



SMART CONTRACT AUDIT



September 6th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

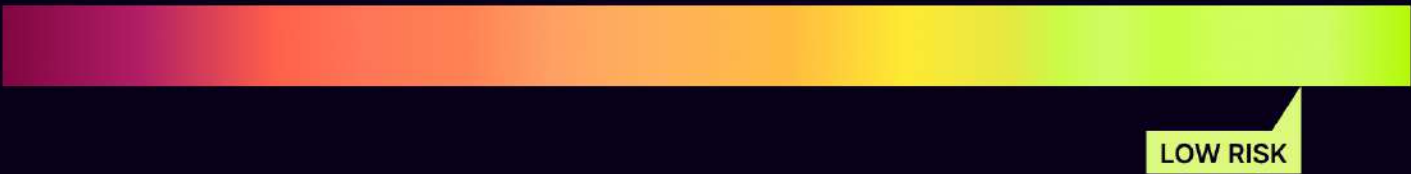


TECHNICAL SUMMARY

This document outlines the overall security of the Wombat smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Wombat smart contract codebase for quality, security, and correctness.

Contract Status



Testable Code



98% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network’s fast-paced and rapidly changing environment, we recommend that the Wombat team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

Auditing Strategy and Techniques Applied	3
Executive Summary	5
Protocol overview	6
Structure and Organization of the Document	27
Complete Analysis	28
Code Coverage and Test Results for all files written by Zokyo Security	35
Code Coverage and Test Results for all files written by the Wombat team	38

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Wombat repository:

Initial commit: b8862798f34edd681203f2fc75c99bd1f4527ed5

Final commit: bae29b2eaab0292300ee7bcc04c6a534935e8e55

Within the scope of this audit, the auditors reviewed the following contract(s):

- BribeRewarderFactory.sol
- BoostedMasterWombat.sol
- BoostedMultiRewarder.sol
- MultiRewarderPerSecV2.sol
- BribeV2.sol

During the audit, Zokyo Security ensured that the contract(s):

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Wombat smart contracts. To do so, our smart contract developers reviewed the code line by line and documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

- | | | | |
|----|---|----|--|
| 01 | Due diligence in assessing the overall code quality of the codebase. | 03 | Testing contract logic against common and uncommon attack vectors. |
| 02 | Cross-comparison with other, similar smart contracts by industry leaders. | 04 | Thorough manual review of the codebase line by line. |

Executive Summary

Zokyo Security received a set of governance contracts from the Wombat team, which included rewarders contracts and gauge contracts.

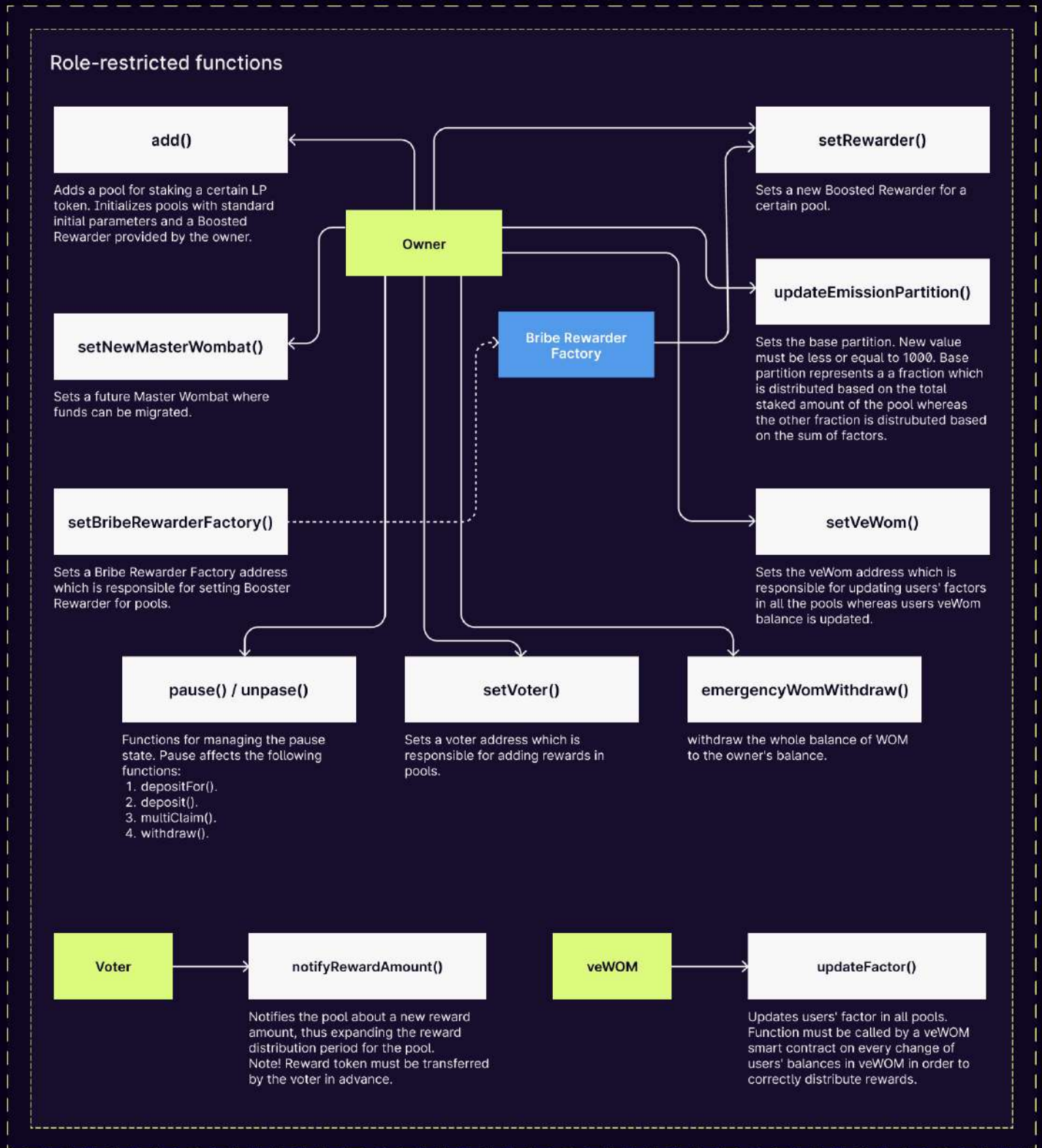
The audit's goal was to verify that the contracts functioned correctly and had a known level of security. The auditors checked the code line by line, compared it against their own checklist of vulnerabilities, validated the business logic of the contracts, and ensured that best practices in terms of gas spending were applied. The setup and deployment scripts of the contracts were also carefully audited.

During the manual part of the audit, one medium-severity issue and several informational-severity issues were found. The medium-severity issue was related to iteration for the entire storage array. The Wombat team resolved most of them, while others were discussed and verified.

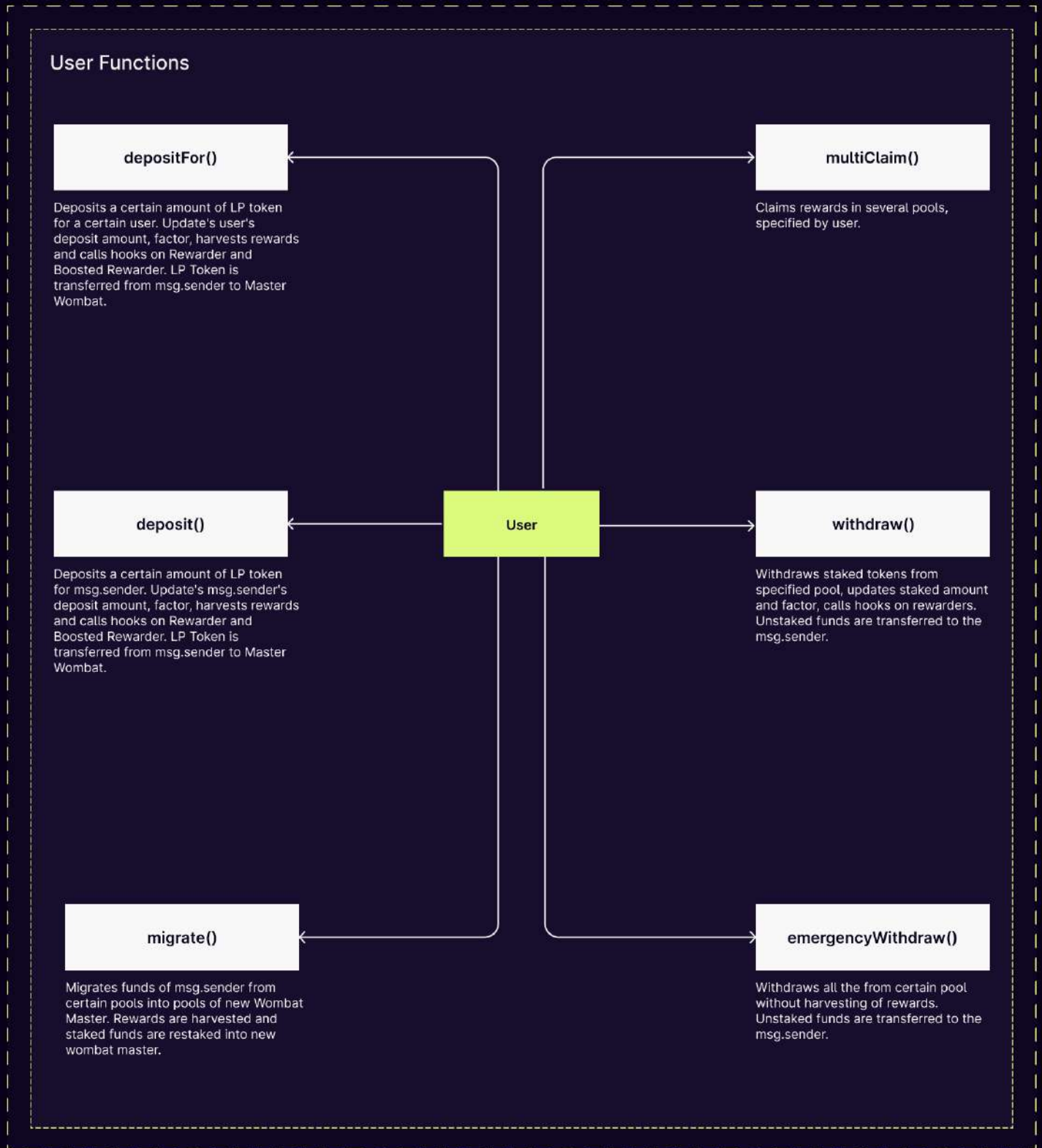
Another part of the audit process involved checking the native tests prepared by the Wombat team and preparing a set of custom test scenarios. It should be noted that the Wombat team provided a solid set of tests covering all the contracts' core logic. Nevertheless, Zokyo Security prepared its own set of unit tests and additional scenarios to cover the complex functionality of Wombat contracts. The complete set of unit tests can be seen in the Code Coverage and Test Result sections.

Overall, the security level of the contracts is high. The contracts are well-written and tested, with a boosted version of MasterWombat, rewarders, and gauge contracts being carefully audited. Wombat also provided detailed documentation on implemented functionalities, which can be accessed at: <https://docs.wombat.exchange/docs/>.

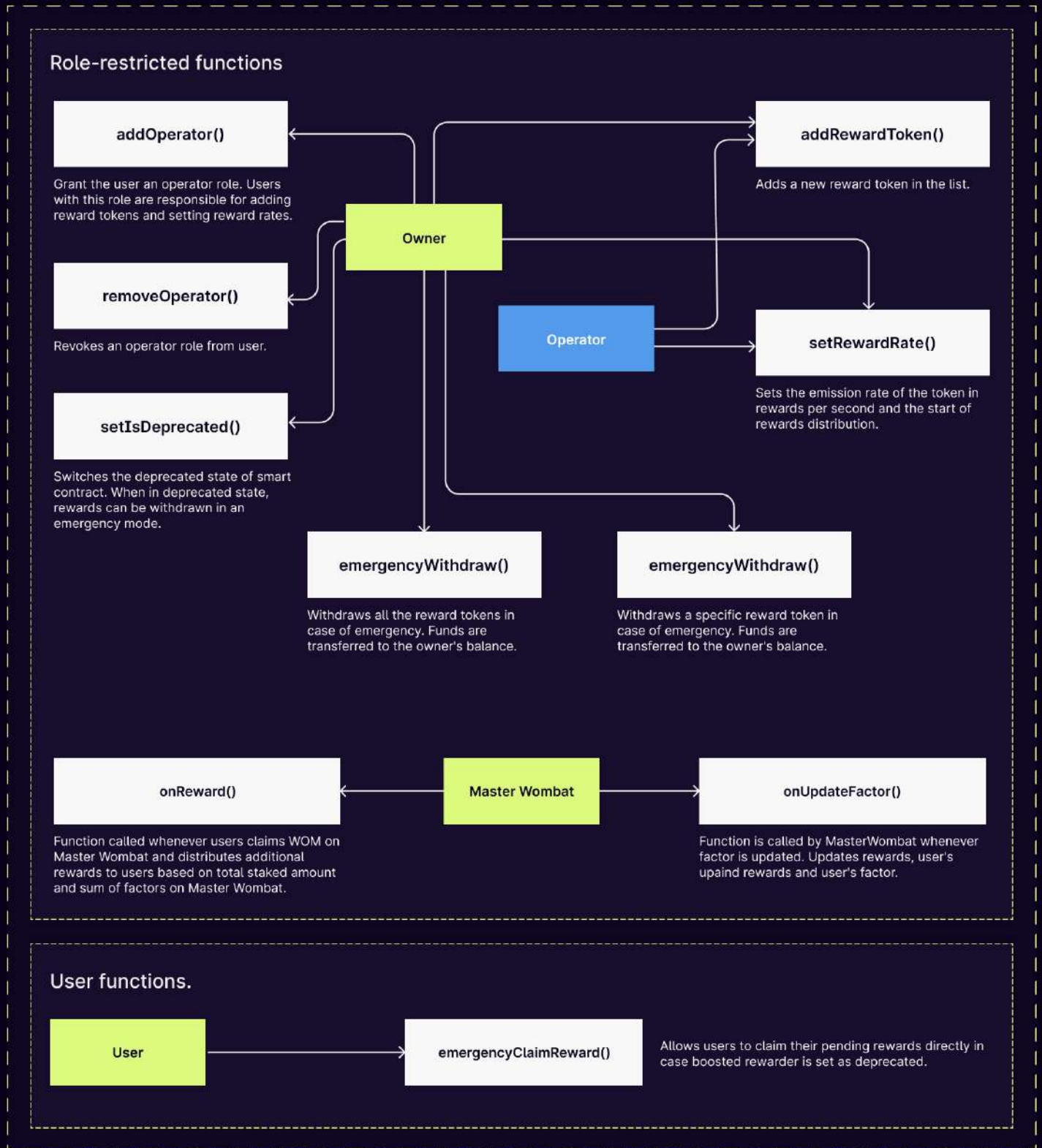
BoostedMasterWombat.sol



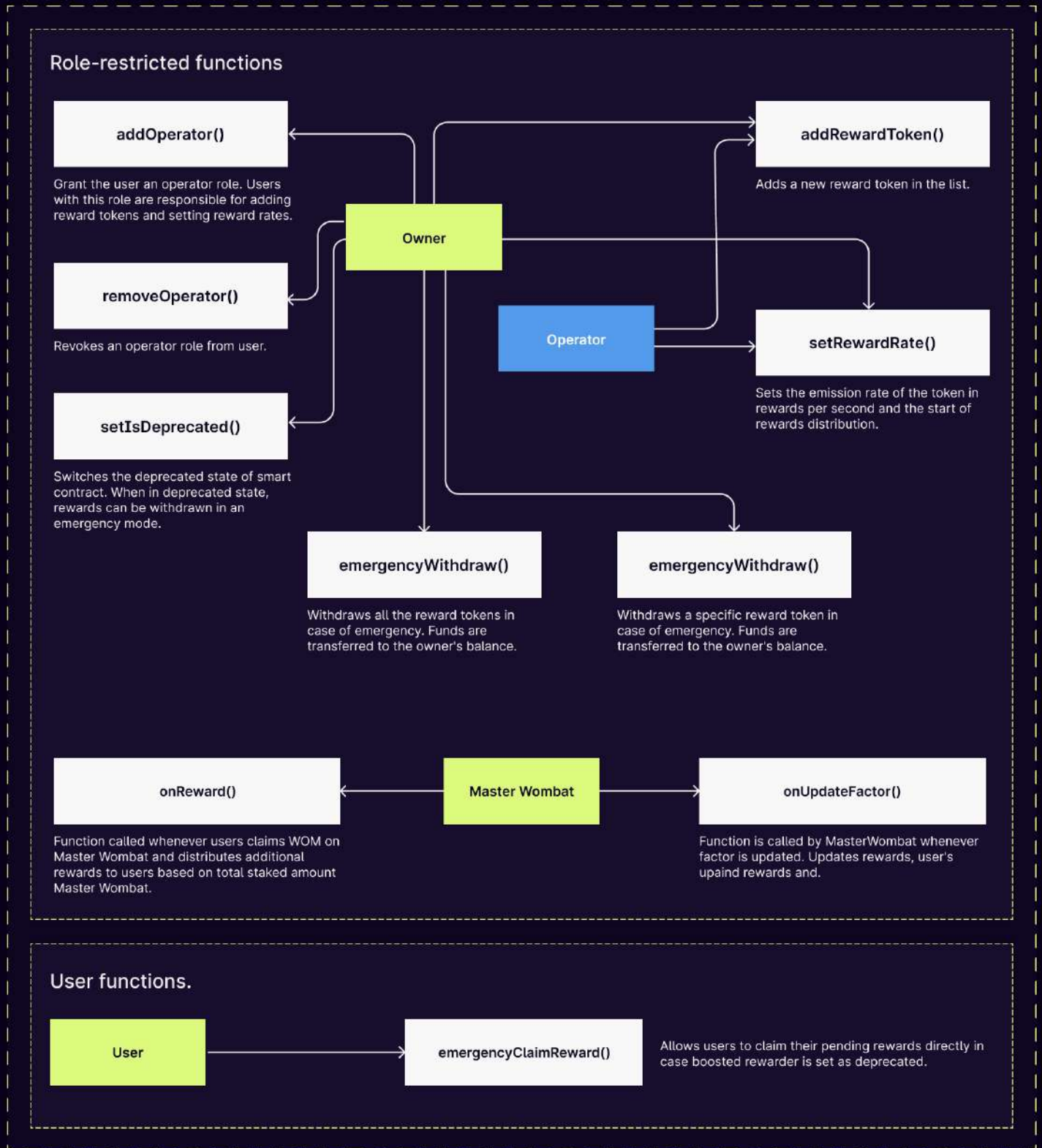
BoostedMasterWombat.sol



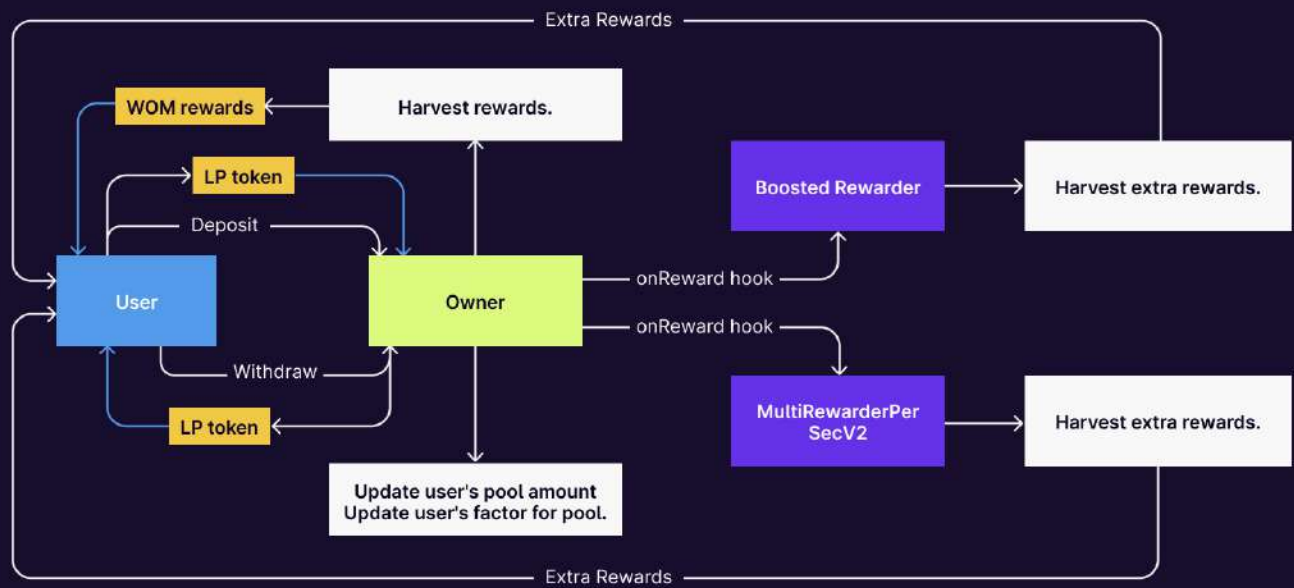
BoostedMultiRewarder.sol



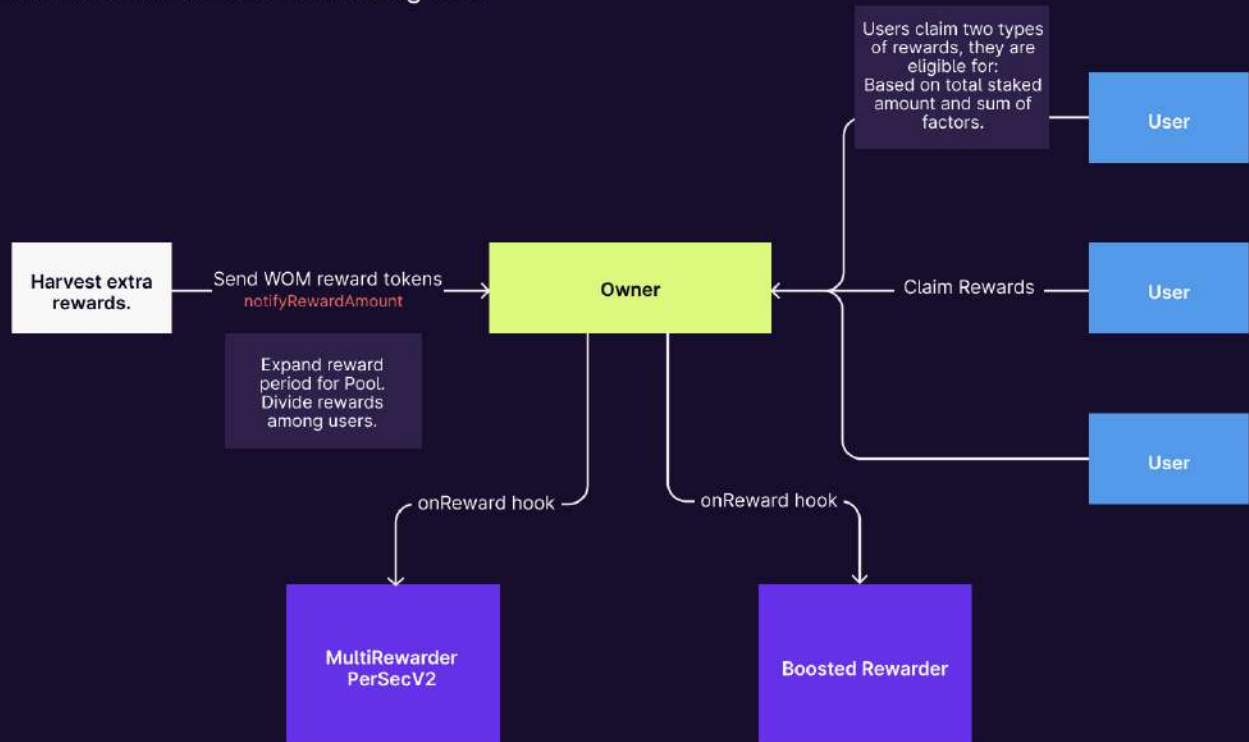
MultiRewarderPerSecV2.sol



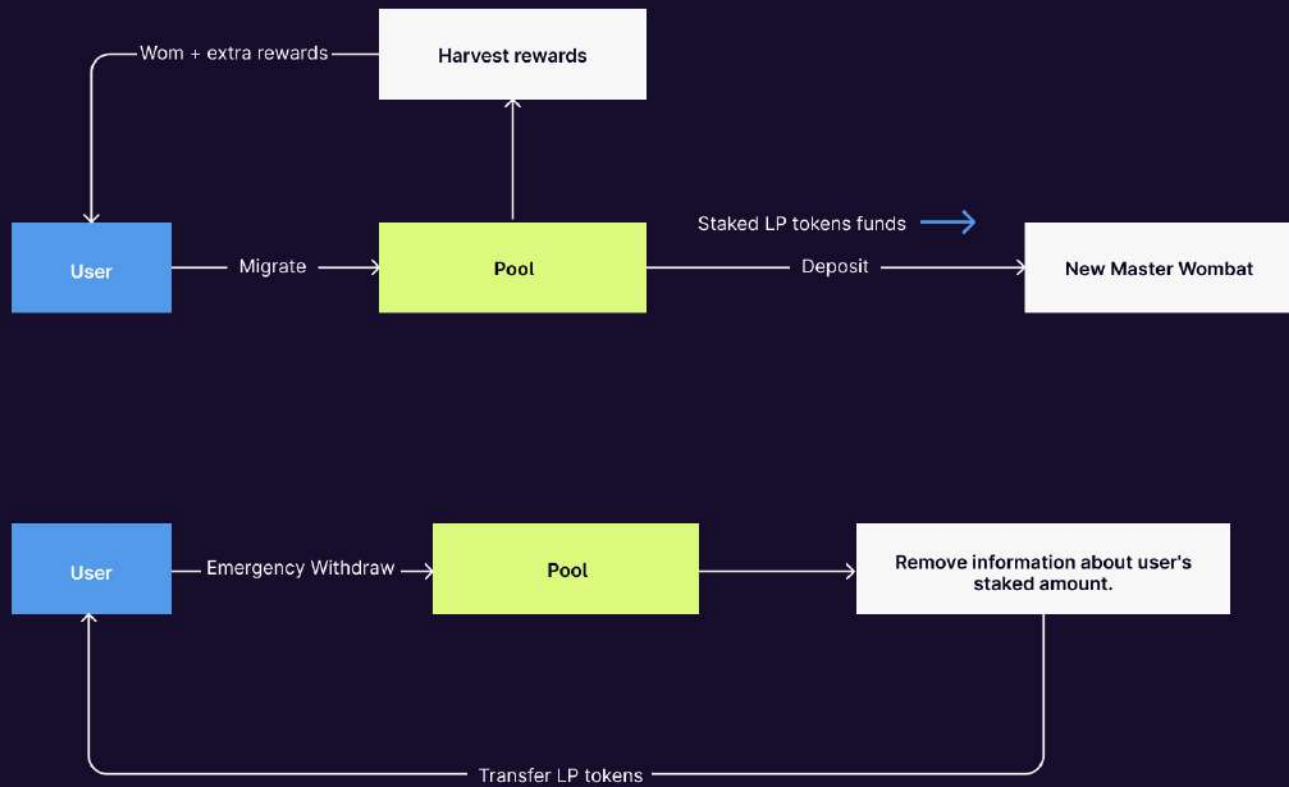
Deposit/Withdraw flow



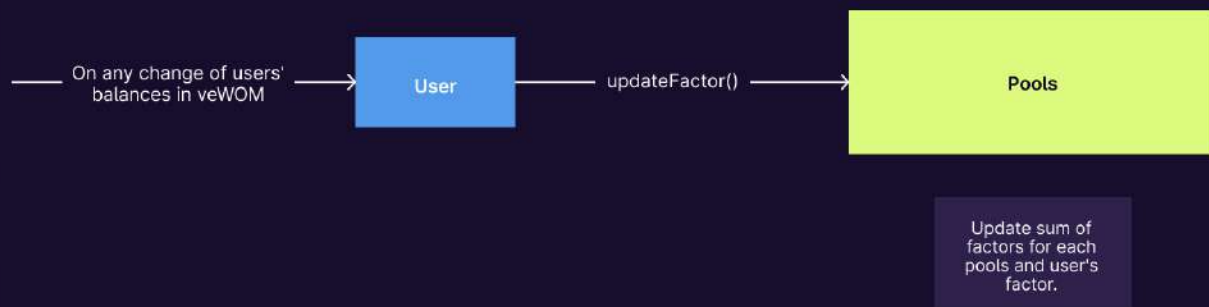
Rewards distribution and claiming flow



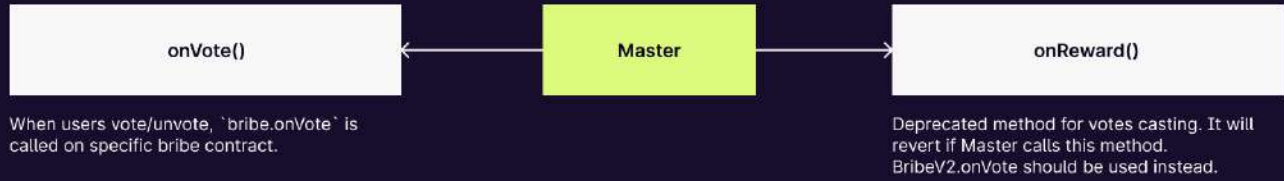
Migration/emergency withdraw flow.



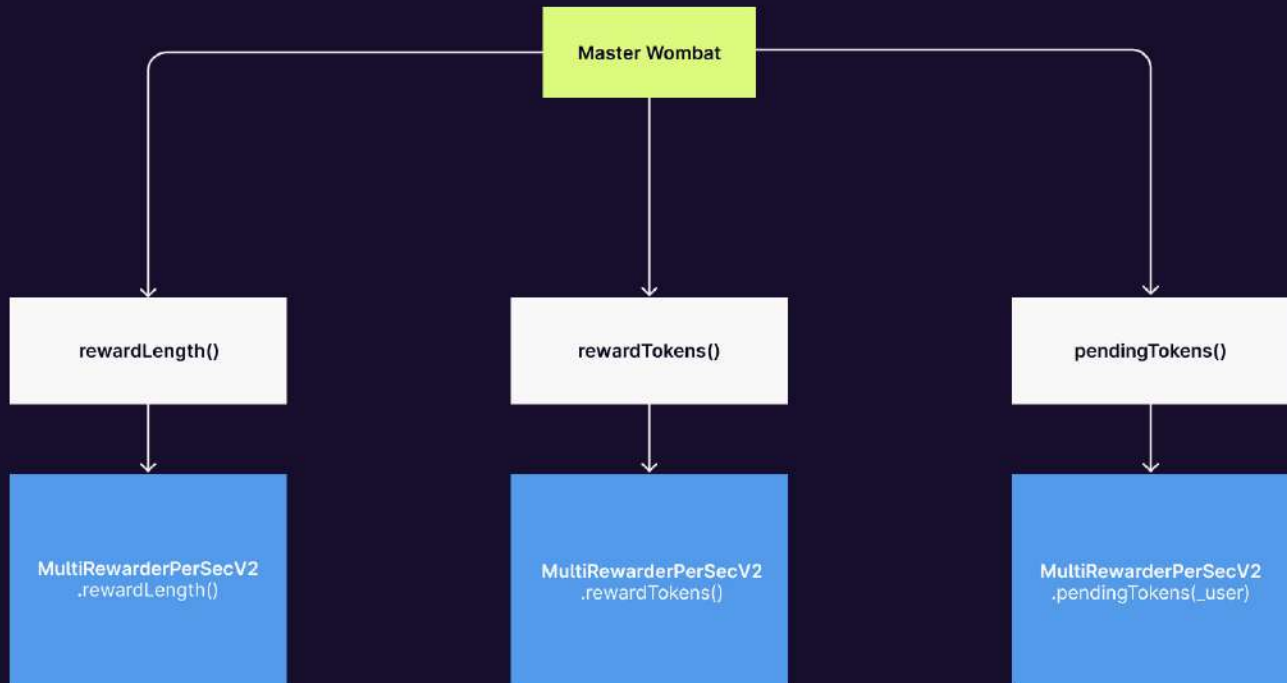
Update factor flow.



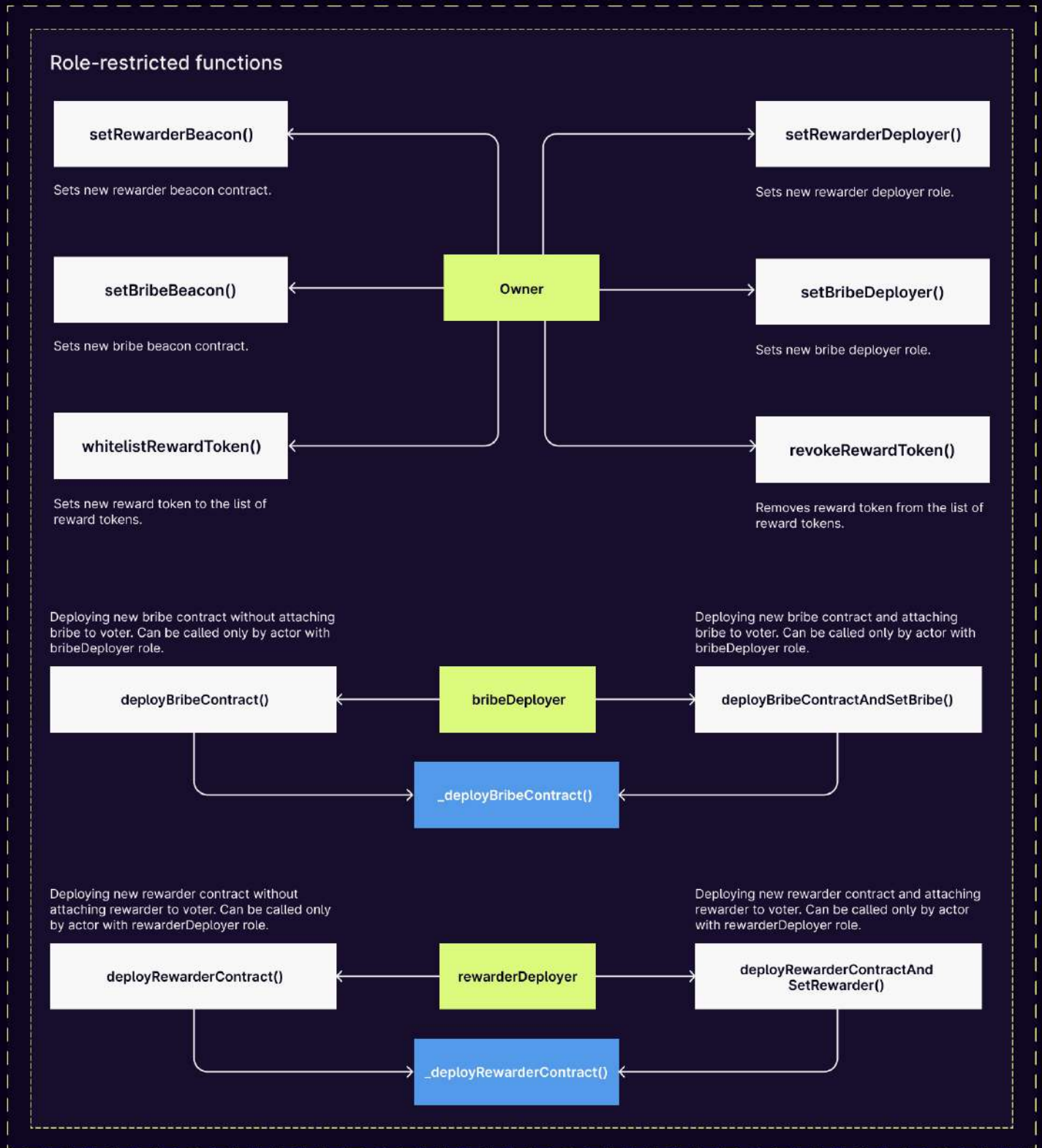
Role-restricted functions



User functions



BribeRewarderFactory.sol



WOMBAT DESCRIPTION

Roles and Responsibilities

BoostedMasterWombat.sol

1. Deployer

Deployer is in charge of deploying the implementation and proxy of the BoostedMasterWombat. He ought to provide valid parameters which pass all the validations inside the initialize() function to initialize the smart contract properly. Deployer has to pass a valid non-zero WOM token address, veWOM and Voter addresses which can initially be set as zero and set later, base partition which must be less or equal to 1000. During the deployment, deployer is assigned with owner role.

2. Owner

Owner role is a single address which is in charge of setting all the essential settings and also calling specific functions. This role is managed by OwnableUpgradable by OpenZeppelin. The following functions can be called only by owner:

- pause()
- unpause()
- setNewMasterWombat()
- setBribeRewarderFactory()
- add()
- setRewarder()
- updateEmissionPartition()
- setVeWom()
- setVoter()
- emergencyWomWithdraw()

As can be seen from the list, aside from all the setters, owner can also execute some specific functions. Owner can add new staking pools for new LP token. Thus, this is an owner's responsibility to create pools only for valid non-malicious tokens with correct Reward Booster. Owner is also in charge of setting emission partition and can withdraw all the rewards at any time.

3. Bribe Rewarder Factory

This is a smart contract which is able to set new Boosted Rewarder smart contracts for pools. The BribeRewarderFactory.sol is a contract which is supposed to be granted with this role. It contains the function deployRewarderContractAndSetRewarder() which deploys a new valid Rewarder and sets it in the BoostedMasterWombat.

WOMBAT DESCRIPTION

4. User

User is the main actor of the protocol. He is supposed to deposit funds in the smart contract, update pools when necessary and claim rewards. The following functions can be called by any user:

- massUpdatePools()
- updatePool()
- migrate()
- depositFor()
- deposit()
- multiClaim()
- withdraw()
- emergencyWithdraw()

User is able to deposit funds for specific pool and earn WOM rewards from it, as well as earn additional rewards from Boosted Rewarder and Rewarder. User can call function for updating the pools to actualize the current rewards. It is user's duty to withdraw staked funds or migrate them to new Master Wombat as well as claim rewards. It is user's responsibility to execute emergency withdraw in case funds are stuck due to rewards distribution.

5. Voter

Voter is a smart contract Voter.sol. Its responsibility is providing the necessary amount of WOM reward and calling the function notifyRewardAmount. Thus the Voter is in charge of adding new rewards in the pools.

6. veWOM

veWOM is a smart contract which represents an ERC-20 token. Its responsibility is updating users' factors every time their balances are changed on the veWOM. This is necessary in order to correctly distribute rewards which are distributed based on the factor.

WOMBAT DESCRIPTION

BoostedMultiRewarder.sol

1. Deployer

Deployer is responsible for deploying the implementation and proxy of the BoostedMultiRewarder.sol. Deployer has to pass valid initial parameters such as addresses of bribe factory and master wombat, address of an LP token which for which Boosted Rewarder is deployed, initial reward token and its emission, start date of rewards distribution. Deployer becomes an owner during the deployment.

2. Owner

Owner is responsible for adjusting all the settings and storage variables as well as adding new reward token or withdrawing them in case of emergency. The following functions can be called by the owner:

- addOperator()
- removeOperator()
- setIsDeprecated()
- addRewardToken()
- setRewardRate()
- emergencyWithdraw()
- emergencyTokenWithdraw()

3. User

Though the user is not supposed to interact with this smart contract as claiming reward hooks are called by Boosted Master Wombat, user is able to update the rewards to update reward per token rate and claim rewards in case of emergency directly. In this case, emergency is supposed to be a case where Boosted Rewarder is changed in the Master Wombat's pool.

4. Operator

Operator is a role granted by the owner. There can be multiple operators in the contract. Operator is able to add new reward tokens or change reward rate.

5. Master Wombat

Master Wombat role is supposed to be granted to the smart contract with implementation BoostedMasterWombat.sol. Master Wombat calls hooks "onReward" or "onFactorUpdate" on every harvesting of reward or updating of the factor on the Master Wombat itself. Therefore, in order to distribute rewards correctly, these hooks must always be called by Master Wombat in certain cases.

WOMBAT DESCRIPTION

MultiRewarderPerSecV2.sol

Multi Rewarder has the same roles as Boosted Multi Rewarder except the necessity of updating the factor since factor is never used in this implementation.

BribeRewarderFactory.sol

1. Owner

The owner is a central figure in the BribeRewarderFactory contract, having the authority to adjust various settings and configurations.

The owner's responsibilities and the functions they can interact with include:

- **Setting the Beacon for Rewarders:** The owner can determine which beacon should be used for deploying rewarder contracts using the `setRewarderBeacon` function.
- **Setting the Beacon for Bribes:** The owner can specify which beacon should be used for deploying bribe contracts with the `setBribeBeacon` function.
- **Determining the Rewarder Deployer:** The owner can set which address is authorized to deploy rewarder contracts for a specific asset using the `setRewarderDeployer` function.
- **Determining the Bribe Deployer:** The owner can decide which address is allowed to deploy bribe contracts for a particular asset through the `setBribeDeployer` function.
- **Whitelisting Reward Tokens:** The owner has the authority to whitelist specific reward tokens, making them eligible for being added to rewarders and bribes. This is done using the `whitelistRewardToken` function.
- **Revoking Whitelisted Reward Tokens:** If needed, the owner can also remove tokens from the whitelist, preventing them from being used in future rewarders or bribes. This action is performed using the `revokeRewardToken` function.

WOMBAT DESCRIPTION

BribeRewarderFactory.sol

2. Rewarder deployer

The Rewarder deployer is an address that has been granted permission by the owner to deploy rewarder contracts. The main responsibility is deploying rewarder contracts: Rewarder deployer can deploy new rewarder contracts. They use the `deployRewarderContract` and `deployRewarderContractAndSetRewarder` functions for this purpose. When deploying, they need to specify the LP token, start timestamp, reward token, and the emission rate.

3. Bribe deployer

The Bribe deployer can deploy new bribe contracts using the `deployBribeContract` and `deployBribeContractAndSetBribe` functions. Similar to the Rewarder deployer, they need to provide details like the LP token, start timestamp, reward token, and the emission rate when deploying.

WOMBAT DESCRIPTION

List of values assets

BoostedMasterWombat.sol

1. Staked LP tokens.

Master Wombat contains several staking pools added by the owner. Each pool has a separate LP token which user can stake and earn rewards based on their stakings. Staked tokens are stored on the contract's balance, transferred from user during either `deposit()` or `depositFor()` functions. Staked assets are stored in the 'amount' field of struct `'userInfo[poolId][user]'` where `poolId` is a number of pool corresponding a certain LP token and `user` is the address of user for whom funds are transferred. Staked funds have no lock period and no fees are taken neither for `deposit` nor for `withdrawal`. User is able to withdraw staked LP tokens by either `withdraw()` or `emergencyWithdraw()` functions.

2. Rewards.

Rewards are distributed in a form of WOM tokens. They are added by the Voter for specific pool with the function `notifyRewardAmount()`. Rewards are distributed among all the stakers in the pools over time and can be claimed with the function `multiClaim()`. Rewards are stored on contract's balance until claimed by user.

3. Factor

Factor is an additional way of distributing rewards based on. Factor is calculated based on deposited amount and user's balance in veWOM. Thus, factor must be recalculate every time user's balance in veWOM is changed. Thanks to factor, users are able to boost their outcomes by holding veWOM tokens and staking LP tokens in the Master Wombat.

WOMBAT DESCRIPTION

BoostedMultiRewarder.sol and MultiRewarderPerSecV2.sol

1. Rewards.

Contract contains several rewards tokens added by the owner. Rewards are distributed to the staked of Wombat Master in specific pool. Rewards are collected every time a user collects rewards on the Wombat Master. Rewards are stored on the contract's balance until claimed by user and are distributed in two modes based on total staked amount and factor.

2. Unpaid rewards.

Unpaid rewards is a value from struct `userRewardInfo[i][user]` where 'i' is the index of reward token and 'user' is the address of user. This value represents the amount of rewards which couldn't be paid to user due to lack of reward balance and should be paid later.

BribeRewarderFactory.sol

1. Rewarder Deployers.

The BribeRewarderFactory contract has a mapping called `rewarderDeployers` that associates specific LP tokens (represented as `IAsset`) with authorized addresses. These addresses, known as Rewarder Deployers, have the permission to deploy rewarder contracts for the associated LP tokens. The owner of the BribeRewarderFactory contract can set or change these deployers. The LP tokens in this mapping represent potential staking assets for users in the deployed rewarder contracts.

2. Bribe Deployers.

Similarly, there's another mapping named `bribeDeployers`. This mapping links LP tokens (again, represented as `IAsset`) to specific addresses, known as Bribe Deployers. These addresses are authorized to deploy bribe contracts for the linked LP tokens. The owner has the authority to determine or modify these deployers. The LP tokens in this mapping indicate potential staking assets for users in the deployed bribe contracts.

WOMBAT DESCRIPTION

BribeRewarderFactory.sol

3. Whitelisted Reward Tokens.

The contract maintains a set of tokens that are whitelisted to be used as rewards. This set, named `whitelistedRewardTokens`, contains the addresses of all approved reward tokens. The owner can add tokens to this whitelist and also has the power to remove them. These whitelisted tokens are valuable assets because they represent potential rewards that users can earn by participating in the rewarder or bribe contracts.

4. Rewarder Beacon.

The `rewarderBeacon` is a crucial asset in the contract. It's an instance of `IBeacon` and points to the logic used for deploying rewarder contracts. This beacon determines the behavior and functionality of any rewarder contract deployed through it. The owner sets this beacon during the contract's initialization and can update it later if needed.

5. Bribe Beacon

The `bribeBeacon` operates similarly to the `rewarderBeacon`. It's an instance of `IBeacon` that holds the logic for deploying bribe contracts. This beacon is essential as it defines the logic and behavior of any bribe contract deployed using it. Like the rewarder beacon, it's set during initialization by the owner, who can also update it later.

WOMBAT DESCRIPTION

List of potential threats

BoostedMasterWombat.sol

1. User is able to withdraw more funds then he staked.
2. User can withdraw same rewards multiple time.

BoostedMultiRewarder.sol and MultiRewarderPerSecV2.sol

1. Is deprecated flag is set multiple times allowing users to claim same rewards.

BribeRewarderFactory.sol

1. The contract allows setting rewarder and bribe deployers for specific LP tokens. If an unauthorized or malicious deployer is set, they can deploy malicious contracts.

Note! All of these threats were checked during testing. The issues weren't confirmed and thus won't lead to loss of funds or other risks.

WOMBAT DESCRIPTION

Settings

BoostedMasterWombat.sol

Most of the settings are setters for smart contracts which represents certain role. Thanks to these settings, vital functions of smart contract can only be called by authorized entities.

The pool structure contains essential settings which are set by function `add()`, `setRewarder()` and `notifyRewardAmount()`. All of these settings are necessary for the pool to receive valid staking LP tokens, distribute WOM and call Rewarder in order to distribute additional rewards.

BoostedMultiRewarder.sol and MultiRewarderPerSecV2.sol

Both contracts have similar settings for roles which allow the execution essential functions by authorized entities only.

Smart contract has vital settings per each added reward token which are set by owner or operators. Settings represent the reward rate and the start of reward distribution. These settings should be set properly in order to distribute a sufficient amount of rewards in the appropriate time bounds.

BribeRewarderFactory.sol

1. Master Wombat contract (`masterWombat`):

This is a reference to the `IBoostedMasterWombat` contract. It is a central contract that manages various staking pools and their associated rewarders. The `BribeRewarderFactory` interacts with this contract to get pool IDs, set rewarders, and more.

2. Rewarder beacon (`rewarderBeacon`):

This is an instance of `IBeacon` that points to the logic used for deploying rewarder contracts. It determines the behavior and functionality of any rewarder contract deployed through it. The owner can update this beacon using the `setRewarderBeacon` function.

WOMBAT DESCRIPTION

3. Voter contract (voter):

This is a reference to the IVoter contract. The BribeRewarderFactory interacts with this contract to get information about assets, gauge managers, and more. It's essential for deploying both rewarder and bribe contracts.

4. Bribe beacon (bribeBeacon):

Similar to the rewarderBeacon, this is an instance of IBeacon that holds the logic for deploying bribe contracts. It defines the logic and behavior of any bribe contract deployed using it. The owner can update this beacon using the setBribeBeacon function.

5. Rewarder deployers (rewarderDeployers mapping):

This mapping refers to specific LP tokens (represented as IAsset) with authorized addresses, known as Rewarder Deployers. These addresses are permitted to deploy rewarder contracts for the associated LP tokens. The owner can set or change these deployers using the setRewarderDeployer function.

6. Bribe deployers (bribeDeployers mapping):

This mapping links LP tokens (represented as IAsset) to specific addresses, known as Bribe Deployers. These addresses are authorized to deploy bribe contracts for the linked LP tokens. The owner can determine or modify these deployers using the setBribeDeployer function.

7. Whitelisted reward tokens (whitelistedRewardTokens set):

This is a set of tokens that are whitelisted to be used as rewards. The owner can add tokens to this whitelist using the whitelistRewardToken function and can remove them using the revokeRewardToken function.

WOMBAT DESCRIPTION

Deployment script

BoostedMasterWombat.sol

The audited deployment script is located at `./deploy/105_BoostedMasterWombat.ts`.

Script deploys the Proxy Admin for updates managements, implementation and proxy. The initial storage variables are initialized with the following values:

wom: initialized with a proper WOM address.

veWom: initialized with zero address.

voter: initialized with zero address.

basePartition: 375 (37,5 %).

The rest of parameters is set in `./deploy/197_BoostedMasterWombatSetup.ts`. In this script the following variables are:

veWom: initialized only if veWom was deployed previously and is present in deployment config.

voter: initialized only if Voter was deployed previously and is present in deployment config.

bribeRewarderFactory: initialized only if bribeRewarderFactory was deployed previously and is present in deployment config.

Initial pools for assets are created in this script as well using the function `add()`.

All in all, the deployment script is well written, follows the best practices for proxy deployment, and stores the deployed addresses in a config file. All the necessary variables, except creation of pool, are properly set.

WOMBAT DESCRIPTION

BoostedMultiRewarder.sol

The implementation and beacon proxy for BoostedMultiRewarder are deployed in **./deploy/195_BribeRewarderBeacon.ts**. The beacon is then set in the BribeRewarderFactory during its deployment in **./deploy/196_BribeRewarderFactory.ts**.

Proxies for Boosted Multi Rewarders are deployed by the Factory in the function **deployRewarderContractAndSetRewarder()**. This function can be called only by a valid deployer who is set by the owner of the Factory. This deployer must provide valid parameters for the Rewarder.

BribeRewarderFactory.sol

The implementation and beacon proxy for BribeRewarderFactory is deployed in **./deploy/196_BribeRewarderFactory.ts**.

Arguments for **initialize()** function are set to **rewarderBeaconAddr**, **bribeBeaconAddr**, **mwAddr** and **voter**, which were previously deployed.

After the deployment, address of newly created **BribeRewarderFactory** is set as parameter in **Voter contract** via **setBribeFactory()** method.

MultiRewarderPerSecV2.sol

No deployment script for this smart contract was found. We recommend Wombat team to add deployment scripts if it needed to deploy this implementation separately without **BribeV2**.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

 **Critical**

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

 **High**

The issue affects the ability of the contract to compile or operate in a significant way.

 **Medium**

The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior.

 **Low**

The issue has minimal impact on the contract's ability to operate.

 **Informational**

The issue has no impact on the contract's ability to operate.

COMPLETE ANALYSIS

MEDIUM-1 | RESOLVED

Iteration through the whole storage array.

BoostedMultiRewarder.sol: `_addRewardToken()`, line 210;
`_updateReward()`, line 244; `onUpdateFactor()`, line 317; `_onReward()`, line 357; `_pendingTokens()`, line 447; `_rewardsToDistribute()`, line 511;
`emergencyWithdraw()`, line 600.
MultiRewarderPerSecV2.sol: same functions as in BoostedMultiRewarder except `onUpdateFactor()`.

In the following functions and several view functions, the loop iterates through the whole storage array ``rewardInfos``. Such iteration may revert due to out-of-gas error in case there are many rewards. Also, in function `_addRewardToken()`, iteration is used to determine if ``_rewardToken`` is already included, which is inefficient regarding gas spending.

BoostedMasterWombat.sol: `massUpdatePools()`, `updateFactor()`, line 546.

The following functions iterate through all the pools in the MasterWombat. Though the issue with `massUpdatePools()` carries informational nature, the `updateFactor()` may revert due to such iteration, especially since this function is already called from the smart contract `veWom`.

Recommendation:

Add an additional mapping to mark if the token is already included in the rewards list. Consider adding a restriction on the maximum number of reward tokens to ensure that it's impossible to add many rewards in the array, which may prevent the iteration through it.

Post-audit.

The Wombat team has added the maximum limitation in the code.

Lack of events.

BoostedMasterWombat.sol: setNewMasterWombat(), setBribeRewarderFactory(), migrate().
BoostedMultiRewarder.sol: setIsDeprecated().
MultiRewarderPerSecV2.sol: setIsDeprecated().
BribeRewarderFactory.sol: setRewarderDeployer(), setBribeDeployer(),
whitelistRewardToken(), revokeRewardToken().
Essential functions such as setters should emit events to keep track of historical changes in data.

Recommendation:

Emit events in setters and main functions.

Post-audit.

Events for corresponding functions were added.

Owner is able to rewrite boosted rewarder.

BoostedMasterWombat.sol: setRewarder().

The owner can rewrite a boosted rewarder for the pool even if it is already set. As a result, users may lose part of their rewards if they weren't on time to claim them. This issue was marked as info since only the owner can execute the function. However, it is still recommended to ensure that users have claimed pending rewards before updating the address of the boosted rewarder.

Recommendation:

Add a restriction that the boosted rewarder can't be re-written OR ensure that users will have time to claim rewards before the rewarder is set by adding a timelock functionality and notifying users that the boosted rewarder is about to be changed.

Post-audit:

The team has verified this issue, and the owner will be set to multisig.

Custom errors should be used.

Starting from the 0.8.4 version of Solidity, it is recommended to use custom errors instead of storing error message strings in storage and use "require" statements. Using custom errors is more efficient regarding gas spending and increases code readability.

Recommendation:

Use custom errors.

Post-audit:

The team has verified this issue as, according to them, this gas optimization does not warrant rewriting the code.

Both Ownable and AccessControl are used.

BoostedMultiRewarder.sol, MultiRewarderPerSecV2.sol.

Contract inherits both Ownable and AccessControl contracts by OpenZeppelin for role management. However, the role DEFAULT_ADMIN from AccessControl is never granted in the initializer. Therefore, native AccessControl functions such as grantRole() or revokeRole() can never be called by the admin directly. Instead, custom functions addOperator()/removeOperator() are used. Though the contract works fine, inheriting both access management implementations contradicts best practices as it complicates the implementation of the smart contract. Using only AccessControl and using the DEFAULT_ADMIN role as an owner is recommended.

Recommendation:

Remove Ownable and use the DEFAULT_ADMIN role from AccessControl as an owner role.

Post-audit:

According to the development team, this is correct behavior, as having this functionality will make the smart contracts more manageable.

No check for zero balance.

BoostedMasterWombat.sol: `emergencyWithdraw()`

A user who has not made a deposit will successfully call for the withdrawal of tokens with a zero amount.

Recommendation:

Add a zero balance check.

Post-audit:

According to the development team, this is correct behavior as emergencyWithdraw is an emergency function that lets users trigger the rewarder even though they have withdrawn all their LP.

Incorrect display of the number of tokens withdrawn.

BoostedMasterWombat.sol: `emergencyWithdraw()`

The 'user.amount' argument passed to the 'EmergencyWithdraw' event will always be 0 since its state is overwritten above the code 'user.amount = 0'

Recommendation:

Store the user's balance before withdrawal in a local variable and pass it to the event.

Post-audit:

The user's balance before withdrawal is now stored in a local variable and passed to the event.

Rewarder is never set.

BoostedMasterWombat.sol: struct PoolInfoV3, field `rewarder`.

Field rewarder is used in several functions when rewards are claimed. Though the variable is checked not to be zero address before every usage, it is never assigned with any other value other than zero address. Hence, the variable is currently unused.

Recommendation:

Remove unused field OR verify that the setter will be added later.

Post-audit:

Setter for the rewarder was added.

Storage constant should be used.

BoostedMasterWombat.sol: value `1000` representing the maximum base partition.

BoostedMultiRewarder.sol: setRewardRate(), line 278, value `10000e18`.

Value 1000 was used several times across the contract. Therefore, it is recommended to use a storage constant instead for better code readability.

The same happens for BoostedMultiRewarder with value `10000e18`.

Recommendation:

Use a storage constant instead of the value directly in the code.

Post-audit:

A storage constant is now used.

Function visibility might be changed from “public” to “external”.

BribeRewarderFactory.sol: getWhitelistedRewardTokens()

The function is not used inside any of the contracts. Therefore, visibility might be changed for the sake of optimization.

Recommendation:

Change visibility for getWhitelistedRewardTokens().

Post-audit.

Visibility was changed from “public” to “external”.

BribeRewarderFactory.sol BoostedMasterWombat.sol BoostedMultiRewarder.sol MultiRewarderPerSecV2.sol BribeV2.sol	
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Wombat in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the Wombat contract requirements for details about issuance amounts and how the system handles these.

BoostedMasterWombat

initialize method

- ✓ Should be a reinitialization error (1317ms)
- ✓ Should be deployed (113ms)

pause method

- ✓ Should be on pause (268ms)
- ✓ Should be an access error

unpause method

- ✓ Should not be on pause (47ms)
- ✓ Should be an access error (42ms)

emergencyWomWithdraw method

- ✓ Should migrate WOM funds from MasterChef to contract owner (70ms)
- ✓ Should be an access error (55ms)

updateEmissionPartition method

- ✓ Should update emission partition (48ms)
- ✓ Should be an error of incorrect base partition amount
- ✓ Should be an access error

add method

- ✓ Should add a new LP to the pool

deposit method

- ✓ Should add a deposit to user (175ms)

depositFor method

- ✓ Should add a deposit to another user (137ms)

emergencyWithdraw method

- ✓ Should withdraw the tokens to the user (202ms)
- ✓ Should be a successful tx (140ms)

withdraw method

- ✓ Should withdraw the specified number of tokens to the user (219ms)

migrate method

- ✓ Should withdraw the specified number of tokens to the user (435ms)

Scenario #1

- ✓ Should issue a reward to the user (306ms)

Scenario #2

- ✓ Should correctly transfer user data (814ms)

BoostedMultiRewarder

initialize method

- ✓ Should be deployed (411ms)
- ✓ Should be a reinitialization error

emergencyTokenWithdraw method

- ✓ Should withdraw ETH
- ✓ Should withdraw tokens (39ms)

emergencyTokenWithdraw method

- ✓ Should withdraw tokens (69ms)

MultiRewarderPerSecV2

initialize method

- ✓ Should be deployed (451ms)
- ✓ Should be a reinitialization error

emergencyTokenWithdraw method

- ✓ Should withdraw ETH
- ✓ Should withdraw tokens (42ms)

emergencyTokenWithdraw method

- ✓ Should withdraw tokens (69ms)

BribeRewarderFactory

Rewarder:

- ✓ should revert when not authorized actor is deploying Rewarder
- ✓ should revert when deploying same Rewarder for the second time
- ✓ should revert when deploying Rewarder with not whitelisted token
- ✓ should deploy Rewarder contract and set Rewarder to Voter

Bribe:

- ✓ should revert when deploying Bribe with non-existing gauge
- ✓ should revert when deploying Bribe when bribe contract already exists for gauge
- ✓ should revert when deploying Bribe with not whitelisted token
- ✓ should deploy Bribe contract and set Bribe to Voter

Scenarios

- ✓ Set a deployer, remove them, and then re-add them. Does their ability to deploy rewarders or bribes get restored correctly?
- ✓ Whitelist a reward token, deploy a rewarder and bribe using it, and then revoke the token from the whitelist. Does the contract and the deployed entities behave correctly?
- ✓ Migration scenario. Four Users make a deposit. A new 'BoostedMasterWombat' is created. The second user performs 'migrate'. Check the vault on the old 'BoostedMasterWombat' contract and the new one for the users. Then the third user executes 'migrate'. Check the repositories again
- ✓ Getting rewarded for locking WOM. User number 1 receives veWOM tokens in exchange for WOM tokens. Other users also receive veWOM, thus increasing the amount of reward for the first user. User number 1 withdraws his reward. Check the user's contract balance before and after requesting rewards.

42 passing (15s)

FILE	% STMTS	% BRANCH	% FUNCS
BribeRewarderFactory.sol	100	100	100
BoostedMasterWombat.sol	100	95.83	100
BoostedMultiRewarder.sol	100	95	100
MultiRewarderPerSecV2.sol	80	98.33	100
BribeV2.sol	100	95.89	100

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by the Wombat team

As a part of our work assisting Wombat in verifying the correctness of their contract code, our team has checked the complete set of tests prepared by the Wombat team.

We need to mention that the original code has a significant original coverage with testing scenarios provided by the Wombat team. All of them were also carefully checked by the team of auditors.

BoostedMultiRewarder

```
shouldDeployMockTokens true [
  {
    tokenName: 'Binance USD',
    tokenSymbol: 'BUSD',
    decimalForMockToken: 18
  },
  {
    tokenName: 'USD Coin',
    tokenSymbol: 'USDC',
    decimalForMockToken: 18
  },
  {
    tokenName: 'Tether USD',
    tokenSymbol: 'USDT',
    decimalForMockToken: 18
  },
  {
    tokenName: 'TrueUSD',
    tokenSymbol: 'TUSD',
    decimalForMockToken: 18
  },
  {
    tokenName: 'Dai Stablecoin',
    tokenSymbol: 'DAI',
    decimalForMockToken: 18
  },
  {
    tokenName: 'Venus USDC',
    tokenSymbol: 'vUSDC',
```

```

    decimalForMockToken: 8
  },
  {
    tokenName: 'Axelar Wrapped USDC',
    tokenSymbol: 'axlUSDC',
    decimalForMockToken: 6
  }
]

```

✓ user can call `rewarder.emergencyClaimReward()` to get the pending tokens.

✓ should emit event when `setIsDeprecated`

axlUSDC (6 decimal)

✓ rewards 1 axlUSDC/s

✓ works with small shares

✓ does not overflow at max reward rate for one month

USDC (18 decimal)

✓ rewards 1 USDC/s

✓ does not overflow at max reward rate for one month

✓ stops emission when balance runs out. single user case

✓ stops emission when balance runs out. multi-user case

runout timestamp

✓ extends when balance increases

✓ is invariant as time passes, until the time passes `runoutTimestamp`, then it becomes 0

✓ is invariant upon `onReward()`, (users' deposit, withdrawal or claim) even though balance decreases

Emission

✓ `UpdateReward` does not crash when there are no shares

✓ A user cannot steal all bribes

✓ rewarder should reuse master's partition

BribeRewarderFactory

✓ deploy validation

rewarder

✓ validation

✓ `deployRewarderContractAndSetRewarder`

✓ upgrade

bribe

✓ validation

✓ `deployBribeContractAndSetBribe`

✓ should emit event when `rewarderDeployer` is set

✓ should emit event when `bribeDeployer` is set

✓ should emit event when change `whitelistRewardToken`

✓ upgrade

BoostedMasterWombat

Master wombat Utils

- ✓ should pause and unpaue
- ✓ withdraw full wom balance on emergencyWomWithdraw
- ✓ only veWom can call updateFactor
- ✓ should revert if the same lpToken is added into the pool
- ✓ should set correct state variables womPerSec, veWom and emission repartition
- ✓ should check rewarder added and set properly
- ✓ should allow emergency withdraw from MasterWombat
- ✓ emergency withdraw
- ✓ should emit event when bribeRewarderFactory is set

[USDC Pool] Base pool only

- ✓ should give out wom when withdrawing

[USDC Pool] Non-dualuting pool only

- should give out wom when withdrawing

[All pools] Base pool only

- should withdraw and claim wom
- should multiclaim from certain pools only
- should multiclaim from one pool only
- should multiclaim from all pools
- should claim with the deposit function

[All pools] Base + boosted pool

- ✓ should withdraw and claim wom
- ✓ should claim with the deposit function

[All pools] veWOM integration test

- ✓ should set & update factor and sumOfFactors correctly

[All pools] LP token Migration

- ✓ should revert if newMasterWombat not set
- ✓ should emit event when newMasterWombat is set
- ✓ user should be able to migrate once only
- ✓ should claim WOM before migrate()
- ✓ should claim WOM in new MasterWombat pool during migration

notifyRewardAmount

- ✓ can be called from voter
- ✓ cannot be called by owner
- ✓ cannot be called by user

RewarderV2

```

shouldDeployMockTokens true [
  {
    tokenName: 'Binance USD',
    tokenSymbol: 'BUSD',
    decimalForMockToken: 18
  },
  {

```

```

tokenName: 'USD Coin',
  tokenSymbol: 'USDC',
  decimalForMockToken: 18
},
{
  tokenName: 'Tether USD',
  tokenSymbol: 'USDT',
  decimalForMockToken: 18
},
{
  tokenName: 'TrueUSD',
  tokenSymbol: 'TUSD',
  decimalForMockToken: 18
},
{
  tokenName: 'Dai Stablecoin',
  tokenSymbol: 'DAI',
  decimalForMockToken: 18
},
{
  tokenName: 'Venus USDC',
  tokenSymbol: 'vUSDC',
  decimalForMockToken: 8
},
{
  tokenName: 'Axelar Wrapped USDC',
  tokenSymbol: 'axlUSDC',
  decimalForMockToken: 6
}
]

```

✓ user can call `rewarder.emergencyClaimReward()` to get the pending tokens.

axlUSDC (6 decimal)

- ✓ rewards 1 axlUSDC/s
- ✓ works with small shares
- ✓ does not overflow at max reward rate for one month

USDC (18 decimal)

- ✓ rewards 1 USDC/s
- ✓ does not overflow at max reward rate for one month
- ✓ stops emission when balance runs out. single user case
- ✓ stops emission when balance runs out. multi-user case

runout timestamp

- ✓ extends when balance increases

- ✓ is invariant as time passes, until the time passes runoutTimestamp, then it becomes 0
- ✓ is invariant upon onReward(), (users' deposit, withdrawal or claim) even though balance decreases

Emission

- ✓ UpdateReward does not crash when there are no shares
- ✓ A user cannot steal all bribes

59 passing (36s)

6 pending

We are grateful for the opportunity to work with the Wombat team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Wombat team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

