SMART CONTRACT AUDIT REPORT

for

Magpie Protocol

Prepared By: Xiaomi Huang

PeckShield

October 28, 2022

Document Properties

Client	Magpie
Title	Smart Contract Audit Report
Target	Magpie 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	October 28, 2022	Luck Hu	Post Release #1
1.0	August 16, 2022	Luck Hu	Final Release
1.0-rc1	August 8, 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 Magpie 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 Magpie Protocol

Magpie is an innovative yield-boosting protocol that provides users with boosted stablecoin yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. Conversely, WOM holders can also benefit from Magpie by converting their WOM token into mWOM (Magpie WOM) to earn a share of Magpie's profit. 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 Magpie Protocol

Item	Description
Issuer	Magpie
Website	https://magpiexyz.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 28, 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 (4ada7e9, e59e4b0)

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 (23ba731, 01b6206);

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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) [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 `Magpie` 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	2	■■
Medium	3	■■■
Low	2	■■
Informational	0	
Total	7	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Magpie Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic To Distribute Caller Fee	Business Logic	Fixed
PVE-002	High	Incorrect Token Flow in withdraw()	Business Logic	Fixed
PVE-003	Low	Improved Sanity Checks For Function Parameters	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	High	Revisited Logic in cancelUnlock()	Business Logic	Fixed
PVE-006	Medium	Improved Quote Between depositToken and LP	Business Logic	Fixed
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	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 Revisited Logic To Calculate The Rewards

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: WombatStaking
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The WombatStaking contract interacts with the Wombat Exchange to provide functionalities, such as adding new liquidity, staking LP token on MasterWombat, and staking WOM to get veWom. It provides an external harvest() interface for users or WombatPoolHelper to claim rewards from MasterWombat. Our analysis shows the current logic in reward calculation needs to be improved.

To elaborate, we show below the code snippet of the harvest() routine. By design, it is implemented to claim rewards from the specified pool in MasterWombat and send the rewards to the rewarder. Specially, if the caller is not the Magpie contracts, the caller can get the caller fee in mWom. The original fee in WOM is locked to get veWom via the convertWOM() routine (line 347). After that, the routine transfers mWom to the caller (line 348). However, the contract may not hold mWom at all, hence simply reverting the harvest operation. Our analysis shows that the WOM rewards will be converted to mWom by invoking `Uint fee = IMWom(mWom).convert(feeAmount)` and the received mWom amount (fee) may be not equal to the feeAmount as the veWom may not be converted from WOM with the $1 : 1$ conversion rate. As a result, it needs to transfer the fee amount of mWom to the caller: `IERC20(mWom).safeTransfer(msg.sender, fee)`.

```
329     /// @notice harvest a Pool from MGP
330     /// @param _depositToken the address of the deposit token Pool to harvest
331     /// @param _isUser true if this function is not called by the magpie Contracts. The
            caller gets the caller fee
332     function harvest(
333         address _depositToken,
```

```
        bool _isUser
    ) _onlyActivePool(_depositToken) external {
        Pool storage poolInfo = pools[_depositToken];
        uint256 beforeBalance = IERC20(wom).balanceOf(address(this));
        uint256[] memory pids = new uint256[](1);
        pids[0] = poolInfo.pid;
        IMasterWombat(masterWombat).multiClaim(pids); //only claim from a specific
            deposit token Pool

        uint256 rewards = IERC20(wom).balanceOf(address(this)) - beforeBalance;
        uint256 afterFee = rewards;
        if (_isUser) {
            uint256 feeAmount = (rewards * CALLER_FEE) / FEE_DENOMINATOR;
            IERC20(wom).approve(mWom, feeAmount);
            this.convertWOM(feeAmount);
            IERC20(mWom).safeTransfer(msg.sender, feeAmount);
            afterFee = afterFee - feeAmount;
        }
        sendRewards(poolInfo.depositToken, poolInfo.rewarder, rewards, afterFee);

        emit WomHarvested(rewards, rewards - afterFee);
    }

    /// @notice convert WOM to mWOM
    /// @param _amount the number of WOM to convert
    /// @dev the WOM must already be in the contract
    function convertWOM(uint256 _amount) external returns(uint256) {
        uint256 veWomMintedAmount = 0;
        if (_amount > 0) {
            IERC20(wom).approve(veWom, _amount);
            veWomMintedAmount = IVeWom(veWom).mint(_amount, IVeWom(veWom).maxLockDays())
                ;
        }

        emit WomConverted(_amount, IVeWom(veWom).maxLockDays());

        return veWomMintedAmount;
    }
```

Listing 3.1: `WombatStaking::harvest()`

Recommendation Revisit the above logic to properly convert the `WOM` to `mWom` and transfer the received amount of `mWom` to the caller.

Status The issue has been fixed by this commit: `4c0677d`.

3.2 Incorrect Token Flow in withdraw()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: `WombatPoolHelper/WombatStaking`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `WombatPoolHelper` is a helper smart contract to interact with `WombatStaking` (to provide liquidity on `Wombat Exchange`, etc.) and `MasterMagpie` (to stake `LP` token as a certificate and farm). While examining the token flow to withdraw assets from `WombatPoolHelper`, we notice the existence of incorrect token flow between `WombatPoolHelper` and `WombatStaking` which needs to be corrected.

To elaborate, we show below the code snippets of the `WombatPoolHelper:withdraw()` and `WombatStaking:withdraw()` routines. Specifically, the `WombatPoolHelper:withdraw()` takes the input `_amount` of `receiptToken` and unstakes it from `MasterMagpie` (line 137). The unstaked `receiptToken _amount` is then used in the `WombatStaking:withdraw()` function (line 140). However, within the `WombatStaking:withdraw()` function, the input `_amount` is expected to be the amount of stables (`_depositToken`) which will be provided to `Wombat Exchange` as liquidity. Our analysis shows that, the `WombatPoolHelper:withdraw()` could be refactored to take the input `_amount` as the amount of the stables to withdraw which is the same as the `WombatPoolHelper:deposit()` (where the input `_amount` represents the amount of the stables to deposit). With that, the `WombatPoolHelper:withdraw()` function shall convert the input `_amount` of the stables to the amount of `receiptToken` which is needed to unstake from `MasterMagpie`.

```
136    function withdraw(uint256 _amount, uint256 _minAmount) external override _harvest {
137        _unstake(_amount, msg.sender);
138        IWombatStaking(wombatStaking).withdraw(
139            depositToken,
140            _amount,
141            _minAmount,
142            msg.sender
143        );
144        emit NewWithdraw(msg.sender, _amount);
145    }
```

Listing 3.2: WombatPoolHelper:withdraw()

```
function withdraw(
    address _depositToken,
    uint256 _amount,
    uint256 _minAmount,
    address _sender
) _onlyPoolHelper(_depositToken) external {
```

```
        // _amount is the amount of stable
        Pool storage poolInfo = pools[_depositToken];
        uint256 sharesAmount = getSharesForDepositTokens(_amount, _depositToken);
        uint256 lpAmount = getLPTokensForShares(sharesAmount, _depositToken);
        IMintableERC20(poolInfo.receiptToken).burn(msg.sender, sharesAmount);

        IERC20(poolInfo.lpAddress).approve(poolInfo.depositTarget, lpAmount);
        IMasterWombat(masterWombat).withdraw(poolInfo.pid, lpAmount);

        uint256 beforeWithdraw = IERC20(_depositToken).balanceOf(address(this));
        IWombatPool(poolInfo.depositTarget).withdraw(
            _depositToken,
            lpAmount,
            _minAmount,
            address(this),
            block.timestamp
        );

        poolInfo.size -= _amount;
        poolInfo.sizeLp -= lpAmount;

        IERC20(_depositToken).safeTransfer(
            _sender,
            IERC20(_depositToken).balanceOf(address(this)) - beforeWithdraw
        );

        emit NewWithdraw(_sender, _depositToken, _amount);
    }
```

Listing 3.3: WombatStaking:withdraw()

Recommendation Revised the `WombatPoolHelper:withdraw()` function to correct the token flow during assets withdrawal.

Status The issue has been fixed by this commit: `0b024dd`.

3.3 Improved Sanity Checks For Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterMagpie
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The MasterMagpie contract is a MasterChef implementation in Magpie, which provides an incentive mechanism that rewards the staking of supported assets with MGP. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And stake holders are rewarded in proportional to their share of deposited token 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 update routine set(), we notice the need of properly validating the input pool before the new pool weight becomes effective.

To elaborate, we show below the code snippet of the set() routine. As the name indicates, it is used by the pool manager to update the weight, rewarder and locker for the pool given by the _stakingToken. At the beginning of the routine, it validates the input _rewarder and _locker. However, it does not validate the input pool. As a result, if the pool is an invalid one, this routine could successfully update the totalAllocPoint which will create an unfair reward distribution to stake holders.

```
    function set(
        address _stakingToken,
        uint256 _allocPoint,
        address _rewarder,
        address _locker,
        bool _overwrite
    ) external _onlyPoolManager {
        if (!Address.isContract(address(_rewarder)) && address(_rewarder) != address(0))
            revert MustBeContractOrZero();

        if (!Address.isContract(address(_locker)) && address(_locker) != address(0))
            revert MustBeContractOrZero();

        massUpdatePools();
        totalAllocPoint =
            totalAllocPoint -
            tokenToPoolInfo[_stakingToken].allocPoint +
            _allocPoint;

        tokenToPoolInfo[_stakingToken].allocPoint = _allocPoint;

```

```
        if (_overwrite) {
            tokenToPoolInfo[_stakingToken].rewarder = _rewarder;
            tokenToPoolInfo[_stakingToken].locker = _locker;
        }

        emit Set(
            _stakingToken,
            _allocPoint,
            IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder),
            tokenToPoolInfo[_stakingToken].locker,
            _overwrite
        );
    }
```

Listing 3.4: `MasterMagpie::set()`

Note it shares the same issue in the `updatePool()` routine which shall update only valid pools.

Recommendation Revisit the above mentioned routines to add proper sanity checks.

Status The issue has been fixed by this commit: `ce1715c`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

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 pool managers). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `MasterMagpie` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `MasterMagpie` allow for the `owner` to set pool manager who can add/set the reward pools, toggle whether emergency withdraw is allowed, update the emission rate, etc.

```
    function setPoolManagerStatus(address _account, bool _allowedManager)
        external
        onlyOwner
    {
        PoolManagers[_account] = _allowedManager;
```

```
        emit PoolManagerStatus(_account, PoolManagers[_account]);
    }

    /// @notice update the status of emergencyWithdraw.
    /// @notice WARNING : the contract should not be used after that action for anything
        other that withdrawing
    function allowEmergency(bool _allow) external onlyOwner {
        emergencyAllowed = _allow;

        emit EmergencyUpdated(emergencyAllowed);
    }

    function updateEmissionRate(uint256 _mgpPerSec) public onlyOwner {
        massUpdatePools();
        uint256 oldEmissionRate = mgpPerSec;
        mgpPerSec = _mgpPerSec;

        emit UpdateEmissionRate(msg.sender, oldEmissionRate, mgpPerSec);
    }
```

Listing 3.5: Example Privileged Operations in the `MasterMagpie` 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 all admin roles.

3.5 Revisited Logic in cancelUnlock()

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

Description

The new Magpie protocol introduces a VLMGP contract as a locker for MGP. If a user locks MGP into this contract, the contract mints the same amount of vlMGP and deposits the vlMGP into MasterMagpie so that the user can get rewards. The user can unlock MGP any time, but MGP will be kept in the VLMGP contract for a cool-down period before they can be transferred to the user. In the cool-down period, the user has the chance to cancel the unlock, which re-deposits the MGP back into MasterMagpie.

To elaborate, we show below the code snippet of the cancelUnlock() routine. As the name indicates, it is used for users to cancel the unlock of the MGP in cool down. Specifically, it invokes the _lock() routine (line 280) to deposit the MGP back into MasterMagpie. However, it comes to our attention that the _lock() function will mint the same amount of vlMGP again for the users. As a result, users vlMGP balances are doubled. As a result, users can repeat startUnlock() and unlock() to gain as many vlMGP as they want and drain all the locked MGP from the contract.

```
270     function cancelUnlock(uint256 _slotIndex) external override whenNotPaused {
271         _checkIdexInBoundary(msg.sender, _slotIndex);
272         UserUnlocking storage slot = userUnlockings[msg.sender][_slotIndex];
273 
274         if(slot.endTime <= block.timestamp)
275             revert NotInCoolDown();
276 
277         if (slot.amountInCoolDown == 0)
278             revert UnlockedAlready();
279 
280         _lock(msg.sender, msg.sender, slot.amountInCoolDown, false);
281 
282         slot.amountInCoolDown = 0; // not in cool down anymore
283         emit ReLock(msg.sender, _slotIndex, slot.amountInCoolDown);
284     }
```

Listing 3.6: VLMGP::cancelUnlock()

Recommendation Revisit the logic of the startUnlock()/unlock() routines to avoid the double mint of vlMGP.

Status The issue has been fixed by this commit: 748be39.

3.6 Improved Quote Between depositToken and LP

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WombatStaking`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the `WombatStaking` contract interacts with `Wombat Exchange` to facilitate users to deposit assets in `Wombat` and stake the `LP` in `MasterWombat` for yield. Moreover, the `WombatStaking` contract allows users to deposit `Wombat LP` into `MasterWombat` for yield. To incentivize users deposit in `WombatStaking`, it mints `receiptToken` to users who can further deposit the `receiptToken` into `MasterMagpie` for additional rewards.

To elaborate, we show below the code snippet of the `depositLP()` routine. As the name indicates, it is designed for users to deposit `Wombat LP`. Per current design, the amount of `receiptToken` to mint is the same as the amount of the `depositToken`. Therefore, in order to get the intended amount of `receiptToken` for the deposit of `Wombat LP`, it needs to calculate the amount of the `depositToken` (line 348). To achieve that, the contract implements the `getDepositTokenAmtByLP()` routine (as the code shown below) by referring to the logic in `Wombat`. However, it comes to our attention that the amount returned from the `getDepositTokenAmtByLP()` routine is in `WAD`, not in the decimals of the `depositToken`! To correct, there's a need to convert the decimals from `WAD` to the decimals of the `depositToken`.

Our analysis shows that `Wombat` provides an official interface, i.e., `quotePotentialWithdraw()`, which can be used to get the amount of the `depositToken` for the given amount of `LP` token.

```
334     function depositLP(
335         address _lpAddress,
336         uint256 _lpAmount,
337         address _for
338     ) nonReentrant whenNotPaused _onlyActivePoolHelper(_lpAddress) external {
339         // Get information of the Pool of the token
340         Pool storage poolInfo = pools[_lpAddress];
341 
342         // Transfer lp to this contract and stake it to wombat
343         IERC20(poolInfo.lpAddress).safeTransferFrom(_for, address(this), _lpAmount);
344 
345         _toMasterWomAndSendReward(_lpAddress, _lpAmount, true); // triggers harvest from
                wombat exchange
346 
347         // update variables
348         uint256 caledDepositTokenAmount = getDepositTokenAmtByLP(_lpAmount, poolInfo.
              lpAddress, poolInfo.depositTarget);
```

```
        uint256 receiptTokenToMint = getSharesForDepositTokens(caledDepositTokenAmount,
            _lpAddress);
        IMintableERC20(poolInfo.receiptToken).mint(msg.sender, receiptTokenToMint);
        ...
    }
```

Listing 3.7: WombatStaking:depositLP()

```
function getDepositTokenAmtByLP(uint256 _amount, address _lpToken, address
    _depositTarget)
    public
    view
    returns (
        uint256 amount
    )
{
    IAsset asset = IAsset(_lpToken);

    require(asset.totalSupply() > 0 && asset.liability() > 0, "Insufficient liquidity");

    uint256 ampFactor = IWombatPool(_depositTarget).ampFactor();
    uint256 liabilityToBurn = (asset.liability() * _amount) / asset.totalSupply();

    amount = withdrawalAmountInEquilImpl(
        -int256(liabilityToBurn),
        int256(uint256(asset.cash())),
        int256(uint256(asset.liability())),
        int256(ampFactor)
    ).toUint256();
}
```

Listing 3.8: WombatStaking:getDepositTokenAmtByLP()

Similarly, as shown in the following code snippet, the `withdraw()` routine invokes a `getLPTokensForShares()` routine (line 372) to get the amount of `LP` token to withdraw from `MasterWombat`. Our analysis shows that `Wombat` also provides an interface, i.e., `quotePotentialDeposit()`, which can be used to get the amount of `LP` token for the given amount of the `depositToken`.

```
363     function withdraw(
364         address _lpToken,
365         uint256 _amount,
366         uint256 _minAmount,
367         address _sender
368     ) nonReentrant whenNotPaused _onlyPoolHelper(_lpToken) external {
369         // _amount is the amount of stable
370         Pool storage poolInfo = pools[_lpToken];
371         uint256 sharesAmount = getSharesForDepositTokens(_amount, _lpToken);
372         uint256 lpAmount = getLPTokensForShares(sharesAmount, _lpToken);
373 
374         IERC20(poolInfo.lpAddress).approve(poolInfo.depositTarget, lpAmount);
375         _toMasterWomAndSendReward(_lpToken, lpAmount, false);
376 
```

```
        uint256 beforeWithdraw = IERC20(poolInfo.depositToken).balanceOf(address(this));
        IWombatPool(poolInfo.depositTarget).withdraw(
            poolInfo.depositToken,
            lpAmount,
            _minAmount,
            address(this),
            block.timestamp
        );
        ...
    }
```

Listing 3.9: WombatStaking:withdraw()

Recommendation Properly use the correct `Wombat` interfaces to quote between the `depositToken` and the `LP` token.

Status The issue has been fixed by this commit: `748be39`. And the team have refactored the contract to keep the amounts of the `receiptToken` and the deposited `LP` token as $1 : 1$. Moreover, users need to provide the amount of `LP` token to the `withdraw()` routine. By doing so, there is no need to convert the amount between the `depositToken` and the `LP` token any more in this contract.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 3.10: 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 `compound()` routine in the `ManualCompound` contract. If the `ZRX` token is supported as `bonusTokenAddresses[j]`, the unsafe version of `IERC20(bonusTokenAddresses[j]).transfer(msg.sender, rewardBalance)` (line 144) 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 present in `WombatStaking::_sendRewards()`.

```
129     function compound(address[] calldata _lps, bool _forLock) external {
130         uint256 rewardTokensLength = rewards.length;
131         uint256[] memory beforeBalances = new uint256[](rewardTokensLength);
132         for (uint256 i; i < rewardTokensLength; i++) {
133             beforeBalances[i] = IERC20(rewards[i].tokenAddress).balanceOf(msg.sender);
134         }
135
136         IMasterMagpie(masterMagpie).multiclaimOnBehalf(_lps, msg.sender);
137         address[] memory bonusTokenAddresses;
138         // send none compoundable reward back to caller
139         for(uint256 i; i < _lps.length; i++) {
140             (bonusTokenAddresses, ) = IMasterMagpie(masterMagpie).rewarderBonusTokenInfo
                    (_lps[i]);
141             for (uint j; j < bonusTokenAddresses.length; j++) {
142                 if (!compoundableRewards[bonusTokenAddresses[j]]) {
```

```
                    uint256 rewardBalance = IERC20(bonusTokenAddresses[j]).balanceOf(
                        address(this));
                    IERC20(bonusTokenAddresses[j]).transfer(msg.sender, rewardBalance);
                }
            }
        }
        ...
    }
```

Listing 3.11: `ManualCompound::compound()`

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. Note the same issue is also present in other routines, e.g., `WombatStaking.deposit()/_sendRewards()` routines, etc.

Recommendation Accommodate the above-mentioned idiosyncrasies with safe-version implementation of ERC20-related `approve()/transfer()`. 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: `748be39`.

4 Conclusion

In this audit, we have analyzed the Magpie design and implementation. Magpie is an innovative yield-boosting protocol that provides users with boosted stablecoin yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. Conversely, WOM holders can also benefit from Magpie by converting their WOM token into mWOM to earn a share of Magpie's profit. 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-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] PeckShield. PeckShield Inc. https://www.peckshield.com.