 uint amountAOptimal = QuantoSwapV2Library.quote(amountBDesired, reserveB
57                     , reserveA);
58                 assert(amountAOptimal <= amountADesired);
59                 require(amountAOptimal >= amountAMin, 'QuantoSwapV2Router:
60                     INSUFFICIENT_A_AMOUNT');
61                 (amountA, amountB) = (amountAOptimal, amountBDesired);
62             }
63         }
64     }
65 }
66
67 function addLiquidity(

```

```

61     address tokenA,
62     address tokenB,
63     uint amountADesired,
64     uint amountBDesired,
65     uint amountAMin,
66     uint amountBMin,
67     address to,
68     uint deadline
69 ) external virtual override ensure(deadline) returns (uint amountA, uint amountB,
    uint liquidity) {
70     (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
        amountBDesired, amountAMin, amountBMin);
71     address pair = QuantoSwapV2Library.pairFor(factory, tokenA, tokenB);
72     TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
73     TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
74     liquidity = IQuantoSwapV2Pair(pair).mint(to);
75 }

```

Listing 3.7: QuantoSwapV2Router02::addLiquidity()

It comes to our attention that the QuantoSwapV2Router02 has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on SquadSwap V2 Router may not be checked and may not be taken into account at all in certain scenarios.

**Recommendation** Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

**Status** This issue has been fixed in the following commit: [6ba929d](#).

### 3.6 Inconsistent Fee Share Calculation in QuantoSwapV2LiquidityMathLibrary

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: QuantoSwapV2LiquidityMathLibrary
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

In this section, we examine a specific QuantoSwapV2LiquidityMathLibrary library that is designed to provide a number of convenience functions, e.g. computing their exact value in terms of the underlying tokens. Our analysis of this library exposes a specific function `computeLiquidityValue()` for improvement.

To elaborate, we show below this `computeLiquidityValue()` routine. This routine implements a rather straightforward logic in computing the liquidity value given all six parameters of the pair, i.e., `reservesA`, `reservesB`, `totalSupply`, `liquidityAmount`, `feeOn`, and `kLast`. Notice that this routine uses 1/3 of collected swap fee for protocol fee while default 1/2 of collected swap fee, if turned on, will be collected for protocol fee.

```

75     function computeLiquidityValue(
76         uint256 reservesA ,
77         uint256 reservesB ,
78         uint256 totalSupply ,
79         uint256 liquidityAmount ,
80         bool feeOn ,
81         uint kLast
82     ) internal pure returns (uint256 tokenAAmount, uint256 tokenBAmount) {
83         if (feeOn && kLast > 0) {
84             uint rootK = Babylonian.sqrt(reservesA.mul(reservesB));
85             uint rootKLast = Babylonian.sqrt(kLast);
86             if (rootK > rootKLast) {
87                 uint numerator1 = totalSupply;
88                 uint numerator2 = rootK.sub(rootKLast);
89                 uint denominator = rootK.mul(2).add(rootKLast);
90                 uint feeLiquidity = FullMath.mulDiv(numerator1, numerator2, denominator)
91                 ;
92                 totalSupply = totalSupply.add(feeLiquidity);
93             }
94         }
95         return (reservesA.mul(liquidityAmount) / totalSupply, reservesB.mul(
96             liquidityAmount) / totalSupply);
97     }

```

Listing 3.8: QuantoSwapV2LiquidityMathLibrary::computeLiquidityValue()

**Recommendation** Revise the above `computeLiquidityValue()` routine to be consistent in collecting the percentage of swap fee for protocol fee.

**Status** This issue has been fixed in the following commit: [6ba929d](#).

### 3.7 Staking Incompatibility With Deflationary Tokens in SmartChef

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: SmartChef, MasterChef
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

In the QuantoSwap protocol, the SmartChef 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 using the `safeTransfer()/safeTransferFrom()` routines 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.

```

710     function deposit(uint256 _amount) public {
711         PoolInfo storage pool = poolInfo[0];
712         UserInfo storage user = userInfo[msg.sender];

714         require(user.amount.add(_amount) <= limitAmount, 'Limit amount');

716         updatePool(0);
717         if (user.amount > 0) {
718             uint256 pending = user.amount.mul(pool.accQNSPerShare).div(PRECISION_FACTOR)
719                             .sub(user.rewardDebt);
720             if(pending > 0) {
721                 rewardToken.safeTransfer(address(msg.sender), pending);
722             }
723         }
724         if(_amount > 0) {
725             pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
726             user.amount = user.amount.add(_amount);
727         }
728         user.rewardDebt = user.amount.mul(pool.accQNSPerShare).div(PRECISION_FACTOR);

```

```

730     emit Deposit(msg.sender, _amount);
731 }

```

Listing 3.9: SmartChef::deposit()

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. 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.accQNSPerShare` via dividing the reward by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 692). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 Wei and thus may yield a huge `pool.accQNSPerShare` as the final result, which dramatically inflates the pool's reward.

```

687     function updatePool(uint256 _pid) public {
688         PoolInfo storage pool = poolInfo[_pid];
689         if (block.number <= pool.lastRewardBlock) {
690             return;
691         }
692         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
693         if (lpSupply == 0) {
694             pool.lastRewardBlock = block.number;
695             return;
696         }
697         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
698         uint256 QNSReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint);
699         pool.accQNSPerShare = pool.accQNSPerShare.add(QNSReward.mul(PRECISION_FACTOR).
            div(lpSupply));
700         pool.lastRewardBlock = block.number;
701     }

```

Listing 3.10: SmartChef::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 `QuantoSwap` 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** This issue has been resolved as the team confirms no plan to support deflationary tokens.

### 3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

#### Description

In the `QuantoSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure various parameters, set the migrator, manage reward pools, and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```

139     function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
140         BONUS_MULTIPLIER = multiplierNumber;
141     }
142     ...
143     function newDevAddress(address _devaddr) public onlyOwner {
144         devaddr = _devaddr;
145     }
146     ...
147     function newQNSPerBlock(uint256 newAmount) public onlyOwner {
148         require(newAmount > 1, 'Bad per block');
149         QNSPerBlock = newAmount;
150     }
151     ...

```

```
152     function set( uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner
153     {
154         if (_withUpdate) {
155             massUpdatePools();
156         }
157         totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
158             );
159         poolInfo[_pid].allocPoint = _allocPoint;
160     }
161     ...
162     // Set the migrator contract. Can only be called by the owner.
163     function setMigrator(IMigratorChef _migrator) public onlyOwner {
164         migrator = _migrator;
165     }
```

Listing 3.11: Example Privileged Functions in `MasterChef`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

**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** This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `QuantoSwap` protocol, which is a DEX that is forked from the popular `UniswapV2` with the further integration with `MasterChef` to engage protocol users. Harnessing the power of multiple liquidity pools, `QuantoSwap` LPs are fungible and are composable for a wide variety of utility, including on-chain collateralization, farming, as well as other yield strategies. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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



## References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. <https://cwe.mitre.org/data/definitions/628.html>.
- [3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [7] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [8] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.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](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

