SMART CONTRACT AUDIT REPORT

for

Wombat Exchange

Prepared By: Patrick Lou

PeckShield

April 18, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	April 18, 2022	Luck Hu	Final Release
1.0-rc	April 10, 2022	Luck Hu	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Wombat 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 Wombat

The Wombat is a BNB-native multichain stableswap with a unique invariant curve that has a closed-form solution (and is more computationally efficient when compared to the Curve's solution). In addition, Wombat uses the concept of asset liability management to get rid of liquidity constraints and thus remove existing scalability barriers. The overall vision of Wombat is to fuel the DeFi growth and push boundaries with greater capital efficiency, accessibility, and scalability in a multichain world. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Wombat

Item	Description
Name	Wombat Exchange
Website	https://www.wombat.exchange/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 18, 2022

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

- https://github.com/wombat-exchange/wombat.git (f3349a4)

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

- https://github.com/wombat-exchange/wombat.git (7208823)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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 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

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 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) [8], 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 `Wombat` 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	0	
Medium	2	■■
Low	4	■■■■
Informational	2	■■
Total	8	

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

Table 2.1: Key Wombat Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Public Writable _unlockIntervalsCount From vestedAmount()	Business Logic	Fixed
PVE-002	Low	Improved Initialization Logic in VeERC20Upgradeable	Coding Practices	Fixed
PVE-003	Low	Suggested Use Of whenNotPaused For depositFor()	Coding Practices	Fixed
PVE-004	Low	Proper Cleanup in emergencyWithdraw()	Business Logic	Fixed
PVE-005	Informational	Consistent WAD Denomination Between minimumAmount And amount	Code Practices	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-007	Low	Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-008	Informational	Suggested Events Generation In Wombat	Coding Practices	Fixed

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 Public Writable _unlockIntervalsCount From vestedAmount()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: TokenVesting
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The TokenVesting contract handles the vesting of WOM (Wombat token) for a list of admin-settable beneficiaries. The WOM token transferred to this contract will be locked and the contract will release the token to the beneficiary according to a given vesting schedule. With the vesting schedule, 10% of the total tokens will be unlocked in each interval (6 months). And a total of 10 intervals will be taken to unlock all the tokens. The contract maintains a _unlockIntervalsCount variable for each beneficiary to record the number of unlocked intervals. By design, the _unlockIntervalsCount shall be updated each time the vested tokens have been released to the beneficiary. While examining the logic to update the _unlockIntervalsCount, we notice the variable could be updated publicly from the vestedAmount() routine, which needs to be corrected.

To elaborate, we show below the code snippet from the TokenVesting contract. As the name indicates, the vestedAmount() routine is designed to calculate and return the amount of WOM tokens that have already been vested to the given beneficiary by the given timestamp. The vestedAmount() invokes the _vestingSchedule() routine which implements the vesting formula. Especially, when the given timestamp equals the current block.timestamp, the _unlockIntervalsCount will be updated to the latest (line 172). It comes to our attention that the vestedAmount() routine is public accessible. That is to say, everybody could invoke it to update the _unlockIntervalsCount of any beneficiary. As a result, the release of the vested tokens to the beneficiary will be delayed.

```
    /**
     * @dev Calculates the amount of WOM tokens that has already vested. Default implementation is a linear vesting curve.
     */
    function vestedAmount(address beneficiary, uint256 timestamp) public returns (uint256) {
        uint256 _vestedAmount = _vestingSchedule(
            beneficiary,
            _beneficiaryInfo[beneficiary]._allocationBalance + released(beneficiary),
            uint256(timestamp)
        );
        emit ReleasableAmount(beneficiary, _vestedAmount);
        return _vestedAmount;
    }

    /**
     * @dev implementation of the vesting formula. This returns the amount vested, as a function of time, for
     * an asset given its total historical allocation.
     * 10% of the Total Number of Tokens Purchased shall unlock every 6 months from the Network Launch,
     * with the Total Number * of Tokens Purchased becoming fully unlocked 5 years from the Network Launch.
     * i.e. 6 months cliff from TGE, 10% unlock at month 6, 10% unlock at month 12, and final 10% unlock at month 60
     */
    function _vestingSchedule(
        address beneficiary,
        uint256 totalAllocation,
        uint256 timestamp
    ) internal returns (uint256) {
        if (timestamp < start()) {
            return 0;
        } else if (timestamp > start() + duration()) {
            return totalAllocation;
        } else if (timestamp == uint256(block.timestamp)) {
            uint256 currentInterval = _calculateInterval(timestamp);
            bool isUnlocked = currentInterval > _beneficiaryInfo[beneficiary]._unlockIntervalsCount;
            if (isUnlocked) {
                _beneficiaryInfo[beneficiary]._unlockIntervalsCount = currentInterval;
                return (totalAllocation * currentInterval * 10) / 100;
            }
        } else {
            return ((totalAllocation * _calculateInterval(timestamp) * 10) / 100);
        }
    }
```

Listing 3.1: TokenVesting.sol

Recommendation Correct the above mentioned logic to update the `_unlockIntervalsCount` only after the vested tokens have been released to the beneficiary.

Status The issue has been fixed by this commit: `8bb1735`.

3.2 Improved Initialization Logic in VeERC20Upgradeable

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VeWom`, `VeERC20Upgradeable`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The `VeWom` contract allows for lazy contract initialization, i.e., the initialization does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the `initializer()` and `onlyInitializing()` modifiers. The `initializer()` protects an initializer function from being invoked twice, and the `onlyInitializing()` modifier protects an initialization function so that it can only be invoked by functions with the `initializer()` modifier, directly or indirectly. While examining the usage of these two modifiers, we notice the existence of abuse of the `initializer()` modifier, which needs to be corrected.

To elaborate, we show below the code snippet of the `VeWom::initialize()` routine. As the name indicates, it is an initialization function for the `VeWom` contract. The `VeWom::initialize()` is protected by the `initializer()` and it further invokes the subcalls to `__ERC20_init()/__Ownable_init()`, etc. (lines 62 - 65). It comes to our attention that the `initializer()` is also applied to `__ERC20_init()/__ERC20_init_unchained()` (line 35 and line 40) in the `VeERC20Upgradeable` contract. As a result, the initialization will fail at the validation (line 53) in the `Initializable::initializer()`, because the `_initializing/_initialized` have both been set to `true` by the `initializer()` of `VeWom::initialize()`. It is suggested to protect the subcalls with the `onlyInitializing()` modifier, as recommended by `Openzeppelin`: #3006.

```
57     function initialize(IERC20 _wom, IMasterWombat _masterWombat) external initializer {
58         require(address(_masterWombat) != address(0), 'zero address');
59         require(address(_wom) != address(0), 'zero address');
60 
61         // Initialize veWOM
62         __ERC20_init('Wombat Waddle', 'veWOM');
63         __Ownable_init();
64         __ReentrancyGuard_init_unchained();
65         __Pausable_init_unchained();
66 
67         masterWombat = _masterWombat;
68         wom = _wom;
69 
```

```
        // Note: one should pay attention to storage collision
        maxBreedingLength = 10000;
        minLockDays = 7;
        maxLockDays = 1461;
    }
```

Listing 3.2: `VeWom::initialize()`

```
    function __ERC20_init(string memory name_, string memory symbol_) internal
        initializer {
        __Context_init_unchained();
        __ERC20_init_unchained(name_, symbol_);
    }

    function __ERC20_init_unchained(string memory name_, string memory symbol_) internal
         initializer {
        _name = name_;
        _symbol = symbol_;
    }
```

Listing 3.3: `VeERC20Upgradeable.sol`

```
modifier initializer() {
    // If the contract is initializing we ignore whether _initialized is set in order to
         support multiple
    // inheritance patterns, but we only do this in the context of a constructor,
        because in other contexts the
    // contract may have been reentered.
    require(_initializing ? _isConstructor() : !_initialized, "Initializable: contract
        is already initialized");

    bool isTopLevelCall = !_initializing;
    if (isTopLevelCall) {
        _initializing = true;
        _initialized = true;
    }

    _;

    if (isTopLevelCall) {
        _initializing = false;
    }
}
```

Listing 3.4: `Initializable::initializer()`

Recommendation Enforce the `__ERC20_init()/__ERC20_init_unchained()` subcalls with the `onlyInitializing` modifier.

Status The issue has been fixed by this commit: `f437a44`.

3.3 Suggested Use Of whenNotPaused For depositFor()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterWombat`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The `MasterWombat` contract is the `MasterChef` implementation of the `Wombat` protocol where users could deposit `LP` tokens to earn `WOM` rewards. In the `MasterWombat` contract, the `owner` has the privilege to pause the current contract. This feature is designed for emergency use only. And normal operations of the `MasterWombat` are only allowed when the contract is not paused. Our analysis shows that there are still three functions that somehow still allow transactions to proceed even when the current contract is paused.

To elaborate, we take the `depositFor()` routine for example and show blow the code snippets of the `depositFor()/deposit()` routines. As the names indicate, the two routines support the caller to deposit `LP` tokens to `MasterWombat` for the `WOM` allocation. It comes to our attention that the `deposit()` is protected by the `whenNotPaused` modifier (line 411), but the `depositFor()` is not, which means the `whenNotPaused` of `deposit()` could be bypassed by invoking `depositFor()` instead. So it is suggested to apply `whenNotPaused` to `depositFor()` as well.

Note the same issue also exists in the `migrate()/emergencyWithdraw()` routines, which shall be enforced by adding the `whenNotPaused` modifier.

```
362     function depositFor(
363         uint256 _pid,
364         uint256 _amount,
365         address _user
366     ) external override nonReentrant {
367         PoolInfo storage pool = poolInfo[_pid];
368         UserInfo storage user = userInfo[_pid][_user];

370         // update pool in case user has deposited
371         _updatePool(_pid);
372         if (user.amount > 0) {
373             // Harvest WOM
374             uint256 pending = ((user.amount * pool.accWomPerShare + user.factor * pool.
                    accWomPerFactorShare) / 1e12) +
375                 pendingWom[_pid][_user] -
376                 user.rewardDebt;
377             pendingWom[_pid][_user] = 0;

379             pending = safeWomTransfer(payable(_user), pending);
```

```
            emit Harvest(_user, _pid, pending);
        }

        // update amount of lp staked by user
        user.amount += _amount;

        // update boosted factor
        uint256 oldFactor = user.factor;
        user.factor = DSMath.sqrt(user.amount * veWom.balanceOf(_user), user.amount);
        pool.sumOfFactors = pool.sumOfFactors + user.factor - oldFactor;

        // update reward debt
        user.rewardDebt = (user.amount * pool.accWomPerShare + user.factor * pool.
            accWomPerFactorShare) / 1e12;

        IRewarder rewarder = poolInfo[_pid].rewarder;
        if (address(rewarder) != address(0)) {
            rewarder.onReward(_user, user.amount);
        }

        pool.lpToken.safeTransferFrom(msg.sender, address(this), _amount);
        emit DepositFor(_user, _pid, _amount);
    }

    /// @notice Deposit LP tokens to MasterChef for WOM allocation.
    /// @dev it is possible to call this function with _amount == 0 to claim current
        rewards
    /// @param _pid the pool id
    /// @param _amount amount to deposit
    function deposit(uint256 _pid, uint256 _amount)
        external
        override
        nonReentrant
        whenNotPaused
        returns (uint256, uint256)
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        _updatePool(_pid);
        uint256 pending;
        if (user.amount > 0) {
            // Harvest WOM
            pending =
                ((user.amount * pool.accWomPerShare + user.factor * pool.
                    accWomPerFactorShare) / 1e12) +
                pendingWom[_pid][msg.sender] -
                user.rewardDebt;
            pendingWom[_pid][msg.sender] = 0;

            pending = safeWomTransfer(payable(msg.sender), pending);
            emit Harvest(msg.sender, _pid, pending);
        }
```

```
        // update amount of lp staked by user
        user.amount += _amount;

        // update boosted factor
        uint256 oldFactor = user.factor;
        user.factor = DSMath.sqrt(user.amount * veWom.balanceOf(msg.sender), user.amount
            );
        pool.sumOfFactors = pool.sumOfFactors + user.factor - oldFactor;

        // update reward debt
        user.rewardDebt = (user.amount * pool.accWomPerShare + user.factor * pool.
            accWomPerFactorShare) / 1e12;

        IRewarder rewarder = poolInfo[_pid].rewarder;
        uint256 additionalRewards;
        if (address(rewarder) != address(0)) {
            additionalRewards = rewarder.onReward(msg.sender, user.amount);
        }

        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        emit Deposit(msg.sender, _pid, _amount);
        return (pending, additionalRewards);
    }
```

Listing 3.5: MasterWombat.sol

Recommendation Add `whenNotPaused` to the `depositFor()`/`migrate()`/`emergencyWithdraw()` routines.

Status The issue has been fixed by this commit: `3e988b8`.

3.4 Proper Cleanup in emergencyWithdraw()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterWombat`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `MasterWombat` contract is the `MasterChef` implementation of `Wombat`. It provides an incentive mechanism that rewards the staking of supported assets with the `WOM` token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of assets (with the boosted factor applied) in the reward pool.

In the `MasterWombat` contract, it provides an `emergencyWithdraw()` routine which is mainly designed for emergency use only. To elaborate, we show below the implementation of the `emergencyWithdraw()`. As the name indicates, this routine is designed for staking users to withdraw their assets from the given pool without caring about rewards. However, it comes to our attention that the current logic only returns the assets back to the staking users, but does not reset their pending rewards. As a result, the pending rewards could be reduced in a new deposit to the pool.

```
  /// @notice Withdraw without caring about rewards. EMERGENCY ONLY.
  /// @param _pid the pool id
  function emergencyWithdraw(uint256 _pid) external override nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);

    // update boosted factor
    pool.sumOfFactors = pool.sumOfFactors - user.factor;
    user.factor = 0;

    // update base factors
    user.amount = 0;
    user.rewardDebt = 0;

    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
}
```

Listing 3.6: MasterWombat::emergencyWithdraw()

Recommendation Revise the `emergencyWithdraw()` logic to properly reset the pending rewards of the caller.

Status This issue has been fixed in this commit: `692e4d6`.

3.5 Consistent WAD Denomination Between minimumAmount And amount

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Pool
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The Wombat protocol implements a WAD converter to ensure prices amongst stable coins with different decimals are calculated accurately. This is because all cryptocurrency amounts are internally normalized to the WAD decimals before performing any operation on them. And a WAD is a decimal number with 18 digits of precision.

To elaborate, we show below the implementation of the withdraw() routine. As the name indicates, this routine is designed for LP to withdraw their assets to the given address while ensuring the minimum withdrawn amount is met. It further calls the _withdraw() routine with the liquidity amount of assets to withdraw and the minimumAmount of the underlying tokens to accept. The _withdraw() routine calculates the amount of the underlying tokens and compares it with the given minimumAmount to check whether the withdraw operation may proceed or not. While examining the precisions of amount and minimumAmount, we notice they possibly share different precisions. Because the amount is a WAD (18 decimals), while the precision of the minimumAmount is given by the asset.underlyingTokenDecimals() which possibly may not be equal to 18. As a result, the comparison between amount and minimumAmount may give an unexpected result which may wrongly proceed/refuse the withdrawal. So it is suggested to convert the minimumAmount for the WAD denomination as well.

```
    /**
     * @notice Withdraws liquidity amount of asset to ‘to‘ address ensuring minimum
        amount required
     * @param asset The asset to be withdrawn
     * @param liquidity The liquidity to be withdrawn
     * @param minimumAmount The minimum amount that will be accepted by user
     * @return amount The total amount withdrawn
     */
    function _withdraw(
        IAsset asset,
        uint256 liquidity,
        uint256 minimumAmount
    ) private returns (uint256 amount) {
        // collect fee before withdraw
        _mintFee(asset);

```

```
        // calculate liabilityToBurn and Fee
        uint256 liabilityToBurn;
        (amount, liabilityToBurn, ) = _withdrawFrom(asset, liquidity);
        _checkAmount(minimumAmount, amount);

        asset.burn(address(asset), liquidity);
        asset.removeCash(amount);
        asset.removeLiability(liabilityToBurn);

        // revert if cov ratio < 1% to avoid precision error
        if (asset.liability() > 0 && uint256(asset.cash()).wdiv(asset.liability()) < WAD
            / 100)
            revert WOMBAT_FORBIDDEN();
    }

    /**
     * @notice Withdraws liquidity amount of asset to `to` address ensuring minimum
         amount required
     * @param token The token to be withdrawn
     * @param liquidity The liquidity to be withdrawn
     * @param minimumAmount The minimum amount that will be accepted by user
     * @param to The user receiving the withdrawal
     * @param deadline The deadline to be respected
     * @return amount The total amount withdrawn
     */
    function withdraw(
        address token,
        uint256 liquidity,
        uint256 minimumAmount,
        address to,
        uint256 deadline
    ) external nonReentrant whenNotPaused returns (uint256 amount) {
        _checkLiquidity(liquidity);
        _checkAddress(to);
        _ensure(deadline);

        IAsset asset = _assetOf(token);
        // request lp token from user
        IERC20(asset).safeTransferFrom(address(msg.sender), address(asset), liquidity);
        amount = _withdraw(asset, liquidity, minimumAmount).fromWad(asset.
            underlyingTokenDecimals());
        asset.transferUnderlyingToken(to, amount);

        emit Withdraw(msg.sender, token, amount, liquidity, to);
    }
```

Listing 3.7: `Pool.sol`

Note the same issue also exists in the `withdrawFromOtherAsset()` routine, where the input `minimumAmount` argument shall be converted to a `WAD` before any calculation.

Recommendation Revise the above mentioned `withdraw()/withdrawFromOtherAsset()` routines to convert the input `minimumAmount` argument with the `WAD` denomination before any calculation.

Status This issue has been fixed in below commits: `5fc8154` and `692e4d6`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

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

Firstly, the privileged functions in the `Pool` contract allow for the the `owner` to configure parameters for the pools and transfer the `tip bucket` out from the contract.

```
/**
* @notice Changes the pools haircutRate. Can only be set by the contract owner.
* @param haircutRate_ new pool's haircutRate_
*/
function setHaircutRate(uint256 haircutRate_) external onlyOwner {
    if (haircutRate_ > WAD) revert WOMBAT_INVALID_VALUE(); // haircutRate_ should not be set bigger than 1
    haircutRate = haircutRate_;
}

function setFee(uint256 lpDividendRatio_, uint256 retentionRatio_) external onlyOwner {
    if (retentionRatio_ + lpDividendRatio_ > WAD) revert WOMBAT_INVALID_VALUE();
    mintAllFee();
    retentionRatio = retentionRatio_;
    lpDividendRatio = lpDividendRatio_;
}

/**
* @notice Changes the fee beneficiary. Can only be set by the contract owner.
* This value cannot be set to 0 to avoid unsettled fee.
* @param feeTo_ new fee beneficiary
*/
function setFeeTo(address feeTo_) external onlyOwner {
    if (feeTo_ == address(0)) revert WOMBAT_INVALID_VALUE();
    feeTo = feeTo_;
}
```

Listing 3.8: `Pool.sol`

```
    function transferTipBucket(
        address token,
        uint256 amount,
        address to
    ) external onlyOwner {
        IAsset asset = _assetOf(token);
        uint256 tipBucketBal = asset.underlyingTokenBalance().toWad(asset.
            underlyingTokenDecimals()) -
            asset.cash() -
            _feeCollected[asset];

        if (amount > tipBucketBal) {
            // revert if there's not enough amount in the tip bucket
            revert WOMBAT_INVALID_VALUE();
        }

        asset.transferUnderlyingToken(to, amount.fromWad(asset.underlyingTokenDecimals()
            ));
    }
```

Listing 3.9: `Pool::transferTipBucket()`

Secondly, the privileged functions in the `MasterWombat` contract allows for the the `owner` to configure parameters for the contract and emergency withdraw `WOM` funds from the contract. In particular, the `owner` is privileged to set `newMasterWombat` which could accept user funds from the migration.

```
    function setNewMasterWombat(IMasterWombat _newMasterWombat) external onlyOwner {
        newMasterWombat = _newMasterWombat;
    }
```

Listing 3.10: `MasterWombat::setNewMasterWombat()`

```
    /// @notice In case we need to manually migrate WOM funds from MasterChef
    /// Sends all remaining wom from the contract to the owner
    function emergencyWomWithdraw() external onlyOwner {
        wom.safeTransfer(address(msg.sender), wom.balanceOf(address(this)));
    }
```

Listing 3.11: `MasterWombat::emergencyWomWithdraw()`

Lastly, the privileged function in the `TokenVesting` contract allows for the `owner` to add new beneficiary who can receive `WOM` tokens from vesting.

```
    /**
     * @dev Setter for adding a beneficiary address.
     */
    function setBeneficiary(address beneficiary, uint256 allocation) external onlyOwner
        {
        require(beneficiary != address(0), 'Beneficiary: address cannot be zero');
        require(_beneficiaryInfo[beneficiary]._allocationBalance == 0, 'Beneficiary:
            allocation already set');
        _beneficiaryInfo[beneficiary] = BeneficiaryInfo(allocation, 0, 0);
```

```
        _totalAllocationBalance += allocation;
        _beneficiaryAddresses.push(beneficiary);
        emit BeneficiaryAdded(beneficiary, allocation);
    }
```

Listing 3.12: `TokenVesting::setBeneficiary()`

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

Recommendation Make the list of extra privileges granted to `owner` explicit to `Wombat` protocol users.

Status This issue has been confirmed.

3.7 Incompatibility With Deflationary Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Wombat` protocol, the `Pool` contract acts as the main entry for interaction with trading users. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and mints new `LP` tokens to user. Another interface, i.e, `swap()`, accepts asset transfer-in and sends asset transfer-out to user.

For the above two operations, i.e., `deposit()` and `swap()`, the contract makes the use of `safeTransferFrom()` routine to transfer assets into its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(
        address token,
        uint256 amount,
        uint256 minimumLiquidity,
        address to,
        uint256 deadline,
        bool shouldStake
    ) external nonReentrant whenNotPaused returns (uint256 liquidity) {
```

```
        if (amount == 0) revert WOMBAT_ZERO_AMOUNT();
        _checkAddress(to);
        _ensure(deadline);
        requireAssetNotPaused(token);

        IAsset asset = _assetOf(token);
        IERC20(token).safeTransferFrom(address(msg.sender), address(asset), amount);

        if (!shouldStake) {
            liquidity = _deposit(asset, amount.toWad(asset.underlyingTokenDecimals()),
                minimumLiquidity, to);
        } else {
            _checkAddress(address(masterWombat));
            // deposit and stake on behalf of the user
            liquidity = _deposit(asset, amount.toWad(asset.underlyingTokenDecimals()),
                minimumLiquidity, address(this));

            asset.approve(address(masterWombat), liquidity);

            uint256 pid = masterWombat.getAssetPid(address(asset));
            masterWombat.depositFor(pid, liquidity, to);
        }

        emit Deposit(msg.sender, token, amount, liquidity, to);
    }
```

Listing 3.13: Pool::deposit()

```
function swap(
        address fromToken,
        address toToken,
        uint256 fromAmount,
        uint256 minimumToAmount,
        address to,
        uint256 deadline
    ) external nonReentrant whenNotPaused returns (uint256 actualToAmount, uint256
        haircut) {
        _checkSameAddress(fromToken, toToken);
        if (fromAmount == 0) revert WOMBAT_ZERO_AMOUNT();
        _checkAddress(to);
        _ensure(deadline);
        requireAssetNotPaused(fromToken);

        IAsset fromAsset = _assetOf(fromToken);
        IAsset toAsset = _assetOf(toToken);

        uint8 toDecimal = toAsset.underlyingTokenDecimals();

        (actualToAmount, haircut) = _swap(
            fromAsset,
            toAsset,
            fromAmount.toWad(fromAsset.underlyingTokenDecimals()),
            minimumToAmount.toWad(toDecimal)
```

```
        );

        actualToAmount = actualToAmount.fromWad(toDecimal);
        haircut = haircut.fromWad(toDecimal);

        IERC20(fromToken).safeTransferFrom(msg.sender, address(fromAsset), fromAmount);
        toAsset.transferUnderlyingToken(to, actualToAmount);

        emit Swap(msg.sender, fromToken, toToken, fromAmount, actualToAmount, to);
    }
```

Listing 3.14: Pool::swap()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind the asset-transferring routines. In other words, the above operations, such as deposit(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in safeTransfer() or safeTransferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the safeTransfer() or safeTransferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into Wombat for support.

Note the deposit() and depositFor() routines in the MasterWombat contract share the same issue.

Recommendation Check the balance before and after the safeTransferFrom() call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed.

3.8 Suggested Events Generation In Wombat

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the newMasterWombat dynamics in MasterWombat, we notice there is a lack of emitting an event to reflect newMasterWombat changes. To elaborate, we show below the related code snippet of the contract.

```
    function setNewMasterWombat(IMasterWombat _newMasterWombat) external onlyOwner {
        newMasterWombat = _newMasterWombat;
    }
```

Listing 3.15: MasterWombat::setNewMasterWombat()

With that, we suggest to add a new event NewMasterWombat whenever the new newMasterWombat is changed. Also, the new newMasterWombat information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the newMasterWombat information is typically queried, it is better treated as a topic, hence the need of being indexed.

Note the other routines, i.e., VeWom::setMasterWombat(), VeWom::setWhitelist(), VeWom::setMaxBreedingLength(), MasterWombat::emergencyWomWithdraw(), Pool::setDev(), Pool::setMasterWombat(), Pool::setAmpFactor(), Pool::setHaircutRate(), Pool::setFee(), Pool::setFeeTo(), Pool::setMintFeeThreshold(), Pool::transferTipBucket(), and Asset::setMaxSupply() can also benefit from the meaningful events generation.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: 7f530e3.

4 Conclusion

In this audit, we have analyzed the Wombat protocol design and implementation. The protocol is a decentralized exchange for stableswap with the features of open liquidity pools, single sided staking and low slippage. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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