SMART CONTRACT AUDIT REPORT

for

MorpheusSwap

Prepared By: Patrick Lou

PeckShield

April 11, 2022

Document Properties

Client	MorpheusSwap Finance
Title	Smart Contract Audit Report
Target	MorpheusSwap
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 11, 2022	Luck Hu	Final Release
1.0-rc	March 21, 2022	Luck Hu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the MorpheusSwap 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 contract can be further improved due to the presence of several issues related to business logic, security or performance. This document outlines our audit results.

1.1 About MorpheusSwap

Morpheus Swap is a decentralized exchange that is powered by Fantom Opera, with the main goal to provide the highest revenue share for a community-owned DEX. By staking the core token PILLS, users are entitled to a percentage of all protocol revenue that may be paid out in other tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the MorpheusSwap

Item	Description
Name	MorpheusSwap Finance
Website	https://morpheusswap.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 11, 2022

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

- https://github.com/daedboi/morpheus_contracts.git (6cb89ae)

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

- https://github.com/daedboi/morpheus_contracts.git (ac37d02)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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) [7], 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 `MorpheusSwap` 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	4	■■■■
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 some 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key MorpheusSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools() During Pool Weight Changes	Business Logic	Confirmed
PVE-002	Low	Inaccurate Calculation For multiplier in getRewardRate()	Business Logic	Fixed
PVE-003	Medium	Inconsistent Fee Calculation Between Matrix And PancakePair	Business Logic	Fixed
PVE-004	Low	Inaccurate wFTM Reward Supply in updateRewardPerSec()	Business Logic	Fixed
PVE-005	Medium	Incorrect Amount Return of Target Token in Zapper	Business Logic	Fixed
PVE-006	Low	Sybil Attacks on PILLS And MORPH Voting	Business Logic	Confirmed
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-008	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	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 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterChef`, `MasterChefV2`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `MasterChef` contract provides an incentive mechanism that rewards the staking of supported assets with the `MORPH` token. 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. Note the `MasterChefV2` contract shares the same issue.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) external onlyOwner {
        // No matter _withUpdate is true or false, we need to execute updatePool once
            before set the pool parameters.
        updatePool(_pid);

        if (_withUpdate) {
            massUpdatePools();
        }

        if (poolInfo[_pid].isRegular) {
```

```
            totalRegularAllocPoint = totalRegularAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        }
        poolInfo[_pid].allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.1: 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, these interfaces are 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 _withUpdate parameter to the set() and add() routines can be simply ignored or removed.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) external onlyOwner {
        // No matter _withUpdate is true or false, we need to execute updatePool once
            before set the pool parameters.
        updatePool(_pid);

        if (_withUpdate) {
            massUpdatePools();
        }

        if (poolInfo[_pid].isRegular) {
            totalRegularAllocPoint = totalRegularAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        }
        poolInfo[_pid].allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.2: Revised MasterChef::set()

Status This issue has been confirmed. The Morpheus team clarified that they have been aware of updating the pools before pool weights change.

3.2 Inaccurate Calculation For multiplier in getRewardRate()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `NeoPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `NeoPool` contract is a continuous revenue share staking pool where users stake `PILLS` to earn `wFTM`. Whenever the `updateRewardPerSec()` routine is called, it will calculate the `wFTM` per second to distribute for the next 7 days based off the amount of `wFTM` in the contract. If there is no `wFTM` in the contract, the `wFTM` per second to distribute will be set to 0 until the update function is called again. Each time the `updateRewardPerSec()` routine is called, a new reward `segment` (reward window) is created. And the `updatePool()` routine is used to accumulate the new distributed `wFTM` rewards per share to the pool. The new overall distributed `wFTM` rewards are calculated in the `getRewardRate()` routine. When analyzing the logic in the `getRewardRate()` routine, we notice there is a logic error in current implementation.

To elaborate, we show below the full implementation of the `getRewardRate()` routine. When the `updateRewardPerSec()` routine is called to create a new reward `segment`, it will also call the `updatePool()` routine to accumulate all the `wFTM` rewards distributed in the previous `segment` (if any). In particular, within the `getRewardRate()` routine (line 740), it counts the full `segment` period into the `multiplier` (the passed time in second). There is no problem if this is the first time to accumulate the new distributed `wFTM` rewards in this `segment`. Otherwise, if the `wFTM` rewards for this `segment` have been accumulated before (e.g, triggered from a deposit or withdraw action), the `multiplier` is counted larger than expectation. The expected value for the `multiplier` shall be `multiplier = rewardUpdateTimestamps[j + 1] - lastRewardTimestamp;`.

```
712     function getRewardRate() public view returns (uint rewards) {
713         for (uint j = 0; j < rewardUpdateTimestamps.length; j++) {
714             uint256 multiplier = 0;
715             if (j == rewardUpdateTimestamps.length - 2) {
716                 // if we have reached the end of the rewards
717                 if(rewardUpdateTimestamps[j + 1] <= block.timestamp)
718                     multiplier = rewardUpdateTimestamps[j + 1] - lastRewardTimestamp;
719                 // if the last reward timestamp was before a new segment started
720                 // the time since the start of this segment
721                 else if(lastRewardTimestamp <= rewardUpdateTimestamps[j])
722                     multiplier = block.timestamp - rewardUpdateTimestamps[j];
723                 // if the last reward timestamp was in the current segment
724                 // the time since last reward timestamp
```

```
                else
                    multiplier = block.timestamp - lastRewardTimestamp;

                // we are at the end
                rewards = rewards.add(multiplier.mul(rewardSegments[
                    rewardUpdateTimestamps[j]]));
                break;
            }

            // if the last reward timestamp was after this segment
            // it means we've already added this segment
            else if (rewardUpdateTimestamps[j] <= lastRewardTimestamp &&
                rewardUpdateTimestamps[j + 1] <= lastRewardTimestamp) continue;

            // if we haven't added this segment
            // add the full segment
            else if (rewardUpdateTimestamps[j + 1] <= block.timestamp)
                multiplier = rewardUpdateTimestamps[j + 1] - rewardUpdateTimestamps[j];

            rewards = rewards.add(multiplier.mul(rewardSegments[rewardUpdateTimestamps[j
                ]]));
        }
    }
```

Listing 3.3: `NeoPool::getRewardRate()`

Recommendation Revise the above-mentioned logic in the `getRewardRate()` routine to properly return the unfulfilled `wFTM` rewards.

Status The issue has been fixed by this commit: `ca96131`.

3.3 Inconsistent Fee Calculation Between Matrix And PancakePair

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Matrix`, `PancakePair`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [3]

Description

The `Matrix` contract is a fee distributer/converter contract. It takes the `DEX` fees in `LP tokens` and converts them to `wFTM`, then automatically distributes per set allocation points to various addresses. The `LP tokens` are firstly withdrawn from the related pools in the built-in `MorpheusFactory` to get the two underlying tokens, which will then be swapped to the `wFTM` token directly or via bridges. When examining the logic to swap tokens, we notice there is a logic error in current implementation.

```
243     function _swap(
244         address fromToken,
245         address toToken,
246         uint256 amountIn,
247         address to
248     ) internal returns (uint256 amountOut) {
249         // Checks
250         // X1 - X5: OK
251         IUniswapV2Pair pair = IUniswapV2Pair(
252             factory.getPair(fromToken, toToken)
253         );
254         require(address(pair) != address(0), "MatrixMaker: Cannot convert");
255 
256         // Interactions
257         // X1 - X5: OK
258         (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
259         uint256 amountInWithFee = amountIn.mul(997);
260         if (fromToken == pair.token0()) {
261             amountOut =
262                 amountInWithFee.mul(reserve1) /
263                 reserve0.mul(1000).add(amountInWithFee);
264             IERC20(fromToken).safeTransfer(address(pair), amountIn);
265             pair.swap(0, amountOut, to, new bytes(0));
266             // TODO: Add maximum slippage?
267         } else {
268             amountOut =
269                 amountInWithFee.mul(reserve0) /
270                 reserve1.mul(1000).add(amountInWithFee);
271             IERC20(fromToken).safeTransfer(address(pair), amountIn);
272             pair.swap(amountOut, 0, to, new bytes(0));
273             // TODO: Add maximum slippage?
274         }
275     }
```

Listing 3.4: `Matrix::_swap()`

To elaborate, we show above the related code snippet of the `_swap()` routine. In the above `_swap()` routine implementation, the reserved swap fee is 3‰ (line 259) of the `amountIn`. This is inconsistent with the fee rate (15‱ – lines 426 and 427) in the `PancakePair` contract where swap operations are performed. The code snippet of the `swap()` routine in the `PancakePair` contract is shown as below.

```
405     function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data)
            external lock {
406         require(amount0Out > 0  amount1Out > 0, 'Morpheus: INSUFFICIENT_OUTPUT_AMOUNT');
407         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
408         require(amount0Out < _reserve0 && amount1Out < _reserve1, 'Morpheus:
                INSUFFICIENT_LIQUIDITY');
409 
410         uint balance0;
```

```
        uint balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'Morpheus: INVALID_TO');
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically
            transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically
            transfer tokens
        if (data.length > 0) IPancakeCallee(to).pancakeCall(msg.sender, amount0Out,
            amount1Out, data);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 -
            amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 -
            amount1Out) : 0;
        require(amount0In > 0  amount1In > 0, 'Morpheus: INSUFFICIENT_INPUT_AMOUNT');
        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
        uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(15));
        uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(15));
        require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1)
            .mul(10000**2), 'Morpheus: K');
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }
```

Listing 3.5: `PancakePair::swap()`

The inconsistent fee calculation between the `Matrix` and the `PancakePair` will not block the token swap, but it will make the amount of the target token smaller than expectation. Hence, the `wFTM` token amount to be distributed will be smaller than what is expected.

Recommendation Be consistent on the fee calculation between `Matrix` and `PancakePair`.

Status The issue has been fixed by this commit: `ca96131`.

3.4 Inaccurate wFTM Reward Supply in updateRewardPerSec()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: NeoPool
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.2, the NeoPool contract provides an incentive mechanism that rewards the staking of PILLS with wFTM. Whenever the updateRewardPerSec() routine is called, it will calculate the wFTM per second to distribute based off the amount of wFTM in the contract, and a new reward segment (reward window) is created for pool users to claim their rewards per time. When analyzing the logic in the updateRewardPerSec() routine, we notice there is a logic error in current implementation.

To elaborate, we show below the full implementation of the updateRewardPerSec() routine. This routine calculates the wFTM to distribute from the amount of wFTM in the contract (line 766). However, it comes to our attention that the current wFTM amount in this contract may include certain amounts of wFTM that have already been distributed to the previous reward segment (because users may have not claimed their distributed rewards yet). In other words, the wFTM amount in this contract is not exactly the wFTM amount new distributed from Matrix since the last time the updateRewardPerSec() is called. As a result, more wFTM than expected may be distributed to the new reward segment.

```
760     // constant neo
761     function updateRewardPerSec() public {
762         require(msg.sender == oracle  msg.sender == owner(), "Only the oracle or Neo
            himself can get through...");
763 
764         // uint256 pillsSupply = pills.balanceOf(address(this));
765         // if (pillsSupply == 0) return;
766         uint256 rewardSupply = wftm.balanceOf(address(this));
767         if (rewardSupply == 0) return;
768 
769         // amount of seconds in a week + 1 day padding in case of network congestion
770         // don't want to run out of that good good
771         uint256 rewardPerSec = rewardSupply.div(period);
772 
773         // if it's not the first segment
774         // update this segment
775         // replace previous n.length - 1 index with current timestamp
776         if(
777             // not the first segment
778             rewardUpdateTimestamps.length != 0
779             // within bounds of current segment
```

```
            && block.timestamp < rewardUpdateTimestamps[rewardUpdateTimestamps.length - 1])
            rewardUpdateTimestamps[rewardUpdateTimestamps.length - 1] = block.timestamp;
        // this should never happen, but in case there is oracle lag/downtime
        // this prevents extra rewards, so there would be a segment with a 0 rewardPerSec value
        else
            rewardUpdateTimestamps.push(block.timestamp);

        rewardUpdateTimestamps.push(block.timestamp + period);
        rewardSegments[block.timestamp] = rewardPerSec;

        // in case rewardPerSec doesnt update in time
        rewardSegments[block.timestamp + period] = 0;

        updatePool();
    }
```

Listing 3.6: `NeoPool::updateRewardPerSec()`

Recommendation Correct the above `updateRewardPerSec()` routine by counting only the new distributed `wFTM` to the new reward `segment`.

Status The issue has been fixed by this commit: `ac37d02`.

3.5 Incorrect Amount Return of Target Token in Zapper

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Zapper`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Zapper` contract, we notice the `zapInToken()` function is used to exchange the zap-in token to the specified `LP Token` through the specified `router` on behalf of `msg.sender`. The zap-in token will be exchanged to the underlying two tokens in the pair specified by the input argument `_to` of the `zapInToken()` function. If the zap-in token could match with either of the two underlying tokens in the pair, it will exchange half amount of the zap-in token to the other underlying token of the pair. Then the two underlying tokens will be added to the pool to get the `LP Token`. Specially if the zap-in token can not match with either of the underlying tokens, it will firstly exchange the zap-in token to the native token, and then exchange the native token to the two underlying tokens in the pair.

To elaborate, we show below the implementation of the `_swapTokenForNative()` routine. As the name indicates, this routine is used to exchange the input token to the native token. While examining the return value of the routine when the input token supports `isFeeOnTransfer`, it comes to our attention that the routine wrongly returns the amount of the input token in the contract (line 1229). However, the target native token is transferred to the `recipient` directly in the call to the `router.swapExactTokensForETHSupportingFeeOnTransferTokens()` routine.

```
1212     function _swapTokenForNative(address token, uint amount, address recipient, address
             routerAddr) private returns (uint) {
1213         address[] memory path;
1214         IUniswapV2Router01 router = IUniswapV2Router01(routerAddr);
1215
1216         if (tokenBridgeForRouter[token][routerAddr] != address(0)) {
1217             path = new address[](3);
1218             path[0] = token;
1219             path[1] = tokenBridgeForRouter[token][routerAddr];
1220             path[2] = router.WETH();
1221         } else {
1222             path = new address[](2);
1223             path[0] = token;
1224             path[1] = router.WETH();
1225         }
1226
1227         if (isFeeOnTransfer[token]) {
1228             router.swapExactTokensForETHSupportingFeeOnTransferTokens(amount, 0, path,
                     recipient, block.timestamp);
1229             return IERC20(token).balanceOf(address(this));
1230         } else {
1231             uint[] memory amounts = router.swapExactTokensForETH(amount, 0, path,
                     recipient, block.timestamp);
1232             return amounts[amounts.length - 1];
1233         }
```

Listing 3.7: `Zapper::_swapTokenForNative()`

Moreover, we show below the code snippet of the `_swap()` routine. As the name indicates, it is used to exchange one type of token to another specified type of token. While examining the return value of the `_swap()` routine, it comes to our attention that, if the input token supports `isFeeOnTransfer`, the routine will wrongly return the amount of the target token in this contract (line 1248) (while the target token has been transferred to the `recipient` directly in the call to the `router.swapExactTokensForTokensSupportingFeeOnTransferTokens()` routine).

```
1236     function _swap(address _from, uint amount, address _to, address recipient, address
             routerAddr) private returns (uint) {
1237         IUniswapV2Router01 router = IUniswapV2Router01(routerAddr);
1238
1239         address fromBridge = tokenBridgeForRouter[_from][routerAddr];
1240         address toBridge = tokenBridgeForRouter[_to][routerAddr];
1241
```

```
        address[] memory path;
        ...
        uint[] memory amounts;

        if (isFeeOnTransfer[_from]) {
            router.swapExactTokensForTokensSupportingFeeOnTransferTokens(amount, 0, path
                , recipient, block.timestamp);
            return IERC20(_to).balanceOf(address(this));
        } else {
            amounts = router.swapExactTokensForTokens(amount, 0, path, recipient, block.
                timestamp);
        }

        return amounts[amounts.length - 1];
    }
```

Listing 3.8: `Zapper::_swap()`

Recommendation Revise the above-mentioned routines to return the correct amount of the target token.

Status The issue has been fixed by this commit: `1582446`.

3.6 Sybil Attacks on PILLS And MORPH Voting

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `MorpheusToken`, `PillsToken`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In `MorpheusSwap` , there is a `MorpheusToken (MORPH)` which has been enhanced with the functionality to cast and record the votes. Moreover, the `MORPH` contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the votes are counted prior to the proposal's activation.

Our analysis with the `MORPH` token shows that the current token contract is vulnerable to a so-called `Sybil` attacks [1]. For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 `MORPH` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `MORPH`. This `Sybil` attack can be launched as follows:

[1] The same issue occurs to the `SUSHI` token and the credit goes to Jong Seok Park[9].

```
function mint(uint256 amount) public onlyOwner returns (bool) {
    _mint(_msgSender(), amount);
    return true;
}
```

Listing 3.9: `BEP20::mint()`

```
    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        require(sender != address(0), "BEP20: transfer from the zero address");
        require(recipient != address(0), "BEP20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

Listing 3.10: `BEP20::_transfer()`

```
    function _delegate(address delegator, address delegatee)
    internal
{
    address currentDelegate = _delegates[delegator];
    uint256 delegatorBalance = balanceOf(delegator); // balance of underlying VICTs (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 3.11: `MorpheusToken.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 `MorpheusToken` 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 `MORPH`. When M_1 delegates to `Trudy`, since M_1 now has 100 `MORPH`, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `MORPH` 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!

To mitigate, it is necessary to accompany every single `transfer()`, `transferFrom()` and `mint()` with the `_moveDelegates()`, so that the voting power of the sender's delegate will be moved to the destination's delegate.

Similarly, the `PillsToken (PILLS)` which is the governance token in `MorpheusSwap` shares the same issue. The `PillsToken` contract's `transfer()` routine moves the voting power of the sender to the destination (line 884), not their delegates. To mitigate, it is also necessary to accompany every single `transfer()`, `transferFrom()` and `mint()` with the `_moveDelegates()` to move the voting power of the sender's delegate to the destination's delegate.

```
880     function transfer(address recipient, uint256 amount)
881     public virtual override returns (bool)
882     {
883         bool result = super.transfer(recipient, amount); // Call parent hook
884         _moveDelegates(_msgSender(), recipient, amount);
885 
886         return result;
887     }
```

Listing 3.12: `PillsToken::transfer()`

Recommendation Revise the above-mentioned logic to properly move the voting power in every single transfer(), transferFrom() and mint() with the _moveDelegates(). By doing so, we can effectively mitigate the above Sybil attacks.

Status This issue has been confirmed. The Morpheus team clarified that both the PILLS and MORPH will not be used in any voting.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In MorpheusSwap, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

Firstly, the privileged function in the NeoPool contract allows for the the owner to withdraw all the wFTM from the pool.

```
// Withdraw reward. EMERGENCY ONLY.
function emergencyRewardWithdraw(uint256 _amount) public onlyOwner {
    require(_amount < wftm.balanceOf(address(this)), 'not enough token');
    wftm.safeTransfer(address(msg.sender), _amount);
}
```

Listing 3.13: NeoPool::emergencyRewardWithdraw()

Secondly, the privileged functions in the MasterChef contract allows for the the devaddr to set a new devaddr which is used to receive MORPH rewards from the MasterChef. It also allows for the feeAddress to set a new feeAddress which is used to receive the deposit fee from the MasterChef. Our analysis shows that the devaddr and the feeAddress are currently configured as 0x92fcfc79187bc2db094c784d2a1b09e427ede24f which is a proxy to a multi-sig GnosisSafe account.

```
// Update dev address.
function setDevAddress(address _devaddr) external {
    require(msg.sender == devaddr, "dev: wut?");
    devaddr = _devaddr;
    emit SetDevAddress(msg.sender, _devaddr);
}

function setFeeAddress(address _feeAddress) external {
```

```
        require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
        require(_feeAddress != address(0), "!nonzero");
        feeAddress = _feeAddress;
        emit SetFeeAddress(msg.sender, _feeAddress);
    }
```

Listing 3.14: `MorpheusChef.sol`

Lastly, the privileged function in the `Matrix` contract allows for the the `owner` to set the `points` of the `recipient` which is used to share the distribution of `wFTM`.

```
    function setRecipient(address _address, uint8 _points) public onlyOwner {
        uint index = 0;

        // check t see if recipieint is already in list
        for(uint j = 0; j < addresses.length; j++) {
            if(addresses[j] != _address) continue;
            index = j + 1;
            break;
        }

        // create new
        if(index == 0) {
            addresses.push(_address);
            points.push(_points);

        // update existing
        } else {
            points[index - 1] = _points;
        }
    }
```

Listing 3.15: `Matrix::setRecipient()`

There are also some other privileged functions not listed above. And we understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `owner/feeAddress/devaddr`, etc. explicit to `MorpheusSwap` users.

Status This issue has been confirmed. The `Morpheus` team confirms that they will use multi-sig account.

3.8 Accommodation of Non-Compliant ERC20 Tokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Zapper
- Category: Coding Practices [5]
- 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. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. 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.

```
    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 3.16: USDT::transfer()

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer (), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we show the withdraw() routine in the Zapper contract. To accommodate the specific idiosyncrasy, there is a need to use safeTransfer(), instead of transfer() (line 1405).

```
1399    function withdraw(address token) external onlyOwner {
1400        if (token == address(0)) {
```

```
            payable(owner()).transfer(address(this).balance);
            return;
        }

        IERC20(token).transfer(owner(), IERC20(token).balanceOf(address(this)));
    }
```

Listing 3.17: `Zapper::withdraw()`

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been fixed by this commit: `3b189d2`.

4 Conclusion

In this audit, we have analyzed the `MorpheusSwap` design and implementation. The protocol is designed to focus on being able to provide the highest revenue share for a community `DEX`. During the audit, we notice that the current code base is well organized.

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

References

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

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

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

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

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

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

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

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

[9] Jong Seok Park. Sushiswap Delegation Double Spending Bug. https://medium.com/bulldax-finance/sushiswap-delegation-double-spending-bug-5adcc7b3830f.

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