SMART CONTRACT AUDIT REPORT

for

MilkySwap

Prepared By: Yiqun Chen

Hangzhou, China
March 2, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	March 2, 2022	Jing Wang	Final Release
1.0-rc	March 1, 2022	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the MilkySwap 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 MilkySwap 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 MilkySwap

MilkySwap is a custom-built decentralized exchange built on the Milkomeda sidechain for Cardano . MilkySwap allows users to exchange tokens and bridge their Cardano tokens with Ethereum. The protocol forks from the Sushiswap protocol for the core AMM DEX and farming model. The protocol also forks from the CurveDAO for rewards mechanisms.

The basic information of MilkySwap is as follows:

Table 1.1: Basic Information of MilkySwap

Item	Description
Name	MilkySwap
Website	https://www.milkyswap.exchange/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 2, 2022

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

- https://github.com/milkyswap/milkyswap (59f163e)

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

- https://github.com/milkyswap/milkyswap (882ad9e)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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

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

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) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `MilkySwap` 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	2	■■
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key MilkySwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Reentrancy Risk in MasterMilker	Time and State	Confirmed
PVE-002	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed
PVE-003	High	Voting Amplification With Sybil Attacks	Business Logic	Fixed
PVE-004	High	Converter Permission Bypass with MilkyMaker::convertMultiple()	Business Logic	Fixed
PVE-005	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Confirmed
PVE-006	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Medium	Improper Funding Source In CreamyToken::_deposit_for()	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 Reentrancy Risk in MasterMilker

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterMilker
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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 [16] exploit, and the recent Uniswap/Lendf.Me hack [15].

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

Apparently, the interaction with the external contract (line 240) starts before effecting the update on the internal state (lines 241-242), 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.

```
235     // Withdraw without caring about rewards. EMERGENCY ONLY.
236     function emergencyWithdraw(uint256 _pid) public {
237         PoolInfo storage pool = poolInfo[_pid];
238         UserInfo storage user = userInfo[_pid][msg.sender];
239         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
```

```
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

Listing 3.1: `MasterMilker::emergencyWithdraw()`

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

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

Status This issue has been confirmed. The team clarifies they will only use `MilkySwap LP` tokens as the pool token so there is no reentrancy risk.

3.2 Incompatibility with Deflationary Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterMilker`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `MilkySwap` protocol, the `MasterMilker` 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 `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    // Deposit LP tokens to MasterMilker for MILKY allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accMilkyPerShare).div(1e12).sub(
                    user.rewardDebt
                );
```

```
            safeMilkyTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accMilkyPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from MasterMilker.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending =
            user.amount.mul(pool.accMilkyPerShare).div(1e12).sub(
                user.rewardDebt
            );
        safeMilkyTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accMilkyPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.2: `MasterMilker::deposit()`and `MasterMilker::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()` 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.accMilkyPerShare` via dividing `milkyReward` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 178). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accMilkyPerShare` as the final result, which dramatically inflates the pool's reward.

```
172     // Update reward variables of the given pool to be up-to-date.
173     function updatePool(uint256 _pid) public {
174         PoolInfo storage pool = poolInfo[_pid];
175         if (block.number <= pool.lastRewardBlock) {
176             return;
177         }
```

```
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 milkyReward =
            multiplier.mul(milkyPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint
            );
        milky.mint(devaddr, milkyReward.div(10));
        milky.mint(address(this), milkyReward);
        pool.accMilkyPerShare = pool.accMilkyPerShare.add(
            milkyReward.mul(1e12).div(lpSupply)
        );
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.3: `MasterMilker::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 `MilkySwap` protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note other routines, i.e., `deposit()` and `withdraw()`, from the `CreamyToken` contract share the same issue.

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 they will only allowing `MilkySwap LP` tokens into the `MasterMilker` contract.

3.3 Voting Amplification With Sybil Attacks

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `MilkyToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `MILKY` tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the `MilkyToken` 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 `MILKY` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `MILKY`s. 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 MILKYs (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];
            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 3.4: `MilkyToken.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 `MILKY` 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 `MILKYs`. When M_1 delegates to `Trudy`, since M_1 now has 100 `MILKYs`, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `MILKY` 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! Note the same issue also exists on `SYRUP`.

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: 882ad9e.

3.4 Converter Permission Bypass with MilkyMaker::convertMultiple()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: MilkyMaker
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

In the MilkySwap protocol, the MilkyMaker contract is designed to trade tokens collected from fees for MILKY and serves up rewards for CREAMY holders. Specifically, the convert() routine is implemented to swap tokens and it has a permission check of whether the msg.sender is an authorized converter. To elaborate, we show below its full implementation.

```
    function convert(address token0, address token1) external onlyEOA {
        require(_converters[msg.sender], "sender not authorized to call convert");
        _convert(token0, token1);
    }
```

Listing 3.5: MilkyMaker::convert()

It comes to our attention that the permission check for converter is not applied in the convertMultiple () routine, which is initially designed to save gas when the converter has multiple token pairs to convert. A bad actor could call the convertMultiple() routine to force the MilkyMaker contract to remove liquidity and swap in a manipulated imbalance pool to exploit.

```
    function convertMultiple(
        address[] calldata token0,
        address[] calldata token1
    ) external onlyEOA {
        // TODO: This can be optimized a fair bit, but this is safer and simpler for now
        uint256 len = token0.length;
        for (uint256 i = 0; i < len; i++) {
            _convert(token0[i], token1[i]);
        }
    }
```

Listing 3.6: MilkyMaker::convertMultiple()

Recommendation Add the same permission check for converter in the convertMultiple() routine

Status The issue has been fixed in the following commit: dc67d4b.

3.5 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV2Router02`
- Category: Coding Practices [8]
- CWE subcategory: CWE-628 [3]

Description

In the `UniswapV2Router02` contract, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `UniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

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

Listing 3.7: `UniswapV2Router02::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 (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
```

```
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.8: `UniswapV2Router02::_addLiquidity()`

It comes to our attention that the `UniswapV2 Router` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `UniswapV2 Router` may not be checked and may not be taken into account at all in certain scenarios.

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

Status This issue has been confirmed. The team clarifies they will make sure the implicit assumption is always enforced when users interact with the `MilkySwap` frontend.

3.6 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MilkyMaker
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

As mentioned in Section 3.4, the MilkyMaker contract is designed to trade tokens collected from fees for MILKY and serves up rewards for CREAMY holders. The MilkyMaker contract has a helper routine, i.e., _convert(), that is designed to swap token0 for token1. It has a rather straightforward logic in removing liquidity and transferring the funds to specific pairs to actually perform the intended token swap by _convertStep().

```
    function _convert(address token0, address token1) internal {
        // Interactions
        // S1 - S4: OK
        IUniswapV2Pair pair = IUniswapV2Pair(factory.getPair(token0, token1));
        require(address(pair) != address(0), "MilkyMaker: Invalid pair");
        // balanceOf: S1 - S4: OK
        // transfer: X1 - X5: OK
        IERC20(address(pair)).safeTransfer(
            address(pair),
            pair.balanceOf(address(this))
        );
        // X1 - X5: OK
        (uint256 amount0, uint256 amount1) = pair.burn(address(this));
        if (token0 != pair.token0()) {
            (amount0, amount1) = (amount1, amount0);
        }
        emit LogConvert(
            msg.sender,
            token0,
            token1,
            amount0,
            amount1,
            _convertStep(token0, token1, amount0, amount1)
        );
    }
    // F1 - F10: OK
    // C1 - C24: OK
    // All safeTransfer, _swap, _toMILKY, _convertStep: X1 - X5: OK
    function _convertStep(
        address token0,
        address token1,
        uint256 amount0,
```

```
        uint256 amount1
    ) internal returns (uint256 milkyOut) {
        // Interactions
        if (token0 == token1) {
            uint256 amount = amount0.add(amount1);
            if (token0 == milky) {
                IERC20(milky).safeTransfer(dest, amount);
                milkyOut = amount;
            } else if (token0 == wada) {
                milkyOut = _toMILKY(wada, amount);
            } else {
                address bridge = bridgeFor(token0);
                amount = _swap(token0, bridge, amount, address(this));
                milkyOut = _convertStep(bridge, bridge, amount, 0);
            }
        }
        ...
    }

    // F1 - F10: OK
    // C1 - C24: OK
    // All safeTransfer, swap: X1 - X5: OK
    function _swap(
        address fromToken,
        address toToken,
        uint256 amountIn,
        address to
    ) internal returns (uint256 amountOut) {
        // Checks
        // X1 - X5: OK
        IUniswapV2Pair pair = IUniswapV2Pair(
            factory.getPair(fromToken, toToken)
        );
        require(address(pair) != address(0), "MilkyMaker: Cannot convert");

        // Interactions
        // X1 - X5: OK
        (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
        uint256 amountInWithFee = amountIn.mul(997);
        if (fromToken == pair.token0()) {
            amountOut =
                amountInWithFee.mul(reserve1) /
                reserve0.mul(1000).add(amountInWithFee);
            IERC20(fromToken).safeTransfer(address(pair), amountIn);
            pair.swap(0, amountOut, to, new bytes(0));
            // TODO: Add maximum slippage?
        } else {
            amountOut =
                amountInWithFee.mul(reserve0) /
                reserve1.mul(1000).add(amountInWithFee);
            IERC20(fromToken).safeTransfer(address(pair), amountIn);
            pair.swap(amountOut, 0, to, new bytes(0));
```

```
            // TODO: Add maximum slippage?
        }
    }
```

Listing 3.9: `MilkyMaker.sol`

To elaborate, we show above related routines. We notice the remove liquidity and token swap are routed to `pair` and the actual removal or swap operation via `pair.burn(address(this))` or `_swap(token0, bridge, amount, address(this))` essentially do not specify any restriction with output amount 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.

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

Status This issue has been confirmed. The team clarifies they plan on calling the `convert()` function frequently, which alleviates potential sandwiching.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `MilkySwap` 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 in emergency). 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 `updateEmissionRate()` routine to set system parameter. This function allow the `owner` account to change the value of `milkyPerBlock`, which play an important role in computing `milkyReward`.

```
    function updateEmissionRate(uint256 _milkyPerBlock) public onlyOwner {
        massUpdatePools();
        milkyPerBlock = _milkyPerBlock;
    }
```

Also, we show the `kill_me()` routine, which is used to transfer the entire reward tokens to the `emergency_return` account and block the contract's ability to claim or burn.

```
    @external
    def kill_me():
        """
        @notice Kill the contract
        @dev Killing transfers the entire 3CRV balance to the emergency return address
             and blocks the ability to claim or burn. The contract cannot be unkilled.
        """
        assert msg.sender == self.admin

        self.is_killed = True

        token: address = self.token
        assert ERC20(token).transfer(self.emergency_return, ERC20(token).balanceOf(self)
            )
```

Listing 3.10: `FeeDistributor.vy`

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 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 ownership as part of `Phase II` on their `roadmap`.

3.8 Improper Funding Source In CreamyToken::_deposit_for()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CreamyToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `MilkySwap` has a key `CreamyToken` contract that provides the functionality of computing the time-dependent vote weights. By design, the vote weight decays linearly over time and the lock time cannot be more than `MAXTIME` (4 years). While reviewing the current locking logic, we notice the key helper routine `_deposit_for()` needs to be revised.

To elaborate, we show below the implementation of this `_deposit_for()` helper routine. In fact, it is an internal function to perform deposit and lock tokens for a user. This routine has a number of arguments and the first one `_addr` is the address to receive the balance. It comes to our attention that the `_addr` address is also the one to actually provide the assets, `assert ERC20(self.token).transferFrom(_addr, self, _value)` (line 377). In fact, the `msg.sender` should be the one to provide the assets for locking! Otherwise, this function may be abused to lock tokens from users who have approved the locking contract before without their notice.

```
350     @internal
351     def _deposit_for(_addr: address, _value: uint256, unlock_time: uint256,
            locked_balance: LockedBalance, type: int128):
352         """
353         @notice Deposit and lock tokens for a user
354         @param _addr User's wallet address
355         @param _value Amount to deposit
356         @param unlock_time New time when to unlock the tokens, or 0 if unchanged
357         @param locked_balance Previous locked amount / timestamp
358         """
359         _locked: LockedBalance = locked_balance
360         supply_before: uint256 = self.supply

362         self.supply = supply_before + _value
363         old_locked: LockedBalance = _locked
364         # Adding to existing lock, or if a lock is expired - creating a new one
365         _locked.amount += convert(_value, int128)
366         if unlock_time != 0:
367             _locked.end = unlock_time
368         self.locked[_addr] = _locked

370         # Possibilities:
371         # Both old_locked.end could be current or expired (>/< block.timestamp)
372         # value == 0 (extend lock) or value > 0 (add to lock or extend lock)
```

```
        # _locked.end > block.timestamp (always)
        self._checkpoint(_addr, old_locked, _locked)

        if _value != 0:
            assert ERC20(self.token).transferFrom(_addr, self, _value)

        log Deposit(_addr, _value, _locked.end, type, block.timestamp)
        log Supply(supply_before, supply_before + _value)
```

Listing 3.11: `VotingEscrow::_depositFor()`

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the MilkySwap design and implementation. MilkySwap is a custom-built decentralized exchange built on the Milkomeda sidechain for Cardano. 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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