SMART CONTRACT AUDIT REPORT

for

Wombex

Prepared By: Xiaomi Huang

PeckShield
October 11, 2022

Document Properties

Client	Wombex Finance
Title	Smart Contract Audit Report
Target	Wombex
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	October 11, 2022	Luck Hu	Final Release
1.0-rc	September 27, 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 **4**

1.1 About Wombex 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Improved _index Scope Validation in forfeitRewards() 11

3.2 Accommodation of Non-ERC20-Compliant Tokens 12

3.3 Revisited Logic to Release Custom Lock 14

3.4 Proper Pool Shutdown in shutdownSystem() 16

3.5 Trust Issue of Admin Keys 18

4 Conclusion **21**

References **22**

1 Introduction

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

Wombex is a BSC native protocol for boosting yield for liquidity providers and concentrating governance power across Wombat. It accumulates veWOM and aggregates LPs deposits simultaneously, in order to combine the power of liquidity providers and WOM token holders, supercharging each other and accelerating long-term Wombat growth. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Wombex

Item	Description
Name	Wombex Finance
Website	https://wombex.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 11, 2022

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

- https://github.com/wombex-finance/wombex-contracts.git (eb94985)

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

- https://github.com/wombex-finance/wombex-contracts.git (3bac709)

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.

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

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

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 `Wombex` 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 Wombex Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved _index Scope Validation in forfeitRewards()	Coding Practices	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Medium	Revisited Logic to Release Custom Lock	Business Logic	Fixed
PVE-004	Low	Proper Pool Shutdown in shutdownSystem()	Business Logic	Fixed
PVE-005	Medium	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 Improved _index Scope Validation in forfeitRewards()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ExtraRewardsDistributor
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

In Wombex protocol, the ExtraRewardsDistributor contract is introduced to allow anyone to distribute rewards to the holders of WmxLocker at a given epoch. The WmxLocker holder can claim rewards from a specific index of the reward Epochs. Specially, it allows the holders to set their claimed index forward without claiming rewards via the forfeitRewards() routine.

To elaborate, we show below the code snippet of the forfeitRewards() routine. It accepts the input epoch index to forfeit from and validates the index in the valid scope (line 180). The valid index scope shall be [0, rewardEpochs[_token].length - 1]. However, current validation doesn't take the boundaries 0 and rewardEpochs[_token].length - 1] into the valid scope. As a result, user cannot forfeit the rewards for the first epoch only or all the epochs.

```
179     function forfeitRewards(address _token, uint256 _index) external {
180         require(_index > 0 && _index < rewardEpochs[_token].length - 1, "!past");//Luck1
                : >= 0 && _index < rewardEpochs[_token].length - 1
181         require(_index >= userClaims[_token][msg.sender], "already claimed");

183         //set claim checkpoint. next claim starts from index+1
184         userClaims[_token][msg.sender] = _index + 1;

186         emit RewardForfeited(msg.sender, _token, _index);
187     }
```

Listing 3.1: ExtraRewardsDistributor :: forfeitRewards ()

Recommendation Add the boundaries 0 and `rewardEpochs[_token].length - 1]` into the valid scope of the reward epoch index.

Status The issue has been fixed by this commit: `cfe2854`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Booster`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

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

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

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

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

```
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

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

In the following, we show the `_earmarkRewards()` routine in the `Booster` contract. If the `ZRX` token is supported as `token`, the unsafe version of `token.transfer(tDistro.distro, amount)` (line 550) may return `false` while not revert. Without a validation on the return value, the transaction can proceed even when the transfer fails.

```
520     function _earmarkRewards(uint256 _pid) internal {
521         PoolInfo storage pool = poolInfo[_pid];
522         require(pool.shutdown == false, "pool is closed");
523
524         address gauge = pool.gauge;
525         //claim crv/wom and bonus tokens
526         (address[] memory tokens, uint256[] memory balances) = IStaker(voterProxy).
            claimCrv(gauge, _pid);
527
528         uint256 tLen = tokens.length;
529         for (uint256 i = 0; i < tLen; i++) {
530             IERC20 token = IERC20(tokens[i]);
531             uint256 balance = balances[i];
532
533             emit EarmarkRewards(address(token), balance);
534
535             if (balance == 0) {
536                 continue;
537             }
538             uint256 dLen = distributionByTokens[address(token)].length;
539             require(dLen > 0, "!dLen");
540
541             uint256 earmarkIncentiveAmount = balance.mul(earmarkIncentive).div(
                DENOMINATOR);
542             uint256 sentSum = earmarkIncentiveAmount;
543
544             for (uint256 j = 0; j < dLen; j++) {
545                 TokenDistro memory tDistro = distributionByTokens[address(token)][j];
546                 uint256 amount = balance.mul(tDistro.share).div(DENOMINATOR);
547                 if (tDistro.callQueue) {
548                     IRewards(tDistro.distro).queueNewRewards(address(token), amount);
549                 } else {
550                     token.transfer(tDistro.distro, amount);
551                 }
```

```
                sentSum = sentSum.add(amount);
            }
            if (earmarkIncentiveAmount > 0) {
                token.safeTransfer(msg.sender, earmarkIncentiveAmount);
            }
            //send crv to lp provider reward contract
            IRewards(pool.crvRewards).queueNewRewards(address(token), balance.sub(
                sentSum));
        }
    }
```

Listing 3.3: `Booster::_earmarkRewards()`

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

Status The issue has been fixed by this commit: `2739343`.

3.3 Revisited Logic to Release Custom Lock

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WomDepositor`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `Wombex` protocol, the `WomDepositor` contract provides a pair of interfaces, i.e., `depositCustomLock()`/`releaseCustomLock()` routines, which are used to deposit/release `WOM` to/from `veWom` via the `VoterProxy` contract. In this way, `Wombex` accumulates `veWOM` to participate in `Wombat` governance.

To elaborate, we show below the code snippets of the `_smartLock()`/`releaseCustomLock()` routines. As the name indicates, the `releaseCustomLock()` routine is used to release a custom slot locked in the `veWom`. The slot number is given by the input `_index`. After the slot is released, it sets the `lockedCustomSlots[slot.number]` to `false` (line 182) which indicates this is not a custom slot any more. But the slot is not removed from the `slotEnds[]` which means this is still a valid slot. What's more, `lockedCustomSlots[slot.number] = false` implies this a normal slot which will be tried to release again in the `_smartLock()` routine (line 130-133). Because the slot actually has been unlocked from `veWom`, so when it tries to unlock it again in the `_smartLock()`, it will revert. Hence the following deposit of `WOM` will be blocked. Based on this, the released slot shall be properly removed from the `slotEnds[]`.

```
    function releaseCustomLock(uint256 _index) public {
        SlotInfo memory slot = customLockSlots[msg.sender][_index];

        require(slotEnds[slot.number] < block.timestamp, "!ends");

        IStaker(staker).releaseLock(slot.number);
        IERC20(wom).safeTransfer(msg.sender, slot.amount);

        lockedCustomSlots[slot.number] = false;

        uint256 len = customLockSlots[msg.sender].length;
        if (_index != len - 1) {
            customLockSlots[msg.sender][_index] = customLockSlots[msg.sender][len - 1];
        }
        customLockSlots[msg.sender].pop();

        currentSlot = currentSlot.sub(1);

        emit ReleaseCustomLock(msg.sender, _index, slot.number, slot.amount);
    }
```

Listing 3.4: WomDepositor::releaseCustomLock()

```
    function _smartLock(uint256 _amount) internal {
        IERC20(wom).transferFrom(msg.sender, address(this), _amount);

        if (lastLockAt + smartLockPeriod > block.timestamp && customLockDays[msg.sender]
            == 0) {
            return;
        }

        bool releaseExecuted = false;
        if (slotEnds[checkOldSlot] != 0 && slotEnds[checkOldSlot] < block.timestamp) {
            if (!lockedCustomSlots[checkOldSlot]) {
                IStaker(staker).releaseLock(checkOldSlot);
                releaseExecuted = true;
                slotEnds[checkOldSlot] = slotEnds[currentSlot - 1];
            }
            checkOldSlot++;
        }

        uint256 slot = currentSlot;
        if (releaseExecuted) {
            slot = slot.sub(1);
        } else {
            currentSlot = currentSlot.add(1);
        }
        if (currentSlot > 1 && checkOldSlot >= currentSlot - 1) {
            checkOldSlot = 0;
        }

        uint256 senderLockDays = lockDays;
        uint256 amountToLock = _amount;
```

```
        if (customLockDays[msg.sender] > 0) {
            senderLockDays = customLockDays[msg.sender];
            customLockSlots[msg.sender].push(SlotInfo(slot, _amount));
            lockedCustomSlots[slot] = true;
        } else {
            amountToLock = IERC20(wom).balanceOf(address(this));
        }

        IERC20(wom).safeTransfer(staker, amountToLock);
        IStaker(staker).lock(senderLockDays);

        slotEnds[slot] = block.timestamp + senderLockDays * 86400;

        lastLockAt = block.timestamp;

        emit SmartLock(msg.sender, customLockDays[msg.sender] > 0, slot, amountToLock,
            senderLockDays, currentSlot, checkOldSlot, releaseExecuted);
    }
```

Listing 3.5: WomDepositor::_smartLock()

What's more, after the custom slot is released in the `releaseCustomLock()` routine, it reduces the `currentSlot` by 1, which moves the `currentSlot` pointing to the previous existing slot. As a result, a new deposit will overwrite the previous slot.

Recommendation Revisit the `releaseCustomLock()` routine to properly clear a released custom lock.

Status The issue has been fixed by this commit: `9c420a4`.

3.4 Proper Pool Shutdown in shutdownSystem()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Booster`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

In `Wombex` protocol, the `Booster` contract is the main deposit contract which keeps track of the pools information, user deposits and distributes rewards. The contract provides interfaces, i.e., `shutdownPool()/shutdownSystem()`, for the `poolManager/owner` to shut down a pool or shut down the system.

To elaborate, we show below the code snippets of the `shutdownPool()/shutdownSystem()` routines. As the name indicates, the `shutdownSystem()` is used to shut down the whole system. It tries to withdraw all LPs from all the pools and marks them as shutdown. Specially, when the `withdrawAllLp()` throws (line 388), it closes the system without marking the pool as shutdown (line 389). However, in the `shutdownPool()` routine, when the `withdrawAllLp()` throws (line 363), it marks the pool as shutdown. We need to keep the handling of this case the same in both routines. Our analysis shows that, we need to mark the pool as shutdown no matter whether the `withdrawAllLp()` throws or not in the `shutdownSystem()` routine.

```
354     /**
355      * @notice Shuts down the pool by withdrawing everything from the gauge to here (can later be
356      *         claimed from depositors by using the withdraw fn) and marking it as shut down
357      */
358     function shutdownPool(uint256 _pid) external returns(bool){
359         require(msg.sender==poolManager, "!auth");
360         PoolInfo storage pool = poolInfo[_pid];
361 
362         //withdraw from gauge
363         try IStaker(voterProxy).withdrawAllLp(pool.lptoken,pool.gauge){
364         }catch{}
365 
366         pool.shutdown = true;
367 
368         emit PoolShutdown(_pid);
369         return true;
370     }
371 
372     /**
373      * @notice Shuts down the WHOLE SYSTEM by withdrawing all the LP tokens to here and then allowing
374      *         for subsequent withdrawal by any depositors.
375      */
376     function shutdownSystem() external{
377         require(msg.sender == owner, "!auth");
378         isShutdown = true;
379 
380         for(uint i=0; i < poolInfo.length; i++){
381             PoolInfo storage pool = poolInfo[i];
382             if (pool.shutdown) continue;
383 
384             address token = pool.lptoken;
385             address gauge = pool.gauge;
386 
387             //withdraw from gauge
388             try IStaker(voterProxy).withdrawAllLp(token,gauge){
389                 pool.shutdown = true;
390             }catch{}
391         }
392     }
```

Listing 3.6: Booster.sol

Recommendation Mark the pool as shutdown no matter whether the `withdrawAllLp()` throws or not in the `shutdownSystem()` routine.

Status The issue has been confirmed and improved by this commit: `2e72fa2`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In `Wombex` protocol, there are certain privileged accounts, i.e., `operator`, `owner`, `admin`, and `dao`, etc. that play critical roles in governing and regulating the system-wide operations.

Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `Wmx` contract as an example and show the representative functions potentially affected by the privileges of the `operator` account.

Specifically, the privileged functions in `Wmx` allow for the `operator` to mint initial `WMX` tokens, mint `WMX` to a given user based on the `WOM` supply schedule, and set the `minter`, etc. And the `minter` can mint `WMX` to the given address.

```
  function init(address _to, address _minter) external {
    require(msg.sender == operator, "Only operator");
    require(totalSupply() == 0, "Only once");
    require(_minter != address(0), "Invalid minter");

    _mint(_to, INIT_MINT_AMOUNT);
    updateOperator();
    minter = _minter;
    minterMinted = 0;

    emit Initialised();
 }

 /**
 * @dev Mints WMX to a given user based on the WOM supply schedule.
 */
function mint(address _to, uint256 _amount) external {
    require(totalSupply() != 0, "Not initialised");

    if (msg.sender != operator) {
        // dont error just return. if a shutdown happens, rewards on old system
        // can still be claimed, just wont mint wmx
```

```
            return;
        }

        // e.g. emissionsMinted = 6e25 - 5e25 - 0 = 1e25;
        uint256 emissionsMinted = totalSupply() - INIT_MINT_AMOUNT - minterMinted;
        // e.g. reductionPerCliff = 5e25 / 500 = 1e23
        // e.g. cliff = 1e25 / 1e23 = 100
        uint256 cliff = emissionsMinted.div(reductionPerCliff);

        // e.g. 100 < 500
        if (cliff < totalCliffs) {
            // e.g. (new) reduction = (500 - 100) * 2.5 + 700 = 1700;
            // e.g. (new) reduction = (500 - 250) * 2.5 + 700 = 1325;
            // e.g. (new) reduction = (500 - 400) * 2.5 + 700 = 950;
            uint256 reduction = totalCliffs.sub(cliff).mul(5).div(2).add(2);
            // e.g. (new) amount = 1e19 * 1700 / 500 =  34e18;
            // e.g. (new) amount = 1e19 * 1325 / 500 =  26.5e18;
            // e.g. (new) amount = 1e19 * 950 / 500  =  19e17;
            uint256 amount = _amount.mul(reduction).div(totalCliffs);
            // e.g. amtTillMax = 5e25 - 1e25 = 4e25
            uint256 amtTillMax = EMISSIONS_MAX_SUPPLY.sub(emissionsMinted);
            if (amount > amtTillMax) {
                amount = amtTillMax;
            }
            _mint(_to, amount);
        }
    }

    /**
     * @dev Allows minter to mint to a specific address
     */
    function minterMint(address _to, uint256 _amount) external {//Luck1: trust
        require(msg.sender == minter, "Only minter");
        minterMinted += _amount;
        _mint(_to, _amount);
    }
```

Listing 3.7: Example Privileged Operations in the Wmx Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the protocol users. It is worrisome if the privileged accounts 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 privileged accounts to a community-governed DAO.

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

Status This issue has been mitigated as the team confirms: After the protocol launch, the Wombex DAO multi-sig will be used to execute admin functions based on vlWMX holders voting results. In addition to the core team members, this multi-sig will include members of the Wombat team and solid representatives of the BNB chain ecosystem.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Wombex` protocol which is `BSC` native protocol for boosting yield for liquidity providers and concentrating govenance power across `Wombat`. It accumulates `veWOM` and aggregates `LPs` deposits simultaneously, in order to combine the power of liquidity providers and `WOM` token holders, supercharging each other and accelerating long-term `Wombat` growth. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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.