SMART CONTRACT AUDIT REPORT

for

Mojito Protocol

Prepared By: Yiqun Chen

PeckShield

September 18, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 18, 2021	Xuxian Jiang	Final Release
1.0-rc	September 17, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	unavailable
Email	contact@peckshield.com

Contents

1 Introduction

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

The Mojito protocol is a decentralized exchange that comes with Automated Market Maker and a high APR. It is designed with necessary governance support and the popular farming support which allows users to earn rewards by staking supported assets. Overall the protocol provides users an attractive environment for their assets and a place to potentially yield a high return.

The basic information of the MojitoSwap protocol is as follows:

Table 1.1: Basic Information of The MojitoSwap Protocol

Item	Description
Name	Mojito
Website	https://www.mojitoswap.finance/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 18, 2021

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

- https://github.com/MojitoFinance/governance.git (928293a)

- https://github.com/MojitoFinance/merkle-distributor.git (c7b7a7e)
- https://github.com/MojitoFinance/mojito-lib.git (870eea8)
- https://github.com/MojitoFinance/mojito-swap-core.git (f2dd231)
- https://github.com/MojitoFinance/mojito-swap-farm.git (86b318f)
- https://github.com/MojitoFinance/mojito-swap-periphery.git (a75f9af)

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 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 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) [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.

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 1.3: The Full List of Check Items

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

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

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	5	■■■■■
Informational	0	
Undetermined	2	■■
Total	9	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 2 undetermined issues.

Table 2.1: Key Mojito Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Swap Fee Uses in computeProfitMaximizingTrade()	Business Logic	Fixed
PVE-002	Medium	Proper feeToDenominator Uses in computeLiquidityValue()	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Improved Handling of Corner Cases in Proposal Submission	Business Logic	Fixed
PVE-005	Undetermined	Mojito Total Supply Threshold	Business Logic	Confirmed
PVE-006	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-007	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-008	Undetermined	MasterChef Incompatibility With Deflationary Tokens	Business Logics	Confirmed
PVE-009	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-010	Low	Possible Costly Pool Shares From Improper Vault Initialization	Time and State	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Swap Fee Uses in computeProfitMaximizingTrade()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: MojitoLiquidityMathLibrary
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The MojitoSwap protocol has the built-in DEX functionality that is inspired from UniswapV2, but with the extension of flexible support for reconfigurable trading fee and protocol fee. Both fees can be dynamically configured via the MojitoFactory contract on each individual pool pair. In the analysis of the mojito-swap-periphery repository, we notice the current code base has an inconsistent use of the swap fee.

To elaborate, we show below the computeProfitMaximizingTrade() routine inside the the library contract MojitoLiquidityMathLibrary. This function is designed to compute the direction and magnitude of the profit-maximizing trade and has the need of taking into account the swap fee for the related swap pair. However, it comes to our attention that this routine uses the hardcoded trading fee of 0.3%, which may not be consistent especially when the swap fee may be dynamically configured for each individual swap pair.

```
    // computes the direction and magnitude of the profit-maximizing trade
    function computeProfitMaximizingTrade(
        uint256 truePriceTokenA,
        uint256 truePriceTokenB,
        uint256 reserveA,
        uint256 reserveB
    ) pure internal returns (bool aToB, uint256 amountIn) {
        aToB = FullMath.mulDiv(reserveA, truePriceTokenB, reserveB) < truePriceTokenA;

        uint256 invariant = reserveA.mul(reserveB);

```

```
        uint256 leftSide = Babylonian.sqrt(
            FullMath.mulDiv(
                invariant.mul(1000),
                aToB ? truePriceTokenA : truePriceTokenB,
                (aToB ? truePriceTokenB : truePriceTokenA).mul(997)
            )
        );
        uint256 rightSide = (aToB ? reserveA.mul(1000) : reserveB.mul(1000)) / 997;

        if (leftSide < rightSide) return (false, 0);

        // compute the amount that must be sent to move the price to the profit-
            maximizing price
        amountIn = leftSide.sub(rightSide);
    }
```

Listing 3.1: MojitoLiquidityMathLibrary :: computeProfitMaximizingTrade()

Recommendation Revise the above `computeProfitMaximizingTrade()` routine to consider possible different trading fee for different swap pair.

Status This issue has been fixed in the following PR: 1.

3.2 Proper feeToDenominator Uses in computeLiquidityValue()

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `MojitoLiquidityMathLibrary`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

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

To elaborate, we show below this `computeLiquidityValue()` routine. This routine implements a rather straightforward logic in computing the liquidity value given all six parameters of the pair, i.e., `reservesA`, `reservesB`, `totalSupply`, `liquidityAmount`, `feeOn`, and `kLast`. Notice that this routine uses the fixed `1/6` of collected swap fee for protocol fee, without accommodating the possibility that the percentage may be dynamically configured on a pair-basis.

```
    // computes liquidity value given all the parameters of the pair
```

```
    function computeLiquidityValue(
        uint256 reservesA,
        uint256 reservesB,
        uint256 totalSupply,
        uint256 liquidityAmount,
        bool feeOn,
        uint kLast
    ) internal pure returns (uint256 tokenAAmount, uint256 tokenBAmount) {
        if (feeOn && kLast > 0) {
            uint rootK = Babylonian.sqrt(reservesA.mul(reservesB));
            uint rootKLast = Babylonian.sqrt(kLast);
            if (rootK > rootKLast) {
                uint numerator1 = totalSupply;
                uint numerator2 = rootK.sub(rootKLast);
                uint denominator = rootK.mul(5).add(rootKLast);
                uint feeLiquidity = FullMath.mulDiv(numerator1, numerator2, denominator);
                totalSupply = totalSupply.add(feeLiquidity);
            }
        }
        return (reservesA.mul(liquidityAmount) / totalSupply, reservesB.mul(liquidityAmount) / totalSupply);
    }
```

Listing 3.2: MojitoLiquidityMathLibrary::computeLiquidityValue()

Recommendation Revise the above `computeLiquidityValue()` routine to consider possible different protocol fee that may be changed for different swap pair.

Status This issue has been fixed in the following PR: 1.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `MerkleDistributor`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value`

`&& balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.3: ZRX.sol

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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `refund()` routine in the `MerkleDistributor` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).transfer(treasury, IERC20(token).balanceOf(address(this)))` (line 55) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
53      function refund() external override {
54          require(block.timestamp > deadline, 'MerkleDistributor: Drop is not over yet.');
55          require(IERC20(token).transfer(treasury, IERC20(token).balanceOf(address(this)))
            , 'MerkleDistributor: Transfer failed.');
56      }
```

Listing 3.4: `MerkleDistributor::refund()`

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

Status This issue has been resolved as the token used in `refund()` will be restricted to the `Mojito` token.

3.4 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GovernorAlpha`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Mojito` protocol adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. In this section, we elaborate one corner case when a proposal is submitted regarding the proposer qualification.

Specifically, to be qualified as a proposer, the governance subsystem requires the proposer to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`. In `Mojito`, this number requires the votes of 1_000_000e18 (about 1% of `Mojito` token's total supply).

```
    function propose(address[] memory targets, uint[] memory values, string[] memory
        signatures, bytes[] memory calldatas, string memory description) public returns
        (uint) {
        require(mojito.getPriorVotes(msg.sender, sub256(block.number, 1)) >
            proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal
            threshold");
        require(targets.length == values.length && targets.length == signatures.length
            && targets.length == calldatas.length, "GovernorAlpha::propose: proposal
            function information arity mismatch");
        require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
        require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too
            many actions");

        uint latestProposalId = latestProposalIds[msg.sender];
        if (latestProposalId != 0) {
          ProposalState proposersLatestProposalState = state(latestProposalId);
          require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::
              propose: one live proposal per proposer, found an already active proposal"
              );
```

```
142             require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
143         }
144         ...
145     }
```

Listing 3.5: GovernorAlpha::propose()

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies upfront the qualification of the proposer (line 133): `require(mojito.getPriorVotes(msg.sender, sub256(block.number, 1))> proposalThreshold())`. Note that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 203) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
198     function cancel(uint proposalId) public {
199         ProposalState state = state(proposalId);
200         require(state != ProposalState.Executed, "GovernorAlpha::cancel: cannot cancel executed proposal");

202         Proposal storage proposal = proposals[proposalId];
203         require(mojito.getPriorVotes(proposal.proposer, sub256(block.number, 1)) < proposalThreshold(), "GovernorAlpha::cancel: proposer above threshold");

205         proposal.canceled = true;
206         for (uint i = 0; i < proposal.targets.length; i++) {
207             timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal.signatures[i], proposal.calldatas[i], proposal.eta);
208         }

210         emit ProposalCanceled(proposalId);
211     }
```

Listing 3.6: GovernorAlpha::cancel()

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold()`.

Status This issue has been fixed in the following PR: 1.

3.5 Mojito Total Supply Threshold

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

As mentioned earlier, the `Mojito` protocol adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. In particular, the `quorumVotes()` defines the number of votes in support of a proposal required in order for a quorum to be reached and for a vote to succeed. Currently, it is defined as a constant value, i.e., `4_000_000e18` or expected 4% of `Mojito` total supply.

Similarly, the `proposalThreshold()` defines the number of votes required in order for a voter to become a proposer. Currently, it is also defined as a constant value, i.e., `1_000_000e18` or the expected 1% of `Mojito` total supply.

However, our analysis shows the governance token `MojitoToken` does not have a fixed total supply, which may render the above constants `quorumVotes()` and `proposalThreshold()` inappropriate. With that, we make the suggestion to revisit the tokenomics behind the `MojitoToken` design especially on the governance support.

```
contract GovernorAlpha {
    /// @notice The name of this contract
    string public constant name = "MojitoSwap Governor Alpha";

    /// @notice The number of votes in support of a proposal required in order for a quorum to be reached and for a vote to succeed
    function quorumVotes() public pure returns (uint) { return 4_000_000e18; } // 4% of Mojito

    /// @notice The number of votes required in order for a voter to become a proposer
    function proposalThreshold() public pure returns (uint) { return 1_000_000e18; } // 1% of Mojito

    /// @notice The maximum number of actions that can be included in a proposal
    ...
}
```

Listing 3.7: `GovernorAlpha::quorumVotes()/proposalThreshold()`

Recommendation Revisit the tokenomics design to ensure the governance subsystem has the proper `quorumVotes()` and `proposalThreshold()` in place.

Status The issue has been confirmed by the team.

3.6 Duplicate Pool Detection and Prevention

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Mojito` 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,
        accMojitoPerShare: 0
        }));
```

```
        updateStakingPool();
    }
```

Listing 3.8: 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?");
        }
    }

    // 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,
        accMojitoPerShare : 0
        }));
        updateStakingPool();
    }
```

Listing 3.9: Revised TokenMaster::add()

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

Status This issue has been fixed in the following PR: 1.

3.7 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

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

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

```
    // Update the given pool's Mojito 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
            );
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            updateStakingPool();
        }
    }
```

Listing 3.10: 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 Mojito 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
            );
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (prevAllocPoint != _allocPoint) {
            updateStakingPool();
        }
    }
```

Listing 3.11: Revised MasterChef::set()

Status This issue has been fixed in the following PR: 2.

3.8 MasterChef Incompatibility With Deflationary Tokens

- ID: PVE-008
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MasterChef`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

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

```
    // Deposit LP tokens to MasterChef.
    function deposit(uint256 _pid, uint256 _amount) public {
        require(_pid != 0, "MasterChef::deposit: _pid can only be farm pool");

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accMojitoPerShare).div(1e12).sub(user
                .rewardDebt);
            if (pending > 0) {
```

```
                safeMojitoTransfer(msg.sender, pending);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accMojitoPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        require(_pid != 0, "MasterChef::withdraw: _pid can only be farm pool");

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "MasterChef::withdraw: _amount not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accMojitoPerShare).div(1e12).sub(user.
            rewardDebt);
        if (pending > 0) {
            safeMojitoTransfer(msg.sender, pending);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accMojitoPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.12: `MasterChef::deposit()/withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. 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.accMojitoPerShare` via dividing `mojitoReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 188). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accMojitoPerShare` as the final result, which dramatically inflates the pool's reward.

```
182     // Update reward variables of the given pool to be up-to-date.
183     function updatePool(uint256 _pid) public {
```

```
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 blockReward = mintable(pool.lastRewardBlock);
        uint256 mojitoReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        mojito.mint(address(this), mojitoReward);
        pool.accMojitoPerShare = pool.accMojitoPerShare.add(mojitoReward.mul(1e12).div(
            lpSupply));
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.13: `MasterChef::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 `Mojito` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

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

Status This issue has been confirmed.

3.9 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

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

Apparently, the interaction with the external contract (line 284) starts before effecting the update on internal states (lines 286−287), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
280     // Withdraw without caring about rewards. EMERGENCY ONLY.
281     function emergencyWithdraw(uint256 _pid) public {
282         PoolInfo storage pool = poolInfo[_pid];
283         UserInfo storage user = userInfo[_pid][msg.sender];
284         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
285         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
286         user.amount = 0;
287         user.rewardDebt = 0;
288     }
```

Listing 3.14: `MasterChef::emergencyUnstake()`

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

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

Status This issue has been fixed in the following PR: 3.

3.10 Possible Costly Pool Shares From Improper Vault Initialization

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MojitoVault`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [1]

Description

The `Mojito` protocol allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This routine is used for participating users to deposit the supported assets (e.g., `MJT`) and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function deposit(uint256 _amount) external whenNotPaused notContract {
        require(_amount > 0, "MojitoVault::deposit: nothing to deposit");

        uint256 pool = balanceOf();
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 currentShares = 0;
        if (totalShares != 0) {
            currentShares = (_amount.mul(totalShares)).div(pool);
        } else {
            currentShares = _amount;
        }
        UserInfo storage user = userInfo[msg.sender];

        user.shares = user.shares.add(currentShares);
        user.lastDepositedTime = block.timestamp;

        totalShares = totalShares.add(currentShares);

        user.mojitoAtLastUserAction = user.shares.mul(balanceOf()).div(totalShares);
        user.lastUserActionTime = block.timestamp;

```

```
109         _earn();
110 
111         emit Deposit(msg.sender, _amount, currentShares, block.timestamp);
112     }
```

Listing 3.15: MojitoVault::deposit()

Specifically, when the pool is being initialized (line 96), the share value directly takes the value of `_amount` (line 97), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `user.shares = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of `MJT` assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been confirmed. And the team decides to mitigate this issue by properly following a guarded launch process.

4 Conclusion

In this audit, we have analyzed the `Mojito` protocol design and implementation. The `Mojito` protocol provides a decentralized exchange that comes with `Automated Market Maker` and a high `APR`. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

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

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