SMART CONTRACT AUDIT REPORT

for

Wombat v2

Prepared By: Xiaomi Huang

PeckShield

August 29, 2022

Document Properties

Client	Wombat Exchange
Title	Smart Contract Audit Report
Target	Wombat v2
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	August 29, 2022	Luck Hu	Final Release
1.0-rc	August 15, 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 v2

The `Wombat` v1 is a `BNB`-native stableswap protocol with open-liquidity pool, low slippage and single-sided staking. It brings greater capital efficiency to fuel `DeFi` growth and adoption. On top of the `Wombat` v1, `Wombat` v2 is introduced with innovations to cater to liquid staking tokens, i.e. dynamic pool, and more experimental stablecoins, i.e. sidepool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Wombat v2

Item	Description
Name	Wombat Exchange
Website	https://www.wombat.exchange/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 29, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the `MultiRewarderPerSec.sol` is added into the audit scope in commit `e3c2b62`.

- https://github.com/wombat-exchange/wombat.git (ab5fd7e)

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 (12c5629)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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) [6], 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	3	■■■
Informational	0	
Total	5	

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 3 low-severity vulnerabilities.

Table 2.1: Key Wombat v2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inaccurate haircut Decimal Used in quotePotentialSwap()	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Low	Improved haircut Return in Wombat-Router	Coding Practices	Fixed
PVE-004	Low	Improved receive() to Receive Native Assets	Coding Practices	Fixed
PVE-005	Medium	Timely _updateReward() in addRewardToken()	Business Logic	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 Inaccurate haircut Decimal Used in quotePotentialSwap()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Pool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In `Wombat` protocol, the `Pool` contract provides a `quotePotentialSwap()` function for users to get the maximum output token amount and the haircut for a potential swap. While examining the logic to calculate the maximum output token amount and the haircut, we notice the existence of using inaccurate decimal to convert the haircut.

To elaborate, we show below the code snippet of the `quotePotentialSwap()` routine. As the name indicates, it provides quote for a potential swap which is given by three parameters. The first parameter (i.e. `fromToken`) is the token that the user wants to provide for the swap, and the second parameter (i.e. `toToken`) is the token that the user wants to receive from the swap. The last parameter (i.e. `fromAmount`) gives the amount of the `fromToken` the user want to provide for the swap. Specially, if the `fromAmount < 0`, it is a reverse quote where the last parameter (i.e. `fromAmount`) gives the amount of the `fromToken` the user want to receive from the swap. Accordingly, the first parameter (i.e. `fromToken`) is the target token that the user wants to receive and the second parameter (i.e. `toToken`) is the token that the user wants to provide in the swap. In the case of reverse quote, the haircut is charged in the `fromToken`. So the haircut needs to be converted from `WAD` to the decimal of the `fromToken`. The current haircut conversion from `WAD` to the decimal of the `toToken` (line 866) is inaccurate which needs to be fixed.

```
842     /**
843     * @notice Given an input asset amount and token addresses, calculates the
844     * maximum output token amount (accounting for fees and slippage).
```

```
     * @dev To be used by frontend
     * @param fromToken The initial ERC20 token
     * @param toToken The token wanted by user
     * @param fromAmount The given input amount
     * @return potentialOutcome The potential amount user would receive
     * @return haircut The haircut that would be applied
     */
    function quotePotentialSwap(
        address fromToken,
        address toToken,
        int256 fromAmount
    ) public view override returns (uint256 potentialOutcome, uint256 haircut) {
        _checkSameAddress(fromToken, toToken);
        if (fromAmount == 0) revert WOMBAT_ZERO_AMOUNT();

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

        fromAmount = fromAmount.toWad(fromAsset.underlyingTokenDecimals());
        (potentialOutcome, haircut) = _quoteFrom(fromAsset, toAsset, fromAmount);
        potentialOutcome = potentialOutcome.fromWad(toAsset.underlyingTokenDecimals());
        haircut = haircut.fromWad(toAsset.underlyingTokenDecimals());
    }
```

Listing 3.1: Pool::quotePotentialSwap()

Recommendation Correct the haircut decimal in the case of reverse quote.

Status The issue has been fixed by this commit: `c6b2045`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WombatRouter`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [2]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to 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).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses'
        //  allowance to zero by calling 'approve(_spender, 0)' if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.2: USDT Token **Contract**

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.

More importantly, the `approve()` function of some token may return false while not revert on failure. Accordingly, the call to `approve()` is expected to check the return value. If it returns false, the call to `approve()` shall be failed.

Because of that, 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. 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

In the following, we show the `approve()` routine in the `WombatRouter` contract. If the `approve()` of the given `tokens[i]` does not revert on failure, the unsafe version of `IERC20(tokens[i]).approve(pool, type(uint256).max)` (line 40) need to check the return value while not assuming the `approve()` will revert internally.

```
34      /// @notice approve spending of router tokens by pool
35      /// @param tokens array of tokens to be approved
36      /// @param pool to be approved to spend
37      /// @dev needs to be done after asset deployment for router to be able to support
            the tokens
```

```
    function approveSpendingByPool(address[] calldata tokens, address pool) external
        onlyOwner {
        for (uint256 i; i < tokens.length; ++i) {
            IERC20(tokens[i]).approve(pool, type(uint256).max);
        }
    }
```

Listing 3.3: WombatRouter::approveSpendingByPool()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been fixed by this commit: `c6b2045`.

3.3 Improved haircut Return in WombatRouter

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WombatRouter`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `WombatRouter` is a helper contract that brings convenience to users by allowing them interact with a single router contract in swapping across all assets available within the pools, fulfilling the single global pool vision. It provides the ability for users to quote, reverse quote, deposit, withdraw, and swap across all pools on `Wombat`. While reviewing the swap functionality, we notice it could be improved to return a meaningful haircut.

To elaborate, we take the `swapExactTokensForNative()` routine for example and show blow the code snippet from the `WombatRouter`. As the names indicate, the `swapExactTokensForNative()` is used to facilitate users to swap exact tokens for the native token, and the `_swap()` implements the actual logic of the swap. Specifically, for each hop of the swap, a haircut is charged in the target token. The `_swap()` accumulates the haircut from each hop to a single amount (line 194) and returns the accumulated haircut back to user. However, we notice that the target token in each swap hop is different, so it does not make sense to accumulate all haircut together into one single amount. Based on this, it is suggested to return a haircut array with each item records the haircut for each swap hop, or simply just remove the haircut from the returns of the `_swap()`.

```
140     function swapExactTokensForNative(
141         address[] calldata tokenPath,
```

```
        address[] calldata poolPath,
        uint256 amountIn,
        uint256 minimumamountOut,
        address to,
        uint256 deadline
    ) external override returns (uint256 amountOut, uint256 haircut) {
        require(tokenPath[tokenPath.length - 1] == address(wNative), 'the last address
            should be wrapped token');
        require(deadline >= block.timestamp, 'expired');//Luck:require(tokenPath.length
            >= 2
        require(poolPath.length == tokenPath.length - 1, 'invalid pool path');

        // get from token from users
        IERC20(tokenPath[0]).safeTransferFrom(address(msg.sender), address(this),
            amountIn);

        (amountOut, haircut) = _swap(tokenPath, poolPath, amountIn, address(this));
        require(amountOut >= minimumamountOut, 'amountOut too low');

        wNative.withdraw(amountOut);
        _safeTransferNative(to, amountOut);
    }

    /// @notice Private function to swap alone the token path
    /// @dev Assumes router has initial amountIn in balance.
    /// Assumes tokens being swapped have been approve via the approveSpendingByPool
        function
    /// @param tokenPath An array of token addresses. path.length must be >= 2.
    /// @param tokenPath The first element of the path is the input token, the last
        element is the output token.
    /// @param poolPath An array of pool addresses. The pools where the pathTokens are
        contained in order.
    /// @param amountIn the amount in
    /// @param to the user to send the tokens to
    /// @return amountOut received by user
    /// @return haircut total fee charged by pool
    function _swap(
        address[] calldata tokenPath,
        address[] calldata poolPath,
        uint256 amountIn,
        address to
    ) internal returns (uint256 amountOut, uint256 haircut) {
        // haircut of current call
        uint256 localHaircut;
        // next from amount, starts with amountIn in arg
        uint256 nextamountIn = amountIn;

        // first n - 1 swaps
        for (uint256 i; i < poolPath.length - 1; ++i) {
            // make the swap with the correct arguments
            (amountOut, localHaircut) = IPool(poolPath[i]).swap(
                tokenPath[i],
```

```
                tokenPath[i + 1],
                nextamountIn,
                0, // minimum amount received is ensured on calling function
                address(this),
                type(uint256).max // deadline is ensured on calling function);
            nextamountIn = amountOut;
            haircut += localHaircut;
        }

        // last swap
        uint256 i = poolPath.length - 1;
        (amountOut, localHaircut) = IPool(poolPath[i]).swap(
            tokenPath[i],
            tokenPath[i + 1],
            nextamountIn,
            0, // minimum amount received is ensured on calling function
            to,
            type(uint256).max // deadline is ensured on calling function);
        haircut += localHaircut;
    }
```

Listing 3.4: WombatRouter.sol

Recommendation Return a haircut array to record the haircut taken from each swap hop or simply remove the haircut return.

Status The issue has been fixed by this commit: `c6b2045`.

3.4 Improved receive() to Receive Native Assets

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WombatRouter`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `WombatRouter` contract introduces native assets wrapping and unwrapping functionalities for deposits, withdraws, and swaps within the router for users convenience, e.g. directly depositing and withdrawing in `BNB`, or swapping a liquid staking token, such as `stkBNB` to `BNB`.

Specifically, when a user deposits in `BNB`, the `WombatRouter` further deposits the `BNB` to `wNative` to get the wrapped `BNB`. When user withdraws in `BNB`, it withdraws the wrapped `BNB` from `wNative` and sends the `BNB` to user. Overall the `wNative` is the only place where the `WombatRouter` contract can receive `BNB` from. However, we notice the `WombatRouter` implements a `receive()` function (line 32) that can accept BNB from any address. Based on this, it is suggested to add proper validation for the BNB sender to accept BNB transfer only from the wNative.

```
contract WombatRouter is Ownable, IWombatRouter {
  using SafeERC20 for IERC20;

  IWNative public immutable wNative;

  constructor(IWNative _wNative) {
      wNative = _wNative;
  }

  receive() external payable {}
  ...
}
```

Listing 3.5: WombatRouter.sol

Recommendation Revise the receive() routine to accept BNB transfer only from the wNative.

Status The issue has been fixed by this commit: c6b2045.

3.5 Timely _updateReward() in addRewardToken()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MultiRewarderPerSec
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Wombat protocol, the MultiRewarderPerSec contract provides an incentive mechanism that rewards the staking of supported assets in MasterWombatV2. The rewards are carried out by adding reward token with a specific reward speed into the rewarder, and one rewarder can support multiple reward tokens. The staking users are rewarded in each reward token with the specified reward speed per their deposit amount in MasterWombatV2.

The reward token can be dynamically added via addRewardToken() by the owner. When analyzing the logic to add new reward token in the updateMultiplier() routine, we notice the need of timely invoking _updateReward() to update the lastRewardTimestamp before the new reward token gets effective.

```
function addRewardToken(IERC20 _rewardToken, uint96 _tokenPerSec) external onlyOwner {
  // use non-zero amount for accTokenPerShare as we want to check if user
  // has activated the pool by checking rewardDebt > 0
```

```
    RewardInfo memory reward = RewardInfo({
        rewardToken: _rewardToken,
        tokenPerSec: _tokenPerSec,
        accTokenPerShare: 1e18
    });
    rewardInfo.push(reward);
    emit RewardRateUpdated(address(_rewardToken), 0, _tokenPerSec);
}

/// @dev This function should be called before lpSupply and sumOfFactors update
function _updateReward() internal {
    uint256 length = rewardInfo.length;
    uint256 lpSupply = lpToken.balanceOf(address(masterWombat));

    if (block.timestamp > lastRewardTimestamp && lpSupply > 0) {
        for (uint256 i; i < length; ++i) {
            RewardInfo storage reward = rewardInfo[i];
            uint256 timeElapsed = block.timestamp - lastRewardTimestamp;
            uint256 tokenReward = timeElapsed * reward.tokenPerSec;
            reward.accTokenPerShare += toUint128((tokenReward * ACC_TOKEN_PRECISION) /
                lpSupply);
        }

        lastRewardTimestamp = block.timestamp;
    }
}
```

Listing 3.6: `MultiRewarderPerSec::addRewardToken()`

If the call to `_updateReward()` is not immediately invoked before the new reward token gets effective, the reward in the new reward token will be accumulated from the old `lastRewardTimestamp` which is the time when the `_updateReward()` is last invoked. As a result, staking users will get more rewards in the new reward token than expected.

Recommendation Timely invoke `_updateReward()` in the `addRewardToken()` routine.

Status The issue has been fixed by this commit: `e9543e0`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Wombat` v2 protocol which is introduced on top of the `Wombat` v1 with innovations to cater to liquid staking tokens, i.e. dynamic pool, and more experimental stablecoins, i.e. sidepool. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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