SMART CONTRACT AUDIT REPORT

for

Wombat v3

Prepared By: Xiaomi Huang

PeckShield

November 16, 2022

Document Properties

Client	Wombat Exchange
Title	Smart Contract Audit Report
Target	Wombat v3
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 16, 2022	Luck Hu	Final Release
1.0-rc	November 10, 2022	Luck Hu	Release Candidate

Contact

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

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

Contents

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the 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 v3

The Wombat is a BNB-native stableswap protocol with open-liquidity pools, low slippage and single-sided staking. It brings greater capital efficiency to fuel DeFi growth and adoption. The new Wombat protocol introduces a new emissions distribution mechanism, which allows the veWOM holders to vote on how WOM emissions can be distributed to different gauges. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Wombat v3

Item	Description
Name	Wombat Exchange
Website	https://www.wombat.exchange/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 16, 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 (e347de6)

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 (a0eb54a)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

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

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

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

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. 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	2	■■
Informational	0	
Total	4	

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

2.2 Key Findings

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

Table 2.1: Key Wombat v3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic to Distribute WOM emissions in MasterWombat	Business Logic	Fixed
PVE-002	Medium	Timely massUpdatePools During basePartition Update	Business Logic	Fixed
PVE-003	Low	Timely Update of base/voteIndex in resumeAll()	Business Logic	Fixed
PVE-004	Low	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited Logic to Distribute WOM emissions in MasterWombat

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MasterWombatV3`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Wombat` protocol introduces a new emissions distribution mechanism, which allows the `veWOM` holders to vote on how `WOM` emissions can be distributed to different gauges according to the allocation and voting weight. The `MasterWombatV3` contract is responsible to claim `WOM` emissions from the `Voter` contract and distribute them to users.

To elaborate, we show below the code snippet of the `_updatePool()` routine, which is invoked at the beginning of each operation (e.g., deposit/withdraw) to claim `WOM` emissions from the `Voter` contract and accumulate the `accWomPerShare/ accWomPerFactorShare`. By design, the new claimed emissions shall be rewarded into the next reward duration. However, current implementation invokes the `IVoter(voter).distribute()` (line 218) first to claim the emissions and update the `rewardRate`, then invokes the `calRewardPerUnit()` (line 221) to accumulate the `accWomPerShare/ accWomPerFactorShare` . As a result, the `accWomPerShare/accWomPerFactorShare` are accumulated per the new `rewardRate`, not the expected old `rewardRate`. Our analysis shows that it shall accumulate the `accWomPerShare/ accWomPerFactorShare` before the claiming of new emissions in the `_updatePool()` routine.

```
216     function _updatePool(uint256 _pid) private {
217         PoolInfo storage pool = poolInfo[_pid];
218         IVoter(voter).distribute(address(pool.lpToken));

220         if (block.timestamp > pool.lastRewardTimestamp) {
```

```
221             (uint256 accWomPerShare, uint256 accWomPerFactorShare) = calRewardPerUnit(
                    _pid);
222             pool.accWomPerShare = to104(accWomPerShare);
223             pool.accWomPerFactorShare = to104(accWomPerFactorShare);
224             pool.lastRewardTimestamp = uint40(lastTimeRewardApplicable(pool.periodFinish
                    ));
225         }
226     }
```

Listing 3.1: MasterWombatV3::_updatePool()

Moreover, the `notifyRewardAmount()` routine is invoked indirectly from the `Voter::distribute()` routine. It is used to notify the `MasterWombatV3` contract to update the `pool.rewardRate` and the `pool.periodFinish`. However, it comes to our attention that it also updates the `pool.lastRewardTimestamp` to `block.timestamp`. As a result, it bypasses the accumulation of the `accWomPerShare/ accWomPerFactorShare` in the the `_updatePool()` routine, as the condition `if (block.timestamp > pool.lastRewardTimestamp)` (line 220) becomes false.

```
    function notifyRewardAmount(address _lpToken, uint256 _amount) external override {
        require(_amount > 0, 'notifyRewardAmount: zero amount');

        // this line reverts if asset is not in the list
        uint256 pid = assetPid[_lpToken] - 1;
        PoolInfo storage pool = poolInfo[pid];
        if (block.timestamp >= pool.periodFinish) {
            pool.rewardRate = to128(_amount / REWARD_DURATION);
        } else {
            uint256 remainingTime = pool.periodFinish - block.timestamp;
            uint256 leftoverReward = remainingTime * pool.rewardRate;
            pool.rewardRate = to128((_amount + leftoverReward) / REWARD_DURATION);
        }

        pool.lastRewardTimestamp = uint40(block.timestamp);
        pool.periodFinish = uint40(block.timestamp + REWARD_DURATION);
        ...
    }
```

Listing 3.2: MasterWombatV3::notifyRewardAmount()

Recommendation Revisit the above mentioned logic to accumulate the `accWomPerShare/ accWomPerFactorShare` first per current reward rate, then claim the new emissions for the next reward duration.

Status The issue has been fixed by these commits: `9029448` and `2f29c2b`.

3.2 Timely massUpdatePools During basePartition Update

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MasterWombatV3`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `Wombat` protocol, the `MasterWombatV3` contract is responsible to claim the `WOM` emissions from the `Voter` contract and distribute them to users. The claimed emissions are divided to two partitions, that are the base partition emissions and the boosted partition emissions. The base partition emissions are distributed to users per the amounts of their deposited LPs, and the boosted partition emissions are distributed per users boosted factors.

The amount of base partition emissions is calculated from all the claimed emissions per the percentage `basePartition`, which can be dynamically updated by the owner via the `updateEmissionPartition()` routine. While analyzing the logic to update the `basePartition` in the `updateEmissionPartition()` routine, we notice the need of timely invoking the `massUpdatePools()` routine to accumulate the `accWomPerShare/accWomPerFactorShare` before the new `basePartition` gets effective.

```
function updateEmissionPartition(uint16 _basePartition) external onlyOwner {
  require(_basePartition <= 1000);
  basePartition = _basePartition;
  emit UpdateEmissionPartition(msg.sender, _basePartition, 1000 - _basePartition);
}
```

Listing 3.3: `MasterWombatV3::updateEmissionPartition()`

If the call to the `massUpdatePools()` is not immediately invoked before the new `basePartition` gets effective, certain situations may be crafted to create an unfair reward distribution. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke the `massUpdatePools()` at the beginning of the `updateEmissionPartition()` routine.

Status The issue has been fixed by this commit: `0e92140`.

3.3 Timely Update of base/voteIndex in resumeAll()

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

Description

In the `Wombat` protocol, the `Voter` contract implements the gauge voting, which `veWOM` holders can participate in to apply new allocation for their votes. They can allocate their vote (1 `veWOM` $= 1$ `vote`) to one or more gauges. The `WOM` emission to a gauge is proportional to the amount of vote it receives. Specially, the `WOM` emission to a gauge can be paused by the owner. When a gauge is paused, its un-distributed rewards are forfeited, though users can still vote/unvote and receive bribes.

To elaborate, we show below the code snippets of the `resume()/resumeAll()` routines, which are used to resume the paused gauge(s). Because the un-distributed rewards to a paused gauge are forfeited, so when the gauge is resumed in the `resume()` routine, it invokes the `_distributeWom()` (line 317) and updates its `supplyBaseIndex/supplyVoteIndex` to the latest. However, in the `resumeAll()` routine, which is used to resume all paused gauges, it doesn't catch up the `supplyBaseIndex/supplyVoteIndex` of each paused gauge to the latest. As a result, the un-distributed rewards when the gauge is paused are still available. Based on this, we suggest to update the `supplyBaseIndex/supplyVoteIndex` to the latest for all the paused gauges in the `resumeAll()` routine.

```
312     function resume(IERC20 _lpToken) external onlyOwner {
313         require(infos[_lpToken].whitelist == false, 'voter: not paused');
314         _checkGaugeExist(_lpToken);

316         // catch up supplyVoteIndex
317         _distributeWom();
318         infos[_lpToken].supplyBaseIndex = baseIndex;
319         infos[_lpToken].supplyVoteIndex = voteIndex;
320         infos[_lpToken].whitelist = true;
321     }

323     function resumeAll() external onlyOwner {
324         _unpause();
325     }
```

Listing 3.4: Voter.sol

Recommendation Revisit the logic in the `resumeAll()` routine to forfeit the un-distributed rewards by catching up the `supplyBaseIndex/supplyVoteIndex` of each paused gauge to the latest.

Status The issue has been fixed by this commit: `9844fcd`.

3.4 Trust Issue of Admin Keys

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

Description

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

Specifically, the privileged functions in `Voter` allow for the `owner` to set the `WOM` emission rate, pause/unpause gauge(s), set the gauge manager/bribe address for a LP, emergency withdraw the `WOM` in the contract, etc.

```
    function setWomPerSec(uint88 _womPerSec) external onlyOwner {
        require(_womPerSec <= 10000e18, 'reward rate too high'); // in case `voteIndex`
            overflow
        _distributeWom();
        womPerSec = _womPerSec;
    }

    /// @notice Pause emission of WOM tokens. Un-distributed rewards are forfeited
    /// Users can still vote/unvote and receive bribes.
    function pause(IERC20 _lpToken) external onlyOwner {
        require(infos[_lpToken].whitelist, 'voter: not whitelisted');
        _checkGaugeExist(_lpToken);

        infos[_lpToken].whitelist = false;
    }

    /// @notice Resume emission of WOM tokens
    function resume(IERC20 _lpToken) external onlyOwner {
        require(infos[_lpToken].whitelist == false, 'voter: not paused');
        _checkGaugeExist(_lpToken);

        // catch up supplyVoteIndex
        _distributeWom();
        infos[_lpToken].supplyBaseIndex = baseIndex;
        infos[_lpToken].supplyVoteIndex = voteIndex;
```

```
        infos[_lpToken].whitelist = true;
    }

    /// @notice Pause emission of WOM tokens for all assets. Un-distributed rewards are
        forfeited
    /// Users can still vote/unvote and receive bribes.
    function pauseAll() external onlyOwner {
        _pause();
    }

    /// @notice Resume emission of WOM tokens for all assets
    function resumeAll() external onlyOwner {
        _unpause();
    }

    /// @notice get gaugeManager address for LP token
    function setGauge(IERC20 _lpToken, IGauge _gaugeManager) external onlyOwner {
        require(address(_gaugeManager) != address(0));
        _checkGaugeExist(_lpToken);

        infos[_lpToken].gaugeManager = _gaugeManager;
    }

    /// @notice get bribe address for LP token
    function setBribe(IERC20 _lpToken, IBribe _bribe) external onlyOwner {
        _checkGaugeExist(_lpToken);

        infos[_lpToken].bribe = _bribe; // 0 address is allowed
    }

    /// @notice In case we need to manually migrate WOM funds from Voter
    /// Sends all remaining wom from the contract to the owner
    function emergencyWomWithdraw() external onlyOwner {
        // SafeERC20 is not needed as WOM will revert if transfer fails
        wom.transfer(address(msg.sender), wom.balanceOf(address(this)));
    }
```

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

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

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

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

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Wombat` v3 protocol which is introduced on top of the `Wombat` v2 with new feature to allow the `veWOM` holders to vote on how `WOM` emissions can be distributed to different gauges. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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