SMART CONTRACT AUDIT REPORT

for

YuzuSwap

Prepared By: Yiqun Chen

PeckShield

December 6, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	December 6, 2021	Jing Wang	Final Release
1.0-rc	November 30, 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 YuzuSwap 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 YuzuSwap

YuzuSwap is a decentralized exchange on the Oasis Emerald paratime that includes incentives like liquidity and trade mining. It provides the features of a DEX that the DeFi community expects to have, e.g., swaps between ETH-based and Oasis-based tokens from liquidity pools/pairs, rewards of YUZU tokens for providing liquidity to pools, and a governance system with the collected funds from 20% of the swap fees.

The basic information of the YuzuSwap protocol is as follows:

Table 1.1: Basic Information of The YuzuSwap Protocol

Item	Description
Name	YuzuSwap
Website	https://yuzu-swap.com/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 6, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. This audit covers the smart contracts under the specific contracts_for_review directory.

- https://github.com/yuzuswap-oasis/yuzuswap-contract.git (fc7e84c)

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

- https://github.com/yuzuswap-oasis/yuzuswap-contract.git (0e54952)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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.

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

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) [8], 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 `YuzuSwap` 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	1	
Low	5	
Informational	0	
Total	6	

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 1 medium-severity vulnerability and 5 low-severity vulnerabilities.

Table 2.1: Key YuzuSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-002	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-003	Low	Duplicate Pool Detection and Prevention	Business Logics	Fixed
PVE-004	Low	Safe-Version Replacement With safeTransfer()	Coding Practices	Fixed
PVE-005	Low	Incompatibility with Deflationary Tokens	Business Logics	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Improved Validation Of Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UniswapV2Router02
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

Description

In the UniswapV2Router02 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 uniswapRouterV2::_addLiquidity() routine. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB,
        uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Listing 3.1: UniswapV2Router02::addLiquidity()

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.2: `UniswapV2Router02::_addLiquidity()`

It comes to our attention that the `Uniswap V2 Router` 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 `Uniswap 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 The issue has been fixed by this commit: `0e54952`.

3.2 Timely massUpdatePools During Pool Weight Changes

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

Description

The `YuzuSwap` protocol has a `YuzuPark` contract that 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,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.3: `YuzuPark::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 `YuzuSwapMining::set()` shares the same issue.

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.

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

Listing 3.4: YuzuPark::set()

Status The issue has been fixed by this commit: 0e54952.

3.3 Duplicate Pool Detection and Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.2, the YuzuSwap 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.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
      do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accYuzuPerShare: 0
            })
        );
    }
```

Listing 3.5: `YuzuPark::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?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
      do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public
        onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accALDPerShare: 0
```

```
        }));
    }
```

Listing 3.6: Revised `YuzuPark::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. Note other routine `YuzuSwapMining::add()` shares the same issue.

Status The issue has been fixed by this commit: `0e54952`.

3.4 Safe-Version Replacement With safeTransfer()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [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 stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
```

```
    }
```

Listing 3.7: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer (address to, uint256 value)external returns (bool)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

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.

In the following, we show the `safeYuzuTransfer()` routine in the `YuzuPark` contract. If `USDT` is given as `token`, the unsafe version of `yuzu.transfer(_to, yuzuBal)` (line 239) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
236     function safeYuzuTransfer(address _to, uint256 _amount) internal {
237         uint256 yuzuBal = yuzu.balanceOf(address(this));
238         if (_amount > yuzuBal) {
239             yuzu.transfer(_to, yuzuBal);
240         } else {
241             yuzu.transfer(_to, _amount);
242         }
243     }
```

Listing 3.8: `YuzuPark::safeYuzuTransfer()`

Note that another routine `YuzuSwapMining::safeYuzuTransfer()` shares the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()`.

Status The issue has been fixed by this commit: `0e54952`.

3.5 Incompatibility with Deflationary Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: YuzuPark
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

In the YuzuSwap protocol, the YuzuPark 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() or safeTransfer() 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.

```
    // Deposit LP tokens to YuzuPark for Yuzu allocation.
    function deposit(uint256 _pid, uint256 _amount) public nonReentrant{
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accYuzuPerShare).div(1e12).sub(
                    user.rewardDebt
                );
            safeYuzuTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accYuzuPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public nonReentrant{
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
```

```
        uint256 pending =
            user.amount.mul(pool.accYuzuPerShare).div(1e12).sub(
                user.rewardDebt
            );
        safeYuzuTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accYuzuPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.9: `YuzuPark::deposit()and YuzuPark::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary ones 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.accYuzuPerShare` via dividing `yuzuReward` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 169). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accYuzuPerShare` as the final result, which dramatically inflates the pool's reward.

```
164     function updatePool(uint256 _pid) public {
165         PoolInfo storage pool = poolInfo[_pid];
166         if (block.number <= pool.lastRewardBlock) {
167             return;
168         }
169         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
170         if (lpSupply == 0) {
171             pool.lastRewardBlock = block.number;
172             return;
173         }
174         uint256 yuzuReward = getYuzuBetweenBlocks(pool.lastRewardBlock, block.number).
                mul(pool.allocPoint).div(
175                 totalAllocPoint
176             );
177
178         yuzuReward = yuzukeeper.requestForYUZU(yuzuReward);
179
180         pool.accYuzuPerShare = pool.accYuzuPerShare.add(
181             yuzuReward.mul(1e12).div(lpSupply)
182         );
183         pool.lastRewardBlock = block.number;
184     }
```

Listing 3.10: `DepositPool::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 `YuzuPark` 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.

Status The issue has been fixed by this commit: `0e54952`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `YuzuSwap` 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., minting tokens, setting protocol-wide risk parameters, etc.). 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 their related privileged accesses in current contracts.

To elaborate, we show below the `mint()` functions in the `YuzuToken` contract, which allows the `owner` to add tokens into circulation and the recipient can be directly provided when the mint operation takes place.

```
    function mint(address _to, uint256 _amount) public onlyOwner returns (bool) {
        if (_amount.add(totalSupply()) > TOTAL_SUPPLAY) {
            return false;
        }
        _mint(_to, _amount);
```

```
        return true;
    }
```

Listing 3.11: `YuzuToken::mint()`

In addition, the `YuzuSwap` protocol uses the following mechanism to reward stakers with `YUZU` tokens:

- The `YuzuToken` contract has a `mint()` function that allows its owner to mint new tokens.
- The `YuzuKeeper` contract acts as the `YuzuToken` contract's owner and it mints `YUZU` per request from the applications that reward stakers with `YUZU`.
- The owner of `YuzuKeeper` could determine how many `YUZU` tokens could be delivered per application.

Our on-chain analysis shows that the owner of the `YuzuKeeper` contract is an `EOA` account, `0x8aC3195AEca398AaC7882520dd19d3C7c5e69E46`. And the owners of other contracts in the `YuzuSwap` protocol are also plain `EOA` accounts. And the owners take the important responsibility to configure `allocPoint` for a liquidity pool.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.12: `YuzuPark::set()`

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.

4 Conclusion

In this audit, we have analyzed the `YuzuSwap` protocol design and implementation. `YuzuSwap` is a decentralized exchange on the `Oasis Emerald paratime` that includes incentives like liquidity and trade mining. 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-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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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