SMART CONTRACT AUDIT REPORT

for

MagpieV2 Protocol

Prepared By: Xiaomi Huang

PeckShield

December 19, 2022

Document Properties

Client	Magpie
Title	Smart Contract Audit Report
Target	MagpieV2 Protocol
Version	1.1
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
1.1	December 19, 2022	Luck Hu	Post Release #1
1.0	December 1, 2022	Luck Hu	Final Release
1.0-rc1	November 28, 2022	Luck Hu	Release Candidate #1

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 MagpieV2 protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About MagpieV2 Protocol

Magpie is an innovative yield-boosting protocol that provides users with boosted yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. The new Magpie protocol, i.e., MagpieV2, enables the vlMGP holders to use veWom accumulated on Magpie to vote the Wom emission on Wombat and receive bribe rewards. Magpie implements the Magpie token (MGP) for the protocol management, which is deployed at address 0xD06716E1Ff2E492Cc5034c2E81805562dd3b45fa. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The MagpieV2 Protocol

Item	Description
Issuer	Magpie
Website	https://magpiexyz.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 19, 2022

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

- https://github.com/magpiexyz/magpie_contracts.git (7825bdb, aa7af6b)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/magpiexyz/magpie_contracts.git (31a314c, 6b381ac);

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	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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.

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 `MagpieV2` protocol implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■
Low	3	■■
Informational	0	
Total	6	

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 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key MagpieV2 Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inconsistent Logics to Calculate callerFeeAmount	Business Logic	Fixed
PVE-002	Low	Revisited Logic in SmartWomConvert::_convertFor()	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Medium	Revisited Logic to Accumulate Rewards for vlMGP	Business Logic	
PVE-006	Low	Proper Reset of userRewards in _sendReward()	Business Logic	

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 Inconsistent Logics to Calculate callerFeeAmount

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

Description

The WombatStaking contract interacts with the Wombat Exchange to provide functionalities such as adding new liquidity, staking LP tokens on MasterWombat, and staking WOM to get veWom. With the accumulated veWom, the MagpieV2 protocol can vote the Wom emission on Wombat and receive bribe rewards. The MagpieV2 protocol allows for the vlMGP holders to vote on how the veWom voting powers are distributed to each Wombat LP. To incentivize the caller to cast the pending votes to Wombat, it rewards the caller with a caller fee from the Wombat bribe rewards.

To elaborate, we show below the code snippet of the WombatStaking::Vote() routine. As the name indicates, it is used to vote the Wom emission on Wombat and receive bribe rewards. For each received bribe reward, it calculates protocol fee first (line 386), decreases the protocol fee from the reward amount (line 390), and then calculates the caller fee based on the new reward amount (line 393).

```
359     function vote(
360         address[] calldata _lpVote,
361         int256[] calldata _deltas,
362         address[] calldata _rewarders,
363         address caller
364     ) external returns (IERC20[][] memory rewardTokens, uint256[][] memory
            callerFeeAmounts) {
365         if(msg.sender != bribeManager)
366             revert OnlyBribeMamager();
367
368         if (_lpVote.length != _rewarders.length)
369             revert LengthMismatch();
```

```
        uint256[][] memory rewardAmounts = voter.vote(_lpVote, _deltas);
        rewardTokens = new IERC20[][](rewardAmounts.length);
        callerFeeAmounts = new uint256[][](rewardAmounts.length);

        for (uint256 i; i < rewardAmounts.length; i++) {
            address bribesContract = address(voter.infos(_lpVote[i]).bribe);

            if (bribesContract != address(0)) {
                rewardTokens[i] = IWombatBribe(bribesContract).rewardTokens();
                callerFeeAmounts[i] = new uint256[](rewardAmounts[i].length);

                for (uint256 j; j < rewardAmounts[i].length; j++) {
                    uint256 rewardAmount = rewardAmounts[i][j];
                    uint256 callerFeeAmount = 0;

                    if (rewardAmount > 0) {
                         uint256 protocolFee = (rewardAmount * bribeProtocolFee) / DENOMINATOR;

                        if (protocolFee > 0) {
                            IERC20(rewardTokens[i][j]).safeTransfer(bribeFeeCollector, protocolFee);
                            rewardAmount -= protocolFee;
                        }
                        if (caller != address(0) && bribeCallerFee != 0) {
                            callerFeeAmount = (rewardAmount * bribeCallerFee) / DENOMINATOR;
                            IERC20(rewardTokens[i][j]).safeTransfer(bribeManager, callerFeeAmount);
                            rewardAmount -= callerFeeAmount;
                        }

                        IERC20(rewardTokens[i][j]).safeApprove(_rewarders[i], rewardAmount);
                        IBaseRewardPool(_rewarders[i]).queueNewRewards(rewardAmount, address(rewardTokens[i][j]));
                    }
                    callerFeeAmounts[i][j] = callerFeeAmount;
                }
            }
        }
    }
```

Listing 3.1: `WombatStaking::vote()`

Moreover, the `WombatStaking` contract provides the `pendingBribeCallerFee()` routine to facilitate the calculation of the caller fee for the pending bribe rewards. However, it comes to our attention that the caller fee is calculated directly based on the original bribe rewards amount (line 226) retrieved from `Wombat` and it does not take the protocol fee into consideration as in the `Vote()` routine. As a result, the caller fee amount calculation here is inconsistent with the calculation in the `Vote()` routine.

```
    function pendingBribeCallerFee(address[] calldata pendingPools)
    external
    view
    returns (IERC20[][] memory rewardTokens, uint256[][] memory callerFeeAmount)
{
    // Warning: Arguments do not take into account repeated elements in the pendingPools list
    uint256[][] memory pending = voter.pendingBribes(pendingPools, address(this));

    rewardTokens = new IERC20[][](pending.length);
    callerFeeAmount = new uint256[][](pending.length);

    for (uint256 i; i < pending.length; i++) {
        rewardTokens[i] = IWombatBribe(voter.infos(pendingPools[i]).bribe).rewardTokens();
        callerFeeAmount[i] = new uint256[](pending[i].length);

        for (uint256 j; j < pending[i].length; j++) {
            if (pending[i][j] > 0) {
                callerFeeAmount[i][j] = (pending[i][j] * bribeCallerFee) / DENOMINATOR;
            }
        }
    }
}
```

Listing 3.2: WombatStaking::pendingBribeCallerFee()

Recommendation Revisit the above mentioned Vote()/pendingBribeCallerFee() routines to use the same algorithm for the caller fee calculation.

Status The issue has been fixed by this commit: de3168a.

3.2 Revisited Logic in SmartWomConvert::_convertFor()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: SmartWomConvert
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the MagpieV2 protocol, the SmartWomConvert contract is a smart converter for users to convert their Wom to mWom. The Wom token can be converted to mWom in two ways. The first way is to swap Wom for mWom in Wombat, and the second way is achieved by depositing Wom into the mWom contract to mint mWom.

To elaborate, we show below the code snippet of the `_convertFor()` routine. The `_convertRatio` argument is used to indicate the ratio of the input `Wom` amount that will be deposited to the `mWom` contract to mint `mWom`. The rest of the `Wom` amount (`buybackAmount`) is used to swap `Wom` for `mWom` in `Wombat`. Normally, if `buybackAmount > 0`, we can expect to receive a normal amount out from `Wombat`. In particular, if `buybackAmount == 0`, all the input `Wom` will be deposited to the `mWom` contract to mint `mWom`. However, it still invokes the `IWombatRouter(router).swapExactTokensForTokens()` routine trying to buy `mWom` with 0 amount of `Wom`. As a result, the transaction will be reverted in `Wombat`, because the `Wombat` does not accept 0 amount of the `from` token for the swap. Therefor it is suggested to invoke the `IWombatRouter(router).swapExactTokensForTokens()` (line 122) only when `buybackAmount > 0`.

```
103    function _convertFor(uint256 _amount, uint256 _convertRatio, uint256 _minRec,
           address _for, bool _stake)
104        internal
105        returns (uint256 obtainedmWomAmount)
106    {
107        if (_convertRatio > DENOMINATOR)
108            revert IncorrectRatio();
109
110        IERC20(wom).safeTransferFrom(msg.sender, address(this), _amount);
111        uint256 buybackAmount = _amount - (_amount * _convertRatio / DENOMINATOR);
112        uint256 convertAmount = _amount - buybackAmount;
113
114        address[] memory tokenPath = new address[](2);
115        tokenPath[0] = wom;
116        tokenPath[1] = mWom;
117
118        address[] memory poolPath = new address[](1);
119        poolPath[0] = womMWomPool;
120
121        IERC20(wom).safeApprove(router, buybackAmount);
122        uint256 amountRec = IWombatRouter(router).swapExactTokensForTokens(
123            tokenPath, poolPath, buybackAmount, 0, address(this), block.timestamp
124        );
125
126        IERC20(wom).safeApprove(mWom, convertAmount);
127        IMWom(mWom).deposit(convertAmount);
128        ...
129    }
```

Listing 3.3: SmartWomConvert::_convertFor()

The same issue is applicable to the `estimateTotalConversion()` routine as well.

Recommendation Revisit the above `_convertFor()` function to trigger the swap in `Wombat` only when `buybackAmount > 0`.

Status The issue has been fixed by this commit: `de3168a`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

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

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

Listing 3.4: ZRX.sol

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

In the following, we show the withdraw() routine in the BNBZapper contract. If the ZRX token is supported as token, the unsafe version of IERC20(token).transfer() (line 125) may return false while not revert. Without a validation on the return value, the transaction can proceed even when the transfer fails. The same issue is applicable to the BNBZapper::zapInToken() routine, where the call to transferFrom() is suggested to use the safe version, i.e., safeTransferFrom().

```
119     function withdraw(address token) external onlyOwner {
120         if (token == address(0)) {
121             payable(owner()).transfer(address(this).balance);
122             return;
123         }
124 
125         IERC20(token).transfer(owner(), IERC20(token).balanceOf(address(this)));
126     }
```

Listing 3.5: BNBZapper::withdraw()

What's more, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the approve() function requires to reduce the allowance to 0 first if it is not, and then set the proper allowance. This requirement is in place to mitigate the known approve()/transferFrom() race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

Because of that, a normal call to approve() with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to approve() twice: the first one reduces the allowance to 0, and the second one sets the new allowance. Moreover, a normal call to approve() is suggested to use the safe version, i.e., safeApprove(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. And 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. The issue is applicable to the WombatBribeManager::_approveTokenIfNeeded() routine, etc.

Recommendation Accommodate the above-mentioned idiosyncrasies with safe-version implementation of ERC20-related approve()/transfer()/transferFrom(). And there is a need to approve() twice: the first one reduces the allowance to 0, and the second one sets the new allowance.

Status The issue has been fixed by this commit: beeba73.

3.4 Trust Issue of Admin Keys

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

Description

In the `Magpie` 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 protocol fee for the bribe rewards). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `WombatStaking` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `WombatStaking` allow for the `owner` to set the bribe manager who can distribute the `veWom` voting powers among the `Wombat LP` tokens, set the bribe protocol fee, set the caller fee, set the protocol fee receiver, etc.

```
    function setBribeManager(address _bribeManager) external onlyOwner {
        address oldBribeManager = bribeManager;
        bribeManager = _bribeManager;

        emit BribeManagerUpdated(oldBribeManager, bribeManager);
    }

    function setBribe(
        address _voter,
        address _bribeManager,
        uint256 _bribeCallerFee,
        uint256 _bribeProtocolFee,
        address _bribeFeeCollector
    ) external onlyOwner {
        voter = IWombatVoter(_voter);
        bribeManager = _bribeManager;
        bribeCallerFee = _bribeCallerFee;
        bribeProtocolFee = _bribeProtocolFee;
        bribeFeeCollector = _bribeFeeCollector;

        emit BribeSet(_voter, _bribeManager, _bribeCallerFee, _bribeProtocolFee,
            _bribeFeeCollector);
    }
```

Listing 3.6: Example Privileged Operations in the `WombatStaking` 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 confirms they plan to use multi-sig for the owner account.

3.5 Revisited Logic to Accumulate Rewards for vlMGP

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MasterMagpie
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the MagpieV2 protocol, the MasterMagpie contract is a customized implementation of MasterChef, which incentivizes user deposits of the supported assets with MGP. In particular, one of the supported assets is vlMGP which is minted to users for their lock of MGP tokens in the VLMGP contract. While examining the MGP rewards calculation for the deposit of vlMGP, we notice the vlMGP in cool down state is not taken into account to share the rewards.

To elaborate, we show below the code snippet of the _calLpSupply() routine which is used to calculate the total supply for the given pool. Normally the total supply, i.e., lpSupply, is the amount of the staking token that is locked in the contract (line 676). Specially, for vlMGP, the total supply is retrieved from the VLMGP::totalLocked() routine (line 674). In the VLMGP::totalLocked() routine, it returns the total amount of vlMGP that is not in cool down state (line 109). However, it is designed that the cool-down vlMGP can also receive rewards, and only the fully unlocked vlMGP can not receive rewards.

```
672     function _calLpSupply(address _stakingToken) internal view returns (uint256) {
673         if (_stakingToken == address(vlmgp))
674             return IVLMGP(vlmgp).totalLocked();
675 
676         return IERC20(_stakingToken).balanceOf(address(this));
```

```
    }
```

Listing 3.7: MasterMagpie::_calLpSupply()

```
    function totalLocked() override public view returns (uint256) {
        return this.totalSupply() - this.totalAmountInCoolDown();
    }
```

Listing 3.8: VLMGP::totalLocked()

Recommendation Revisit the `_calLpSupply()` routine and take the `vlMGP` that is in cool down state into the total supply to share the rewards.

Status The issue has been fixed by this commit: `89a46ec`.

3.6 Proper Reset of userRewards in _sendReward()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `vlMGPBaseRewarder`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `MagpieV2` protocol, the `vlMGPBaseRewarder` contract is a special reward pool for the staking of `vlMGP` in `MasterMagpie`. When a `vlMGP` depositor claims rewards, the `_sendReward()` routine is invoked to take the potential forfeit and send the rewards to the receiver. While examining the logic to distribute the rewards, we notice it doesn't reset the `userRewards[_rewardToken][_account]` when all the user rewards are taken as forfeit.

To elaborate, we show below the code snippet of the `_sendReward()` routine. Firstly, it calculates the forfeit that shall be taken for the fully unlocked `vlMGP` (line 376). After the forfeit is taken, the remaining rewards are to be sent to the user (line 377). Normally, when the rewards amount to user is positive, the `userRewards[_rewardToken][_account]` is reset. However, when the rewards amount to user is 0, the `userRewards[_rewardToken][_account]` is not reset. As a result, the bad actor can claim rewards for his fully unlocked `vlMGP` to queue the forfeit again and again to the reward pool.

```
375     function _sendReward(address _rewardToken, address _account, address _receiver)
            internal {
376         uint256 forfeit = _calExpireForfeit(_account, userRewards[_rewardToken][_account
                ]);
377         uint256 toSend = userRewards[_rewardToken][_account] - forfeit;
378
```

```
        if (toSend > 0) {
            userRewards[_rewardToken][_account] = 0;
            IERC20(_rewardToken).safeTransfer(_receiver, toSend);
            emit RewardPaid(_account, _receiver, toSend, _rewardToken);
        }

        if(forfeit > 0)
            _queueNewRewardsWithoutTransfer(forfeit, _rewardToken);
    }
```

Listing 3.9: vlMGPBaseRewarder::_sendReward()

Recommendation Reset the `userRewards[_rewardToken][_account]` anyway in the `_sendReward()` routine.

Status The issue has been fixed by this commit: `89a46ec`.

4 Conclusion

In this audit, we have analyzed the MagpieV2 design and implementation. Magpie is an innovative yield-boosting protocol that provides users with boosted yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. The new Magpie protocol, i.e., MagpieV2, enables the vlMGP holders to use the veWom accumulated on Magpie to vote the Wom emission on Wombat and receive bribe rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-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.