SMART CONTRACT AUDIT REPORT

for

MantisSwap Protocol

Prepared By: Xiaomi Huang

PeckShield

November 22, 2022

Document Properties

Client	Mantissa Finance
Title	Smart Contract Audit Report
Target	MantisSwap Protocol
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	November 22, 2022	Luck Hu	Final Release
1.0-rc	November 20, 2022	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the MantisSwap 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About MantisSwap Protocol

MantisSwap is a next-generation decentralized AMM for stablecoins and pegged assets, which is to be launched on Polygon. With its new design and enhanced security mechanisms for liquidity providers and token holders, MantisSwap is targeted to be one of the most efficient protocol for trading pegged assets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of MantisSwap Protocol

Item	Description
Name	Mantissa Finance
Website	https://mantissa.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 22, 2022

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

- https://github.com/Mantissa-Finance/audit-v2.git (28dabc2d)

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

- https://github.com/Mantissa-Finance/audit-v2.git (37439ed9)

1.2 About PeckShield

PeckShield Inc. [7] 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 the OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy

Table 1.3: The Full Audit Checklist

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

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

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

2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	1	■
Informational	0	
Total	4	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key MantisSwap Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Proper Update of user.lastClaim in veMNT::deposit()	Business Logic	Fixed
PVE-002	Medium	Improved Validation of Unclaimed List in claimAuctionBid()	Business Logic	Fixed
PVE-003	Medium	Potential Reentrancy Risk in Rewarder::onMntReward()	Business Logic	Fixed
PVE-004	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 Proper Update of user.lastClaim in veMNT::deposit()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: veMNT
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the MantisSwap protocol, the veMNT contract provides the functionalities for users to deposit MNT and claim veMNT. The claimable amount of the veMNT is calculated per the deposit amount of MNT, the reward rate, and the time elapses since the last claim.

To elaborate, we show below the code snippet of the deposit() routine. As the name indicates, it is used for users to deposit MNT in the contact. Specially, for a new user's deposit, the reward rate (veMntRate) is set to veMntPerSec (line 100). So, if the user further deposits based on the current deposit, it will update the reward rate based on the new total deposit amount (line 104). This is because the current deposit is rewarded with the current veMntRate, while the new deposit will be rewarded with the veMntPerSec. However, it comes to our attention that there is a lack of updating the lastClaim variable based on the new veMntRate and the new total deposit amount. As a result, following a claim operation, the user may claim more veMNT than it is expected.

```
95      function deposit(uint256 amount) external checkCaller nonReentrant {
96          require(amount > 0, "Cannot be 0");
97          mntLp.safeTransferFrom(msg.sender, address(this), amount);
98          UserData memory user = userData[msg.sender];
99          if (user.amount == 0) {
100             user.veMntRate = veMntPerSec;
101             user.lastClaim = block.timestamp;
102             user.amount = amount;
103         } else {
104             uint256 newRate = _getNewRate(user, amount, veMntPerSec);
105             user.veMntRate = newRate;
```

```
            user.amount += amount;
        }
        userData[msg.sender] = user;
        emit Deposit(msg.sender, amount);
    }
```

Listing 3.1: veMNT::deposit()

Recommendation Revisit the `deposit()` routine to update the `lastClaim` state accordingly per the new `amount` and the new `veMntRate`.

Status The issue has been fixed in this commit: `99c074e5`.

3.2 Improved Validation of Unclaimed List in claimAuctionBid()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Marketplace`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `MantisSwap` protocol, the `Marketplace` contract provides a platform where users can list a part of their deposit of `MNT` and `veMNT` in the `veMNT` contact for sell. The owner can specify the price for the list or specify the start price if the list is an auction. In the case of an auction, the bid winner can claim the list by calling the `claimAuctionBid()` routine after the auction end time.

In the following, we show below the code snippet of the `claimAuctionBid()` routine. Specifically, it transfers the required tokens to the seller and the fee to the treasury, and move the `MNT`/`veMNT` in the list to the bidder. At last, it marks the `listings[seller][lid].sold = true` (line 230) which indicates the list is sold. However, while examining the validation of the list at the beginning of the `claimAuctionBid()` routine, we notice there is a lack of validation for the list state. As a result, a sold list can be claimed more than once.

```
216  function claimAuctionBid(address seller, uint256 lid) external whenNotPaused
         nonReentrant {
217    Listing memory listing = listings[seller][lid];
218    require(block.timestamp >= listing.endTime, "Auction not over");
219    Bid memory bid = bids[seller][lid];
220    address bidder = bid.bidder;
221    require(bidder != address(0), "No bids found");
222    address token = bid.token;
223    uint256 tokenAmount = (bid.amount * (10 ** IERC20(token).decimals())) / 1e6;
224    uint256 feeAmount = tokenAmount * exchangeFees / 1e4;
225    uint256 sellerAmount = tokenAmount - feeAmount;
```

```
    IERC20(token).safeTransfer(seller, sellerAmount);
    IERC20(token).safeTransfer(treasury, feeAmount);
    mntLp.approve(address(veMnt), listing.mntLpAmount);
    require(veMnt.exchangeVeMnt(seller, bidder, listing.mntLpAmount, listing.veMntAmount
        , listing.veMntRate), "Error");
    listings[seller][lid].sold = true;
    emit Bought(seller, lid, bidder, token, bid.amount);
}
```

Listing 3.2: `Marketplace::claimAuctionBid()`

Recommendation Properly validate the state of the list at the beginning of the `claimAuctionBid()` routine.

Status The issue has been fixed in this commit: `99c074e5`.

3.3 Potential Reentrancy Risk in Rewarder::onMntReward()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterMantis/Rewarder`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

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` [9] exploit, and the recent `Uniswap/Lendf.Me` hack [8].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. In the `Rewarder` contract, the `onMntReward()` function (see the code snippet below) is called from the `MasterMantis` to reward the staker with third-party's native token alongside `MNT` by externally calling a token contract to transfer rewards to the staker. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (lines 109, 111) starts before effecting the update on internal states (lines 115-116), hence violating the principle.

```
99  function onMntReward(address _user, uint256 _lpAmount) external onlyMasterMantis {
```

```
    updatePool();
    PoolInfo memory pool = poolInfo;
    UserInfo storage user = userInfo[_user];
    uint256 pending;
    // if user had deposited
    if (user.amount > 0) {
        pending = (user.amount * pool.accTokenPerShare / ACC_TOKEN_PRECISION) - user.
            rewardDebt;
        uint256 balance = rewardToken.balanceOf(address(this));
        if (pending > balance) {
            rewardToken.safeTransfer(_user, balance);
        } else {
            rewardToken.safeTransfer(_user, pending);
        }
    }

    user.amount = _lpAmount;
    user.rewardDebt = user.amount * pool.accTokenPerShare / ACC_TOKEN_PRECISION;

    emit OnReward(_user, pending);
}
```

Listing 3.3: Rewarder::onMntReward()

Because the `onMntReward()` function can only be called from the `MasterMantis`, which seems the issue is mitigated. However, our study shows that the `MasterMantis` itself is reentrant. In the following, we show the code snippet of the `MasterMantis::withdrawFor()` routine. It comes to our attention that this routine is not properly protected by the `nonReentrant` as in `deposit()/withdraw()`. As a result, the `MasterMantis` can be reentrant by calling any of the `deposit()/withdraw()` routines following the `withdrawFor()` routine.

```
  function withdrawFor(address recipient, uint256 _pid, uint256 _amount) external
      override onlyPoolContracts {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][recipient];
    ...
    if (address(rewarders[_pid]) != address(0)) {
        rewarders[_pid].onMntReward(recipient, user.amount);
    }
    emit Withdraw(recipient, _pid, _amount);
  }
```

Listing 3.4: MasterMantis::withdrawFor()

Specifically, in the case when `rewardToken` is an ERC777 token, a bad actor could hijack a `MasterMantis::withdrawFor()` call before `rewardToken.safeTransfer()` in the `onMntReward()` routine with a callback function. Within the callback function, the bad actor could call the `MasterMantis::withdraw()` function which further calls the `onMntReward()`. Since the `user.amount/user.rewardDebt` are not updated yet, the bad actor can claim more rewards than it is expected.

Recommendation Apply the `checks-effects-interactions` design pattern in the `Rewarder::onMntReward()` routine or add the reentrancy guard modifier in the `MasterMantis::withdrawFor()` routine.

Status The issue has been fixed in this commit: `99c074e5`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `MantisSwap` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., set `MNT` reward rate). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `MasterMantis` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `MasterMantis` allow for the `owner` to set the gauge update interval, set the gauge asset weight and vote weight, set the `MNT` reward rate for stakers, set the `veMnt` which impacts users reward factors and voting powers, etc.

```
    function setGaugeUpdateInterval(uint256 _gaugeUpdateInterval) external onlyOwner {
        require(_gaugeUpdateInterval > 0, "Cannot be 0");
        gaugeUpdateInterval = _gaugeUpdateInterval;
        emit GaugeIntervalUpdated(_gaugeUpdateInterval);
    }

    function setGaugeWeights(uint256 _gaugeAssetWeight, uint256 _gaugeVoteWeight)
        external onlyOwner {
        require(_gaugeAssetWeight + _gaugeVoteWeight == 100, "Incorrect sum");
        gaugeAssetWeight = _gaugeAssetWeight;
        gaugeVoteWeight = _gaugeVoteWeight;
        emit GaugeWeightUpdated(_gaugeAssetWeight, _gaugeVoteWeight);
    }

    function setMntPerBlock(uint256 _mntPerBlock) external onlyOwner {
        massUpdatePools();
        mntPerBlock = _mntPerBlock;
        emit MntPerBlockUpdated(_mntPerBlock);
    }

```

```
    function setVeMnt(address _veMnt) external onlyOwner {
        require(_veMnt != address(0), "Cannot be zero address");
        massUpdatePools();
        veMnt = _veMnt;
        emit veMntUpdated(_veMnt);
    }

    function setPoolContract(address _poolContract, bool _status) external onlyOwner {
        require(_poolContract != address(0), "Cannot be zero address");
        poolContracts[_poolContract] = _status;
        emit PoolContractSet(_poolContract, _status);
    }

    function setRewarder(uint256 _pid, address _rewarder) external onlyOwner {
        rewarders[_pid] = IRewarder(_rewarder);
        emit RewarderSet(_pid, _rewarder);
    }
```

Listing 3.5: Example Privileged Operations in the `MasterMantis` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirmed they use multi-sig for the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `MantisSwap` protocol. `MantisSwap` is a next-generation decentralized `AMM` for stablecoins and pegged assets, which is to be launched on `Polygon`. With its new design and enhanced security mechanisms for liquidity providers and token holders, `MantisSwap` is targeted to be one of the most efficient protocol for trading pegged assets. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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

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

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