SMART CONTRACT AUDIT REPORT

for

Liquid Finance

Prepared By: Xiaomi Huang

PeckShield

November 4, 2022

Document Properties

Client	Liquid Finance
Title	Smart Contract Audit Report
Target	Liquid Finance
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	November 4, 2022	Jing Wang	Final Release
1.0-rc	October 14, 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

1 Introduction

Given the opportunity to review the Liquid Finance 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 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 Liquid Finance

Liquid Finance is an innovative platform for liquidity provisioning that aims to address capital inefficiency and value accrual issues. The protocol attempts to solve the problems of liquidity provisioning using a two-token system comprised of lqETH and LIQD. lqETH is a fractional-reserve token pegged to the price of ETH, which can be minted and redeemed by the protocol. It maintains price stability and provides itself using the unique Liquid Arbitrage Mechanism (LAM). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Liquid Finance

Item	Description
Issuer	Liquid Finance
Website	https://liquidfinance.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 4, 2022

In the following, we show the list of reviewed contracts that are currently deployed on Arbitrum.

- https://arbiscan.io/address/0x73700aeCfC4621E112304B6eDC5BA9e36D7743D3 (lqETH)
- https://arbiscan.io/address/0x93C15cd7DE26f07265f0272E0b831C5D7fAb174f (LIQD)

- https://arbiscan.io/address/0xc7B3Cc8320C716D60e723836dA2064ED5754E038 (LIQD Reserve)
- https://arbiscan.io/address/0xB6a0ad0f714352830467725e619ea23E2C488f37 (lqETH LP)
- https://arbiscan.io/address/0x5dCF474814515B58ca0CA5e80bbB00d18C5B5cF8 (LIQD LP)
- https://arbiscan.io/address/0x705ea996D53Ff5bdEB3463dFf1890F83f57CDe97 (Pool)
- https://arbiscan.io/address/0x2582fFEa547509472B3F12d94a558bB83A48c007 (Chef)
- https://arbiscan.io/address/0xA1A988A22a03CbE0cF089e3E7d2e6Fcf9BD585A9 (Staking)
- https://arbiscan.io/address/0x61fb28d32447ef7F4e85Cf247CB9135b4E9886C2 (Treasury)
- https://arbiscan.io/address/0x74b353A2fd8608a7a0Cb9977121793B78Ed7259A (Bonds)
- https://arbiscan.io/address/0x50A9300688E6E6225081B454a23cec1fc623Ff0E (SwapStrategy-POL)
- https://arbiscan.io/address/0xB7C6CbC49fea52d56AA93456e1ea81172A30c285 (Bond Reserve)
- https://arbiscan.io/address/0x8BBD8457829bfE14590e2ba0Fa40Fd8919004183 (Bond Strategy)
- https://arbiscan.io/address/0x6d306e5f9b0b1aE6e74e6A9357f78d10f21F3128 (Team Allocation)
- https://arbiscan.io/address/0x7d0a6069dE1B73724Ce170C1D50E89A7F4a8F356 (lqETH LP Oracle)
- https://arbiscan.io/address/0x119a7CD5e1574615f51d7D1C3d8dC798C184a33C (LIQD LP Oracle)
- https://arbiscan.io/address/0x2Ad992a3ac3cF6DfF518932728b83a17dED124Df (Master Oracle)

1.2 About PeckShield

PeckShield Inc. [12] 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 the OWASP Risk Rating Methodology [11]:

- 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 checklist of 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Liquid Finance 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

- 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) [10], 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

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 Logic	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 implementation of the `Liquid Finance` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Liquid Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant Code Removal	Time and State	Confirmed
PVE-002	Low	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Coding Practices	Confirmed
PVE-003	Low	Potential Overflow Mitigation in _notifyReward()	Numeric Errors	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed

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

3 | Detailed Results

3.1 Redundant Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `presale`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The `presale` contract makes good use of a number of reference contracts, such as `Ownable`, `SafeMath`, and `ReentrancyGuard`, to facilitate its code implementation and organization. While the current implementation is rather thorough and solid, our analysis shows the (minor) inclusion of certain redundant code that can be safely removed.

For example, if we examine closely the `addWhitelistedWallet()` routine, this routine is designed to whitelisted wallet so that the user could buy the presaled tokens.

```
    function addWhitelistedWallet (address _user, uint256 _amount) public isWlAvailable
        (_amount) onlyOwner {
        require (!UserInfo[_user].isWhitelisted, "user already whitelisted");
        require (_amount + totalAllocated <= MAX_PRESALE, "amount pushes totalAllocated
            over max");
        UserInfo[_user].isWhitelisted = true;
        UserInfo[_user].allocation = _amount;
        nWhitelists = nWhitelists.add(1);
        wlWalletArray.push(_user);
        totalAllocated = totalAllocated + _amount;
    }

    modifier isWlAvailable (uint256 _amount) {
        require(_amount + totalAllocated <= MAX_PRESALE, "Not enough total presale
            allocation remaining");
        _;
```

```
126     }
```

Listing 3.1: `presale::addWhitelistedWallet()`

To elaborate, we show above the `addWhitelistedWallet()` routine from the `presale` contract. This routine in essence performs the check on whether the amount pushed `totalAllocated` exceed the `MAX_PRESALE` at line 115. However, the modifier `isWlAvailable()` evaluates the same check. With that, the check at line 115 can be safely removed.

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

Status This issue has been confirmed as the team clarifies the presale contract was only used in the presale and is no longer needed.

3.2 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- 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 3.2: 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, uint tokens)virtual public returns (bool success)`. 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. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `trade()` routine in the `StratBuyAssets` contract. If `USDT` is given as `_outToken`, the unsafe version of `IERC20(_inToken).transfer(address(treasury), IERC20(_inToken).balanceOf(address(this)));` (line 106) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)! We also suggest to address the same issue in other related routines.

```
    function trade (
        address _router,
        address _inToken,
        address _outToken,
        address[] memory _path,
        uint256 _amountIn,
        uint256 _minRecieve
    ) public onlyAdmin () returns (uint256) {
        require(acceptedAssets[_outToken],"StratBuyAssets::trade: asset not accepted");
        require(acceptedRouters[_router], "StratBuyAssets::createLP: must be accepted
            router");
        treasury.requestFund(_inToken, _amountIn);
        IERC20(_inToken).safeTransferFrom(address(treasury), address(this), _amountIn);
        IERC20(_inToken).safeIncreaseAllowance(_router, _amountIn);
        uint256[] memory _amounts = IUniswapV2Router02(_router).swapExactTokensForTokens
            (
            _amountIn,
            _minRecieve,
            _path,
            address(treasury),
            block.timestamp
```

```
        );
        if (IERC20(_inToken).balanceOf(address(this)) > 0) {
            IERC20(_inToken).transfer(address(treasury), IERC20(_inToken).balanceOf(
                address(this)));
        }
        if (IERC20(_outToken).balanceOf(address(this)) > 0) {
            IERC20(_outToken).transfer(address(treasury), IERC20(_outToken).balanceOf(
                address(this)));
        }
        return _amounts[_path.length - 1];
    }
```

Listing 3.3: `StratBuyAssets::trade()`

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

Status This issue has been confirmed as the team clarifies these are admin only contracts that are used by the team to acquire assets for the treasury. It is unlikely that they will interact with non-standard ERC-20 tokens, but as a precaution they will use the safe-version in future treasury strategies.

3.3 Potential Overflow Mitigation in _notifyReward()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `FASTStaking`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The `Liquid Finance` protocol has a built-in incentivizer mechanism, which is based on the popular `StakingRewards` from `Synthetix`. In this section, we focus on a routine, i.e., `_rewardPerToken()`, which is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `FASTStaking` to update and use the latest reward rate.

We notice a potential arithmetic overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `_rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 128 – 131), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistributors` (through the `notifyRewardAmount()` function).

```
    function lastTimeRewardApplicable(address _rewardsToken) public view returns (
        uint256) {
        return Math.min(block.timestamp, rewardData[_rewardsToken].periodFinish);
    }

    function _rewardPerToken(address _rewardsToken, uint256 _supply) internal view
        returns (uint256) {
        if (_supply == 0) {
            return rewardData[_rewardsToken].rewardPerTokenStored;
        }
        return
            rewardData[_rewardsToken].rewardPerTokenStored.add(
                lastTimeRewardApplicable(_rewardsToken)
                .sub(rewardData[_rewardsToken].lastUpdateTime)
                .mul(rewardData[_rewardsToken].rewardRate).mul(1e18).div(_supply)
            );
    }
```

Listing 3.4: `FASTStaking::rewardPerToken()`

```
    function _notifyReward(address _rewardsToken, uint256 reward) internal {
        if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
            rewardData[_rewardsToken].rewardRate = reward.div(rewardsDuration);
        } else {
            uint256 remaining = rewardData[_rewardsToken].periodFinish.sub(block.
                timestamp);
            uint256 leftover = remaining.mul(rewardData[_rewardsToken].rewardRate);
            rewardData[_rewardsToken].rewardRate = reward.add(leftover).div(
                rewardsDuration);
        }

        rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
        rewardData[_rewardsToken].periodFinish = block.timestamp.add(rewardsDuration);
    }
```

Listing 3.5: `FASTStaking::_notifyReward()`

Apparently, this issue is made possible if the reward amount is given as the argument to `_notifyReward()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Mitigating the potential overflow risk by ensuring no oversized reward amount will be provided.

Status This issue has been confirmed as the team clarifies they will follow the recommendations.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Pool
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Liquid Finance protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration). 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 account and its related privileged accesses in current contract.

To elaborate, we show the setFlashLoanContract() and related routines from the Liquid Finance contract. This function allows the owner account set the address of flashLoanContract which could drain all funds from the pool.

```
    function setFlashLoanContract (address _flashLoanContract) public onlyOwner () {
        require (flashLoanContract == address(0), "Pool::setFlashLoanContract: can only
            set once");
        require (_flashLoanContract != address(0), "Pool::setFlashLoanContract: invalid
            address");
        flashLoanContract = _flashLoanContract;
    }

    function toggleFlashLoansActive (bool _flashLoansActive) public onlyOwner () {
        require(address(flashLoanContract) != address(0), "Pool::setFlashLoanContract:
            flashloan contract not set");
        flashLoansActive = _flashLoansActive;
    }

    function sendToFlashLoanContract (uint256 _amount) external {
        require (flashLoansActive, "Pool::sendToFlashLoanConract: flashloans are not
            active");
        require (msg.sender == flashLoanContract, "Pool::sendToFlashLoanContract: only
            flash loan contract");
        require (_amount <= WethUtils.weth.balanceOf(address(this)), "Pool::
            sendToFlashLoanContract: amount exceeds weth balance");
        WethUtils.weth.transfer(flashLoanContract, _amount);
    }
```

Listing 3.6: Liquid Finance::setFlashLoanContract()

We understand the need of the privileged functions for contract maintenance, but 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 This issue has been confirmed. The team clarifies they plan on using an `EOA` to start, and eventually migrating ownership of sensitive contracts to multi-sig in the future and switch to `DAO`-like governance contract.

3.5 Incompatibility with Deflationary Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FASTChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Liquid Finance` protocol, the `FASTChef` contract is designed to receive users' assets and deliver rewards depending on their share. In particular, one user-facing function, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another function, 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.

```
    /// @notice Deposit LP tokens to MCV2 for reward allocation.
    /// @param pid The index of the pool. See `poolInfo`.
    /// @param amount LP token amount to deposit.
    /// @param to The receiver of `amount` deposit benefit.
    function deposit(
        uint256 pid,
        uint256 amount,
        address to
    ) public {
        PoolInfo memory pool = updatePool(pid);
        UserInfo storage user = userInfo[pid][to];
```

```
        // Effects
        user.amount += amount;
        user.rewardDebt += int256((amount * pool.accRewardPerShare) /
            ACC_REWARD_PRECISION);

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

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

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

    /// @notice Withdraw LP tokens from MCV2.
    /// @param pid The index of the pool. See `poolInfo`.
    /// @param amount LP token amount to withdraw.
    /// @param to Receiver of the LP tokens.
    function withdraw(
        uint256 pid,
        uint256 amount,
        address to
    ) public {
        PoolInfo memory pool = updatePool(pid);
        UserInfo storage user = userInfo[pid][msg.sender];

        // Effects
        user.rewardDebt -= int256((amount * pool.accRewardPerShare) /
            ACC_REWARD_PRECISION);
        user.amount -= amount;

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

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

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

Listing 3.7: `FASTChef::deposit()`and `FASTChef::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.accRewardPerShare` via dividing `rewardAmount` by `lpSupply`, where the `lpSupply` is derived from `lpToken[pid].balanceOf(address(this))` (line 74). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accRewardPerShare` as the final result, which dramatically inflates the pool's reward.

```
68      /// @notice Update reward variables of the given pool.
69      /// @param pid The index of the pool. See `poolInfo`.
70      /// @return pool Returns the pool that was updated.
71      function updatePool(uint256 pid) public returns (PoolInfo memory pool) {
72          pool = poolInfo[pid];
73          if (block.timestamp > pool.lastRewardTime) {
74              uint256 lpSupply = lpToken[pid].balanceOf(address(this));
75              if (lpSupply > 0) {
76                  uint256 time = block.timestamp - pool.lastRewardTime;
77                  uint256 rewardAmount = (time * rewardPerSecond * pool.allocPoint) /
                        totalAllocPoint;
78                  pool.accRewardPerShare += (rewardAmount * ACC_REWARD_PRECISION) /
                        lpSupply;
79              }
80              pool.lastRewardTime = block.timestamp;
81              poolInfo[pid] = pool;
82              emit LogUpdatePool(pid, pool.lastRewardTime, lpSupply, pool.
                    accRewardPerShare);
83          }
84      }
```

Listing 3.8: `FASTChef::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 `FASTChef` 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 This issue has been confirmed. The team clarifies the `FASTChef` contract will not support deflationary tokens.

4 Conclusion

In this audit, we have analyzed the Liquid Finance design and implementation. Liquid Finance is an innovative platform for liquidity provisioning that solves capital inefficiency and value accrual issues. Those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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