SMART CONTRACT AUDIT REPORT

for

Quoll V2

Prepared By: Xiaomi Huang

PeckShield

January 6, 2023

Document Properties

Client	Quoll Finance
Title	Smart Contract Audit Report
Target	Quoll V2
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	January 6, 2023	Luck Hu	Final Release
1.0-rc	January 3, 2023	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 Quoll V2 protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Quoll V2

Quoll Finance is a multichain veToken aggregator and yield booster powered by Wombat Exchange and BNB Chain. Quoll V2 introduces the bribe feature which provides the functionality for Quoll users to vote on how WOM shall be distributed in Wombat. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Quoll V2

Item	Description
Target	Quoll V2
Website	https://quoll.finance/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 6, 2023

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

- https://github.com/quollfi/quoll-contracts.git (a67d89e)

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

- https://github.com/quollfi/quoll-contracts.git (23813a5)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Quoll V2` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	3	■■■
Low	2	■■
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

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

Table 2.1: Key Quoll V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Overflow Prevention in vote()	Coding Practices	Fixed
PVE-002	Medium	Improved Handling of Caller Fees in _resetVotes()	Business Logic	Fixed
PVE-003	Low	Improved Support of Native Token as Bribe Reward	Business Logic	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Improved Overflow/Underflow Prevention in vote()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BribeManager/SmartConverter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Quoll V2` protocol implements the bribe feature, which enables `Quoll` users to vote on how to distribute the voting powers among all the supported `MasterWombat` pools. The user can get the voting power by locking `QUO` in the `VlQuoV2` contract. While reviewing the implementation of the voting functionality, we notice the contract is built on solidity 0.6.12 and there is potential overflow/underflow risk for the arithmetic operations.

In the following, we take the `BribeManager::vote()` routine as an example and show the potential overflow/underflow risk. At the beginning of the routine, it adds all the delta votes into the `totalUserVote` (line 245). The addition is not guarded against possible overflow. In particular, if some of the delta votes are oversized, the addition of `delta` to `totalUserVote` may overflow to yield a much smaller negative integer. Similarly, the subtraction of the `totalUserVote` from the `totalVlQuoInVote` (line 281) is not guarded again possible underflow. If the `totalUserVote` is undersized, the subtraction may underflow to yield a much bigger integer.

Note the same issue is also applicable to the `unvote()/SmartConverter::_depositFor()` routines, etc.

```
235     function vote(address[] calldata _lps, int256[] calldata _deltas)
236     external
237     override
238 {
239     uint256 length = _lps.length;
240     int256 totalUserVote;
```

```
    for (uint256 i; i < length; i++) {
        Pool memory pool = poolInfos[_lps[i]];
        require(pool.isActive, "Not active");
        int256 delta = _deltas[i];
        totalUserVote += delta;
        if (delta != 0) {
            if (delta > 0) {
                poolTotalVote[pool.lpToken] += uint256(delta);
                userTotalVote[msg.sender] += uint256(delta);
                userVoteForPools[msg.sender][pool.lpToken] += uint256(
                    delta
                );
                IVirtualBalanceRewardPool(pool.rewarder).stakeFor(
                    msg.sender,
                    uint256(delta)
                );
            } else {
                poolTotalVote[pool.lpToken] -= uint256(-delta);
                userTotalVote[msg.sender] -= uint256(-delta);
                userVoteForPools[msg.sender][pool.lpToken] -= uint256(
                    -delta
                );
                IVirtualBalanceRewardPool(pool.rewarder).withdrawFor(
                    msg.sender,
                    uint256(-delta)
                );
            }

            emit VoteUpdated(
                msg.sender,
                pool.lpToken,
                userVoteForPools[msg.sender][pool.lpToken]
            );
        }
    }
    if (msg.sender != delegatePool) {
        // this already gets updated when a user vote for the delegate pool
        if (totalUserVote > 0) {
            totalVlQuoInVote += uint256(totalUserVote);
        } else {
            totalVlQuoInVote -= uint256(-totalUserVote);
        }
    }
    require(
        userTotalVote[msg.sender] <= getUserLocked(msg.sender),
        "Above vote limit"
    );
}
```

Listing 3.1: BribeManager::vote()

Recommendation Use a higher solidity version (>0.8.0) which integrates auto `safeMath` check or add proper overflow/underflow prevention like the safeMath.

Status The issue has been fixed in this commit: 23813a5.

3.2 Improved Handling of Caller Fees in _resetVotes()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: BribeManager
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the WombatVoterProxy contract, in order to incentivize the caller to cast the pending votes to Wombat, it rewards the caller with the caller fees from the received bribe rewards. The caller fees are sent to the BribeManager contract which further forwards them to the caller. In particular, if no caller is specified in the WombatVoterProxy::vote() routine, no caller fee is rewarded. While reviewing the logic to remove all votes from all pools, we notice the caller is specified but the caller fees are not handled.

To elaborate, we show below the code snippet of the _resetVotes() routine, which is used by the owner to remove all votes from all the supported pools. It specifies the caller as the owner itself. However, after invoking the voterProxy.vote() routine (line 198), it does not properly handle the received caller fees which shall be forwarded to the caller (the owner in this case). As a result, the caller fees are locked in the contract. Based on this, it's suggested to not specify the caller or properly transfer the received caller fees to the caller.

```
187     function _resetVotes() internal {
188         uint256 length = pools.length;
189         address[] memory lpVote = new address[](length);
190         int256[] memory votes = new int256[](length);
191         address[] memory rewarders = new address[](length);
192         for (uint256 i; i < length; i++) {
193             Pool memory pool = poolInfos[pools[i]];
194             lpVote[i] = pool.lpToken;
195             votes[i] = -int256(getVeWomVoteForLp(pool.lpToken));
196             rewarders[i] = pool.rewarder;
197         }
198         voterProxy.vote(lpVote, votes, rewarders, owner());
199         emit AllVoteReset();
200     }
```

Listing 3.2: BribeManager::_resetVotes()

Recommendation Revisit the above `_resetVotes()` routine to not specify the caller or properly transfer the caller fees to the specified caller.

Status The issue has been fixed in this commit: `23813a5`.

3.3 Improved Support of Native Token as Bribe Reward

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Quoll V2` protocol, it introduces the new bribe feature which collects users pending votes and casts the votes to `Wombat` to earn the bribe rewards. The earned bribe rewards are further added to the reward pools for users to claim. While reviewing the handling of the bribe rewards, we notice current implementation doesn't take the support of the native token, i.e., `BNB`, into consideration.

To elaborate, we show below the code snippet of the `vote()` routine, which is used to cast users pending votes to `Wombat` and handle the earned bribe rewards. It loops each of the reward tokens, takes the protocol fee (line 263) and the caller fee (line 273), and adds the remaining rewards to the reward pool (line 290). In particular, `Wombat` supports the native token (`address(0)`) as the bribe reward. However, it does not properly handle the native token in the `vote()` routine and takes all the reward tokens as normal ERC20 tokens. As a result, if the native token is received, it may revert the `vote()` routine due to the invoking of ERC20 function to `address(0)` (lines 267, 276).

```
236     function vote(
237         address[] calldata _lpVote,
238         int256[] calldata _deltas,
239         address[] calldata _rewarders,
240         address _caller
241     )
242         external
243         override
244         returns (address[][] memory rewardTokens, uint256[][] memory feeAmounts)
245     {
246         require(msg.sender == bribeManager, "!auth");
247         uint256 length = _lpVote.length;
248         require(length == _rewarders.length, "Not good rewarder length");
249         uint256[][] memory bribeRewards = voter.vote(_lpVote, _deltas);
250 
251         rewardTokens = new address[][](length);
252         feeAmounts = new uint256[][](length);
```

```
        for (uint256 i = 0; i < length; i++) {
            uint256[] memory rewardAmounts = bribeRewards[i];
            (, , , , , , address bribesContract) = voter.infos(_lpVote[i]);
            feeAmounts[i] = new uint256[](rewardAmounts.length);
            if (bribesContract != address(0)) {
                rewardTokens[i] = IBribe(bribesContract).rewardTokens();
                for (uint256 j = 0; j < rewardAmounts.length; j++) {
                    uint256 rewardAmount = rewardAmounts[j];
                    if (rewardAmount > 0) {
                        uint256 protocolFee = rewardAmount
                            .mul(bribeProtocolFee)
                            .div(FEE_DENOMINATOR);
                        if (protocolFee > 0) {
                            IERC20(rewardTokens[i][j]).safeTransfer(
                                bribeFeeCollector,
                                protocolFee
                            );
                        }
                        if (_caller != address(0) && bribeCallerFee != 0) {
                            uint256 feeAmount = rewardAmount
                                .mul(bribeCallerFee)
                                .div(FEE_DENOMINATOR);
                            IERC20(rewardTokens[i][j]).safeTransfer(
                                bribeManager,
                                feeAmount
                            );
                            rewardAmount -= feeAmount;
                            feeAmounts[i][j] = feeAmount;
                        }
                        rewardAmount -= protocolFee;
                        _approveTokenIfNeeded(
                            rewardTokens[i][j],
                            _rewarders[i],
                            rewardAmount
                        );
                        IVirtualBalanceRewardPool(_rewarders[i])
                            .queueNewRewards(rewardTokens[i][j], rewardAmount);
                    }
                }
            }
        }

        emit Voted(_lpVote, _deltas, _rewarders, _caller);
    }
```

Listing 3.3: `WombatVoterProxy::vote()`

Note the same issue is applicable to the `BribeManager::previewNativeAmountForCast()/_forwardRewards()/_swapFeesForNative()` routines, etc.

Similarly, in the reward pool contracts of the `Quoll V2` protocol, e.g., `BaseRewardPool/DelegateVotePool`

, they also support the native token as one of the reward tokens. However, in the `getReward()/harvest()` routines, it doesn't properly handle the native token.

Recommendation Revisit the logic of the rewards handling to support the native token as one of the rewards.

Status The issue has been fixed in this commit: `23813a5`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NativeZapper`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

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

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transferFrom()`, there is a check, i.e., `if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism."*

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

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

```
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

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

In the following, we show the `zapInToken()` routine in the `NativeZapper` contract. If the `ZRX` token is supported as `_from`, the unsafe version of `IERC20(_from).transferFrom(msg.sender, address(this), _amount)` (line 64) may return false while not revert. As a result, the transaction can proceed even when the token transfer fails.

```
57    function zapInToken(
58        address _from,
59        uint256 _amount,
60        address _receiver
61    ) external override returns (uint256 nativeAmount) {
62        if (_amount > 0) {
63            _approveTokenIfNeeded(_from, _amount);
64            IERC20(_from).transferFrom(msg.sender, address(this), _amount);
65            nativeAmount = _swapTokenForNative(_from, _amount, _receiver);
66
67            emit ZapIn(_from, _amount, _receiver, nativeAmount);
68        }
69    }
```

Listing 3.5: `NativeZapper::zapInToken()`

Another similar violation can be found in the `VIQuoV2::unlock` routine.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()/transferFrom()`.

Status The issue has been fixed in this commit: `23813a5`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Quoll V2` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., set the protocol fee). In the following, we take the `DelegateVotePool` contract as an example, and show the representative functions potentially affected by the privilege of the `owner` account.

Specifically, the privileged functions in the `DelegateVotePool` contract allow for the owner to set the protocol fee for vote delegation, set the fee receiver, and delete a pool, etc.

```
    function setProtocolFee(uint256 _protocolFee) external onlyOwner {
        require(_protocolFee < DENOMINATOR, "invalid _protocolFee!");
        protocolFee = _protocolFee;
    }

    function setFeeCollector(address _feeCollector) external onlyOwner {
        require(_feeCollector != address(0), "invalid _feeCollector!");
        feeCollector = _feeCollector;
    }

    function deletePool(address _lp) external onlyOwner {
        require(isVotePool[_lp], "invalid _lp!");
        require(
            !IBribeManager(bribeManager).isPoolActive(_lp),
            "Pool still active"
        );

        isVotePool[_lp] = false;
        uint256 length = votePools.length;
        address[] memory newVotePool = new address[](length - 1);
        uint256 indexShift;
        for (uint256 i; i < length; i++) {
            if (votePools[i] == _lp) {
                indexShift = 1;
            } else {
                newVotePool[i - indexShift] = votePools[i];
            }
        }
        votePools = newVotePool;
        totalWeight = totalWeight - votingWeights[_lp];
        votingWeights[_lp] = 0;
```

```
        if (_getVoteForLp(_lp) > 0) {
            IBribeManager(bribeManager).unvote(_lp);
        }
        _updateVote();
    }
```

Listing 3.6: Example Privileged Operations in the DelegateVotePool Contract

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the owner is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 confirmed by the team. The team intends to introduce multi-sig mechanism to mitigate this issue.

4 | Conclusion

In this audit, we have analyzed the design and implementation of `Quoll V2`. `Quoll Finance` is a multichain `veToken` aggregator and yield booster powered by `Wombat Exchange` and `BNB Chain`. `Quoll V2` introduces the bribe feature which provides the functionality for `Quoll` users to vote on how `WOM` shall be distributed in `Wombat`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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