SMART CONTRACT AUDIT REPORT

for

Paraluni

Prepared By: Xiaomi Huang

Hangzhou, China

April 27, 2022

Document Properties

Client	Paraluni
Title	Smart Contract Audit Report
Target	Paraluni
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 27, 2022	Jing Wang	Final Release
1.0-rc	April 10, 2022	Jing Wang	Release Candidate

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

PeckShield

1 Introduction

Given the opportunity to review the `Paraluni` protocol design document and related smart contract source code, 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 branch of `Paraluni` 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 Paraluni

`Paraluni` is a decentralized automated market making (`AMM`) protocol on `Binance Smart Chain (BSC)`. The `DEX` forks from the `UniswapV2`'s core design, but extends with features such as liquidity provider incentives and community-based governance. Also, the user could buy `NFT` ticket to gain the access for the `VIP` farming pool. The basic information of `Paraluni` is as follows:

Table 1.1: Basic Information of Paraluni

Item	Description
Name	Paraluni
Website	https://paraluni.org/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 27, 2022

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

- https://gitlab.com/maven42/paraluni_contracts.git (0a7146d)

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

- https://gitlab.com/maven42/paraluni_contracts.git (8e86155)

1.2 About PeckShield

PeckShield Inc. [15] 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

PeckShield

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- 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 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

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

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) [13], 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

PeckShield

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	6	■ ■ ■ ■ ■ ■
Informational	2	■ ■
Total	11	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, and 6 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Paraluni Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Reentrancy Risk in MasterChef	Time and State	Fixed
PVE-002	High	Voting Amplification With Sybil Attacks	Business Logic	Fixed
PVE-003	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-004	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Confirmed
PVE-005	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Fixed
PVE-006	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Informational	Redundant State/Code Removal	Coding Practices	Confirmed
PVE-009	Informational	Possible Overflow Prevention With SafeMath	Coding Practices	Fixed
PVE-010	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-011	Low	Incompatibility with Deflationary Tokens	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Reentrancy Risk in MasterChef

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MasterChef
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

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 [17] exploit, and the recent Uniswap/Lendf.Me hack [16].

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

Apparently, the interaction with the external contract (line 463) starts before effecting the update on the internal state (lines 635-636), 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 deposit() function.

```
444     function depositByAddLiquidityInternal(address _user, uint256 _pid, uint256 amount0,
            uint256 amount1) internal {
445         //Make sure the currency has been transferred in before that
446         PoolInfo memory pool = poolInfo[_pid];
447         //Non-VIP pool
```

```
        require(address(pool.ticket) == address(0), "T:E");
        uint liquidity = addLiquidityInternal(address(pool.lpToken), _user, amount0,
            amount1);
        _deposit(_pid, liquidity, _user);
    }

    function addLiquidityInternal(address lpToken, address _user, uint256 amount0,
        uint256 amount1) internal returns (uint){
        //Stack too deep, try removing local variables
        DepositVars memory vars;
        address token0 = IParaPair(lpToken).token0();
        address token1 = IParaPair(lpToken).token1();
         //Go approve
        approveIfNeeded(token0, address(paraRouter), amount0);
        approveIfNeeded(token1, address(paraRouter), amount1);
        //lp balance checkl
        vars.oldBalance = IParaPair(lpToken).balanceOf(address(this));
        (vars.amountA, vars.amountB, vars.liquidity) = paraRouter.addLiquidity(token0,
            token1, amount0, amount1, 1, 1, address(this), block.timestamp + 600);
        vars.newBalance = IParaPair(lpToken).balanceOf(address(this));
        require(vars.newBalance > vars.oldBalance, "B:E");
        vars.liquidity = vars.newBalance.sub(vars.oldBalance);
        addChange(_user, token0, amount0.sub(vars.amountA));
        addChange(_user, token1, amount1.sub(vars.amountB));
        return vars.liquidity;
    }
```

Listing 3.1: `MasterChef::depositByAddLiquidityInternal()`

```
    function _deposit(uint256 _pid, uint256 _amount, address _user) internal {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        //add total of pool before updatePool
        poolsTotalDeposit[_pid] = poolsTotalDeposit[_pid].add(_amount);
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accT42PerShare).div(1e12).sub(
                    user.rewardDebt
                );
            //TODO
            _claim(pool.pooltype, pending);
        }
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accT42PerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }
```

Listing 3.2: `MasterChef::_deposit()`

Note that other routines share the same issue, including `depositTo()`, `deposit_all_tickets()`, `withdraw_tickets()`, `withdraw()`, `withdraw_tickets()`, and `depositByAddLiquidityETH()` from the same contract as well as `supplyFlex()`, `supplyRegular()`, `withdrawFlex()`, `withdrawRegular()` from the `ParaSupply` contract.

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: 533802c.

3.2 Voting Amplification With Sybil Attacks

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ParaTokenNew`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `V42` tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the `ParaTokenNew` 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 `V42` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `V42s`. This `Sybil` attack can be launched as follows:

```
    function _delegate(address delegator, address delegatee)
        internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying T42s (
            not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
```

PeckShield

```
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].
                    votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].
                    votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Listing 3.3: `ParaTokenNew.sol`

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 `ParaTokenNew` 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 `V42s`. When M_1 delegates to `Trudy`, since M_1 now has 100 `V42s`, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `V42` 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. Since the contract is already deployed, it is safe and acceptable to deploy another contract for governance, and use the current one for other ERC-20 functions only. A cleaner solution would be to migrate the current contract to a new one with the suggested fix, but the migration effort may be costly.

Status This issue has been fixed in the following commit: 8e86155.

3.3 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The Paraluni protocol has a MasterChef 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.

```
    // Update the given pool's T42 allocation point. Can only be called by the owner.
    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 3.4: 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.

```
    // Update the given pool's T42 allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.5: MasterChef::set()

Status This issue has been fixed in the following commit: 348003c.

3.4 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ParaRouter`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [4]

Description

PeckShield

In the `ParaRouter` 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 `ParaRouter::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) {
        noFees(tokenA, tokenB);
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = ParaLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IParaPair(pair).mint(to);
        FeesOn(tokenA, tokenB);
    }
```

Listing 3.6: `ParaRouter::addLiquidity()`

```
    // **** ADD LIQUIDITY ****
    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 (IParaFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IParaFactory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = ParaLibrary.getReserves(factory, tokenA, tokenB
            );
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = ParaLibrary.quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'ParaRouter: INSUFFICIENT_B_AMOUNT
                    ');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = ParaLibrary.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'ParaRouter: INSUFFICIENT_A_AMOUNT
                    ');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.7: `ParaRouter::_addLiquidity()`

It comes to our attention that the `ParaRouter` 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 `ParaRouter` 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 confirmed.

3.5 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

As mentioned in Section 3.3, the `MasterChef` contract is designed to provide incentive mechanisms that reward the staking of supported assets. The `MasterChef` contract has a helper routine, i.e., `swapTokensIn()`, that is designed to swap user input token into desired tokens to add liquidity. It has a rather straightforward logic in calling `swapExactTokensForTokens()` to actually perform the intended token swap.

```
    function swapTokensIn(uint amountIn, address[] memory path) internal returns(address
         tokenOut, uint amountOut){
        uint[] memory amounts = paraRouter.swapExactTokensForTokens(amountIn, 0, path,
            address(this), block.timestamp + 600);
        tokenOut = path[path.length - 1];
        amountOut = amounts[amounts.length - 1];
    }
```

Listing 3.8: `MasterChef::swapTokensIn()`

To elaborate, we show above related routines. We notice the token swap is routed to `paraRouter` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Note that other routines share the same issue, including `swapTokensOut()`, `addLiquidityInternal()`, `withdrawSingle()`, `withdrawAndRemoveLiquidity()` from the same contract as well as `addLiquidityInternal()`, `swapTokensIn()` from the `MasterChefPeriphery` contract.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commit: 7daabbf.

3.6 Accommodation of approve() Idiosyncrasies

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

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 `approve()` routine and analyze 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
```

```
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.9: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `MasterChef::approveIfNeeded()` routine as an example. This routine is designed to approve the `paraRouter` contract to swap tokens. To accommodate the specific idiosyncrasy, for each `approve()` (line 744), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Also, the `IERC20` interface has defined the `approve()` interface with a `bool` return value, but the above implementation does not have the return value. As a result, a normal `IERC20`-based `approve()` with a non-compliant token may unfortunately revert the transaction. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of current `approve()` in the `MasterChef::approveIfNeeded()`.

```
742     function approveIfNeeded(address _token, address spender, uint _amount) private{
743         if (IERC20(_token).allowance(address(this), spender) < _amount) {
744              IERC20(_token).approve(spender, _amount);
745         }
746     }
```

Listing 3.10: `MasterChef::approveIfNeeded()`

Note another routine `MasterChefPeriphery::approveIfNeeded()` shares the same issue.

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

PeckShield

Status This issue has been fixed in the following commit: e273a561.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the `Paraluni` protocol, there is a special administrative account, i.e., `owner/admin`. This `owner/admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., setting various parameters, moving funds during migration). 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 `owner/admin` account and its related privileged accesses in current contract.

To elaborate, we show the `_setMinerAddress()` routine. This function allow the `owner` account to change the value of `minersAddress[]`, which play an important role in minting tokens.

```
    function _setMinerAddress(address _minerAddress, bool flag) external onlyOwner{
        minersAddress[_minerAddress] = flag;
    }
```

Listing 3.11: `ParaTokenNew::_setMinerAddress()`

Also, the `MasterChef` contract supports a migration feature that can migrate current pool liquidity to another contract.

```
    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that
        migrator contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "M:E");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        //TODO use poolsTotalDeposit insteadOf balanceOf(address(this)); ??
        uint256 bal = poolsTotalDeposit[_pid];
        lpToken.safeApprove(address(migrator), bal);
        //uint newLpAmountOld = newLpToken.balanceOf(address(this));
        IERC20 newLpToken = migrator.migrate(lpToken);
        uint newLpAmountNew = newLpToken.balanceOf(address(this));
        require(bal <= newLpAmountNew, "M:B");
        pool.lpToken = newLpToken;
    }
```

Listing 3.12: `ParaTokenNew::_setMinerAddress()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner/admin` 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

3.8 Redundant State/Code Removal

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ParaRouter
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

In the ParaRouter contract, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. For example, the member isAmountIn is defined but not used throughout the entire function implementation. What's more, the parameter is passed in each calling of the _swap() routine. If there is no actual uses of this variable, we suggest to simplify the contract by removing it.

```
    function _swap(uint[] memory amounts, address[] memory path, address _to, bool
        isAmountIn) internal virtual {
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = ParaLibrary.sortTokens(input, output);
            uint amountOut = amounts[i + 1];
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut)
                : (amountOut, uint(0));
            address to = i < path.length - 2 ? ParaLibrary.pairFor(factory, output, path
                [i + 2]) : _to;
            IParaPair(ParaLibrary.pairFor(factory, input, output)).swap(
                amount0Out, amount1Out, to, new bytes(0)
            );
        }
    }
```

Listing 3.13: The ParaRouter::_swap()

Recommendation Consider the removal of the redundant code with a simplified implementation.

Status The issue has been confirmed.

3.9 Possible Overflow Prevention With SafeMath

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: MasterChef
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While analyzing the MasterChef implementation, we observe it can be improved by taking advantage of the improved security from SafeMath.

In the computation of `_totalClaimed[msg.sender][pooltype] += pending.sub(fee)` (line 534) from the `MasterChef::_claim()` routine, the addition of `_totalClaimed[msg.sender][pooltype]` to `pending.sub(fee)` is not guarded against possible overflow. We should point out that this addition will not always overflow in this particular usage scenario. However, it is preferable to guarantee the overflow will always be detected and blocked.

```
531     function _claim(uint256 pooltype, uint pending) internal {
532         uint256 fee = pending.mul(claimFeeRate).div(10000);
533         safeT42Transfer(msg.sender, pending.sub(fee));
534         _totalClaimed[msg.sender][pooltype] += pending.sub(fee);
535         t42.approve(feeDistributor, fee);
536         IFeeDistributor(feeDistributor).incomeClaimFee(msg.sender, address(t42), fee);
537     }
```

Listing 3.14: The `ParaRouter::_claim()`

Recommendation Make use of SafeMath in the above calculations to better mitigate possible overflows.

Status The issue has been fixed by this commit: 0b07505.

3.10 Duplicate Pool Detection and Prevention

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `MasterChef` 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 privileged function). 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,
        uint256 _pooltype,
        IParaTicket _ticket,
        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,
                accT42PerShare: 0,
                pooltype: _pooltype,
                ticket: _ticket
            })
        );
    }
```

Listing 3.15: `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.

```
    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,
        uint256 _pooltype,
        IParaTicket _ticket,
        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,
                accT42PerShare: 0,
                pooltype: _pooltype,
                ticket: _ticket
            })
        );
    }
```

Listing 3.16: 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 The issue has been fixed by this commit: `3fb35ee`.

3.11 Incompatibility With Deflationary Tokens

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ParaSupply`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Paraluni` protocol, the `ParaSupply` contract is designed to take users' assets and deliver `NFT` as reward. In particular, one interface, i.e., `supplyFlex()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdrawFlex()`, allows the user to withdraw the asset. For the above two operations, i.e., `supplyFlex()` and `withdrawFlex()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` 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.

```
    function supplyFlex(uint _amount) external {
        _doTransferIn(msg.sender, tokenAddress, _amount);
        UserInfo storage user = users[msg.sender];
        updateInfo(msg.sender);
        user.demandAmount = user.demandAmount.add(_amount);
        emit Deposit(msg.sender, 0, _amount, 0, 0);
    }
```

Listing 3.17: `ParaSupply::supplyFlex()`

```
    function withdrawFlex() external returns (uint tokenId){
        UserInfo storage user = users[msg.sender];
        updateInfo(msg.sender);
        uint demandx = user.demandX;
        user.demandX = 0;
        (bool flag, uint _level) = isNft(demandx, user.demandAmount);
        if(flag){
            tokenId = _doTransferNFT(_level, msg.sender);
        }

        uint transferAmount = user.demandAmount;
        user.demandAmount = 0;
        _doTransferOut(msg.sender, tokenAddress, transferAmount);
        emit Withdraw(msg.sender, 0, 0, transferAmount, tokenId);
    }
```

Listing 3.18: `ParaSupply::withdrawFlex()`

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 `supplyFlex()` and `withdrawFlex()`, 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.

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 `ParaSupply` for support. Note that other routines share the same issue, including `supplyRegular()`, `withdrawRegular()`, from the same contract, `deposit()` and `withdraw()` from the `MasterChef` contract.

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: `349912c`.

PeckShield

4 Conclusion

In this audit, we have analyzed the `Paraluni` design and implementation. `Paraluni` is a decentralized automated market making (AMM) protocol on `Binance Smart Chain (BSC)`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

PeckShield

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

PeckShield

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

[11] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[17] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

PeckShield