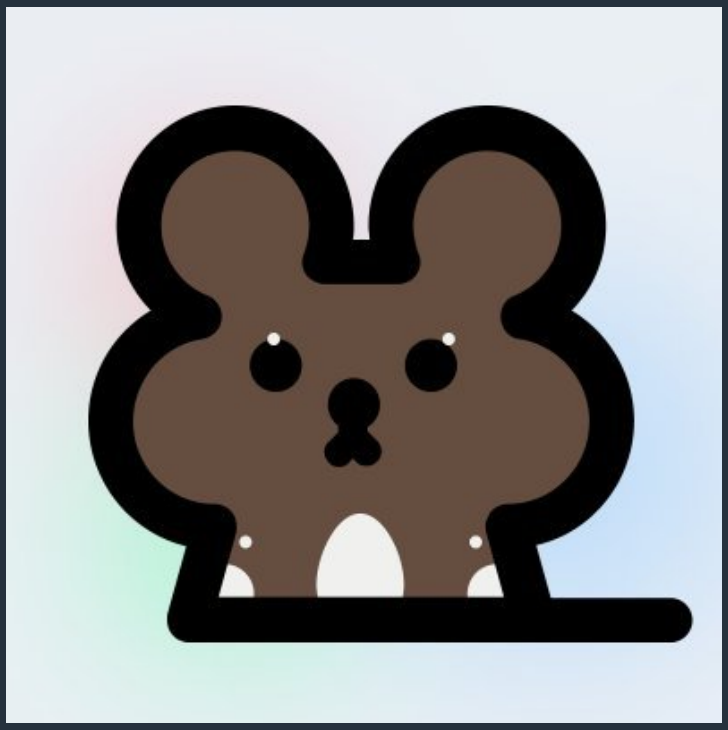


Quoll Finance
SMART CONTRACT AUDIT

ZOKYO.

October 19th 2022 | v. 1.0

PASS

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

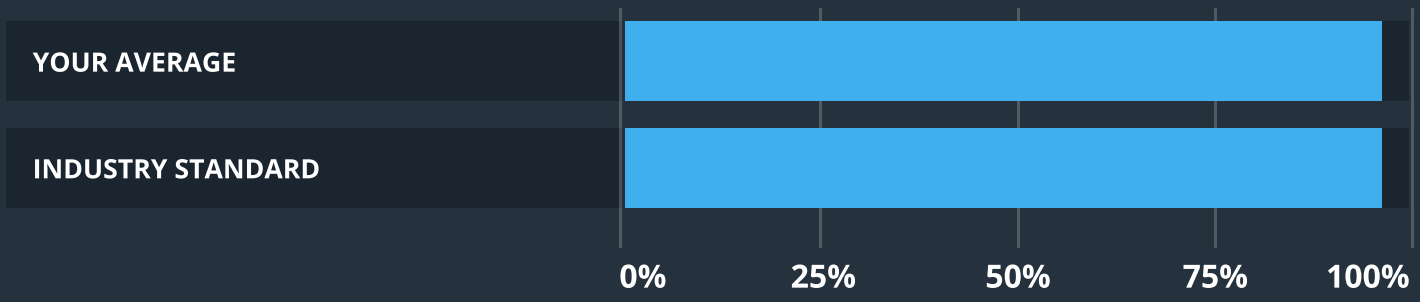
TECHNICAL SUMMARY

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

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

Contract Status

Testable Code



The 95% of the code is testable, which corresponds the standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Quoll Finance 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

The source code of the smart contract was taken from the Quoll Finance repository.
https://github.com/quollfi/quoll

Initial commit: dd385c2aa8aa9bf1bad1506d3afc4bb606244cfa

Final commit: bf3dc740926fdac83fa8352c9c9263084ef37273

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

- BaseRewardPool.sol
- DepositToken.sol
- QuoRewardPool.sol
- QuollExternalToken.sol
- QuollMasterChef.sol
- QuollToken.sol
- QuollZap.sol
- VlQuo.sol
- WomDepositor.sol
- WombatBooster.sol
- WombatVoterProxy.sol

AUDITING STRATEGY AND TECHNIQUES APPLIED

Throughout the review process, Zokyo Security ensures that the contract:

- 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 in efficient use of resources, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Quoll Finance smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented any issues as they were discovered. A part of this work included writing a unit test suite using the Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

1	Due diligence in assessing the overall code quality of the codebase.	3	Testing contract logic against common and uncommon attack vectors.
2	Cross-comparison with other, similar smart contracts by industry leaders.	4	Thorough manual review of the codebase, line by line.

EXECUTIVE SUMMARY

The Zokyo Security team has received a whole set of contract for an audit, which were provided by the Quoll team. The Quoll protocol consists of several contracts built on top of the Wombat exchange and aimed to provide boosted yield and extra rewards to their users.

The goal of the audit was to validate the contracts' code against the list of common security vulnerabilities, verify the security of users' funds, and check the correctness of the interaction with third-party protocols such as Wombat exchange. To do this, the contracts were analyzed manually, with automated tools, and with fork unit-tests.

There were 2 high-severity issues found during the audit. The first one was connected to the possible lack of rewards on the reward pool contract, which could prevent users from withdrawing their funds since rewards are paid during withdrawals as well. The second issue was connected to a possible frontrunning attack while interacting with the Wombat exchange. Both of these issues were succesfully fixed by the Quoll team. Other issues were connected to stuck ETH, variable validation, and bussiness logic validation. All of them were successfully resolved or verified.

However, there are some aspects of the code that should be highlighted in the report. First of all, all contracts are upgradable, including the code of ERC20 tokens. This means, that the owner of the contracts can update their logic at any time. The code of DepositToken and QuollExternalToken contains burn functionality, allowing the operator to burn tokens from any balance. Yet, the Quoll team ensured that the operator role will only be granted to the contracts of the Quoll protocol that were present in the audit scope.

Nevertheless, since ERC20 tokens are upgradeable and contain mandatory burn functions, it is considered that they have backdoors. It is reflected in the table of standard checks.

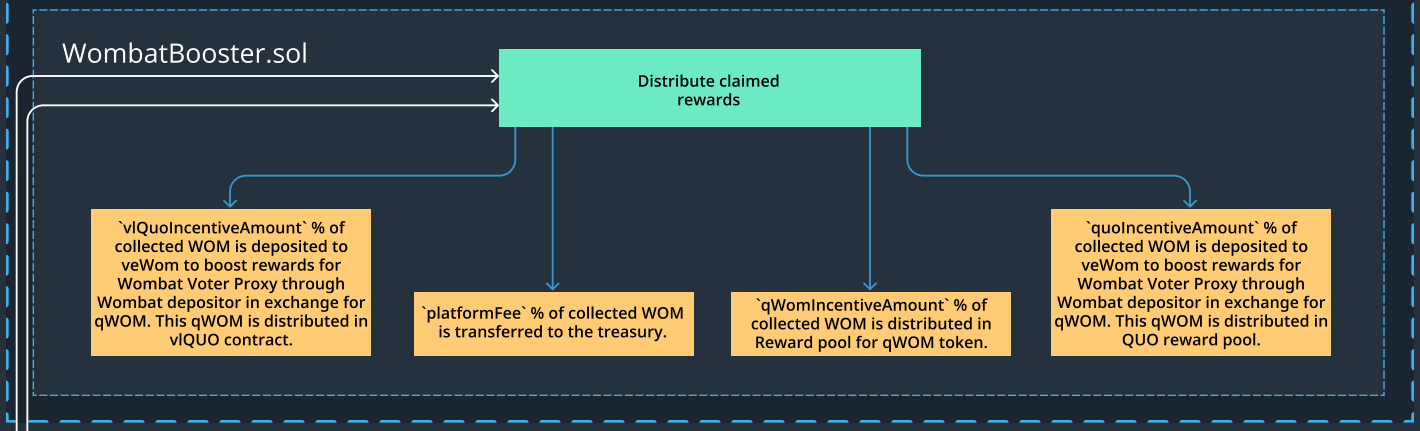
It also needs to be mentioned that the set of contracts contains no unit tests, no deployment scripts, and no project configuration, making it hard to verify if the deployment will be performed in a secure way.

As a part of the audit, Zokyo Security prepared a set of unittests ran on the fork of BNB Smart Chain. The security of funds and the interaction with the Wombat exchange protocol were verified during testing, as well as the bussiness logic of the protocol.

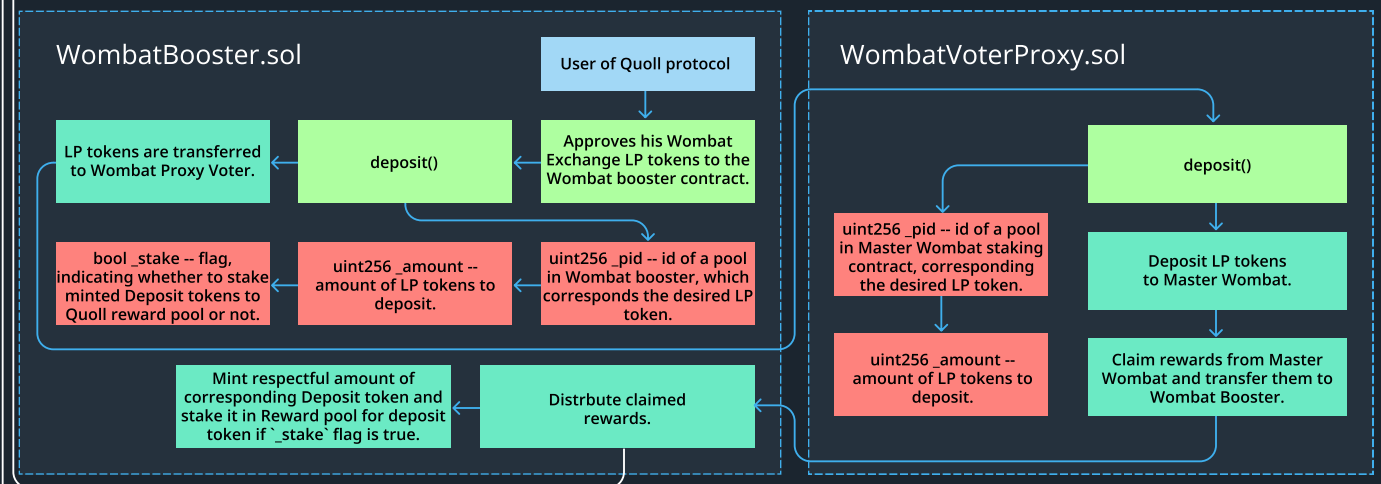
The overall security of the contracts is high enough, though there are still concerns about the tokens upgradeability and lack of deployment verification. All the crucial audit issues were successfully resolved by the Quoll team. The contracts are well-written, but they lack natspec documentation. Also, there were no unit-tests provided by the Quoll team. Nevertheless, Zokyo Security has prepared their own set of unit-tests to validate the bussiness logic of the contracts.

WombatBooster, WombatVoterProxy, WomDepositor

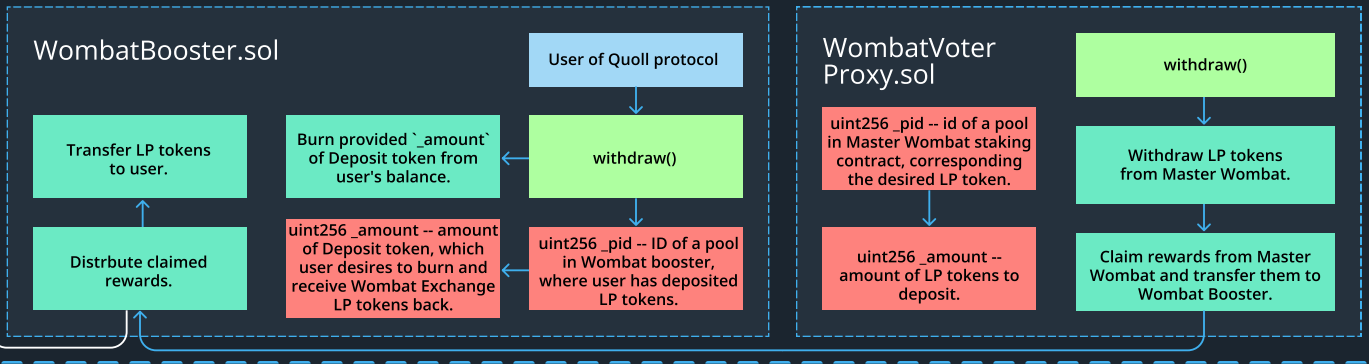
Rewards distribution in WombatBooster.sol



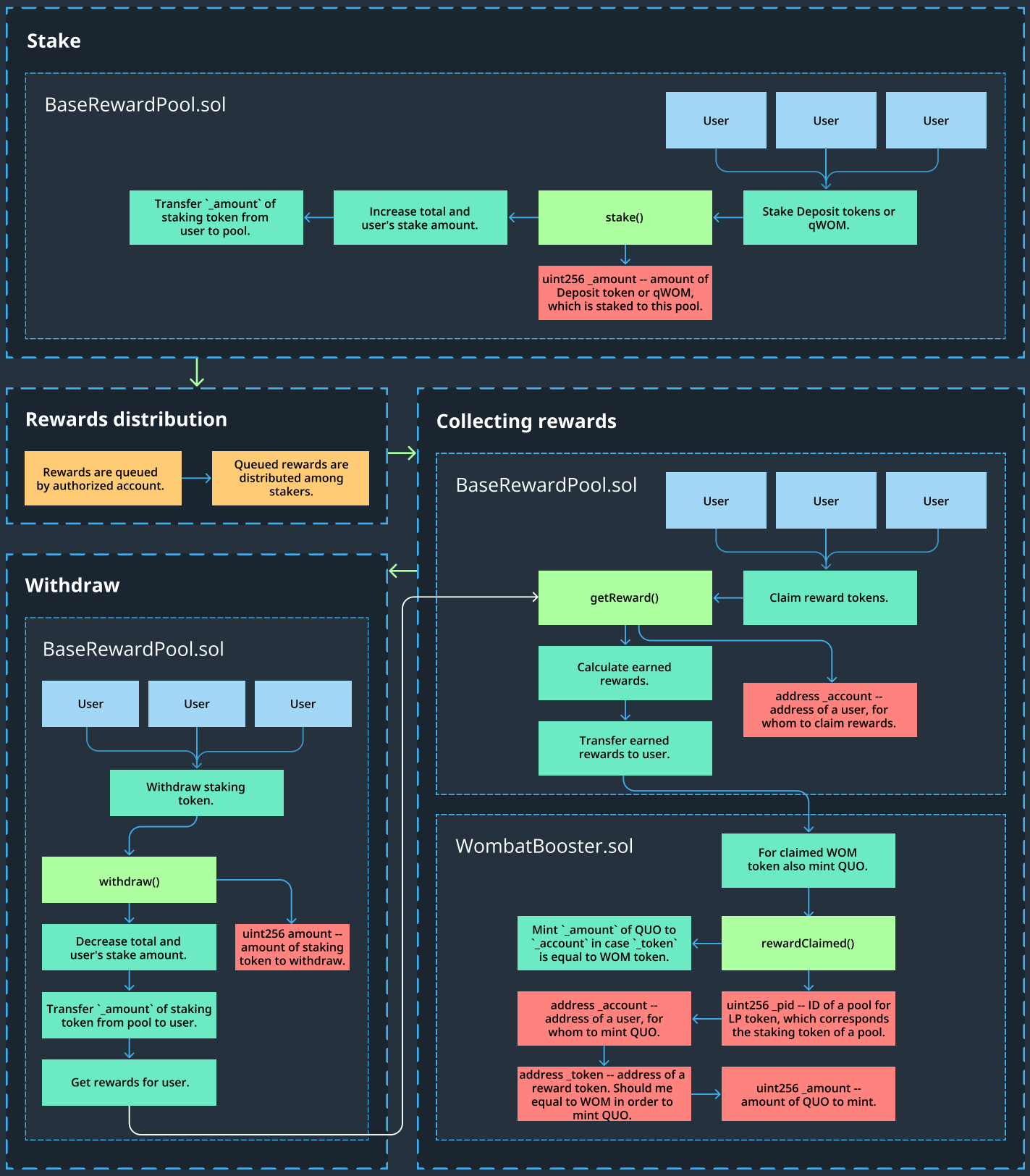
Deposit flow in WombatBooster.sol



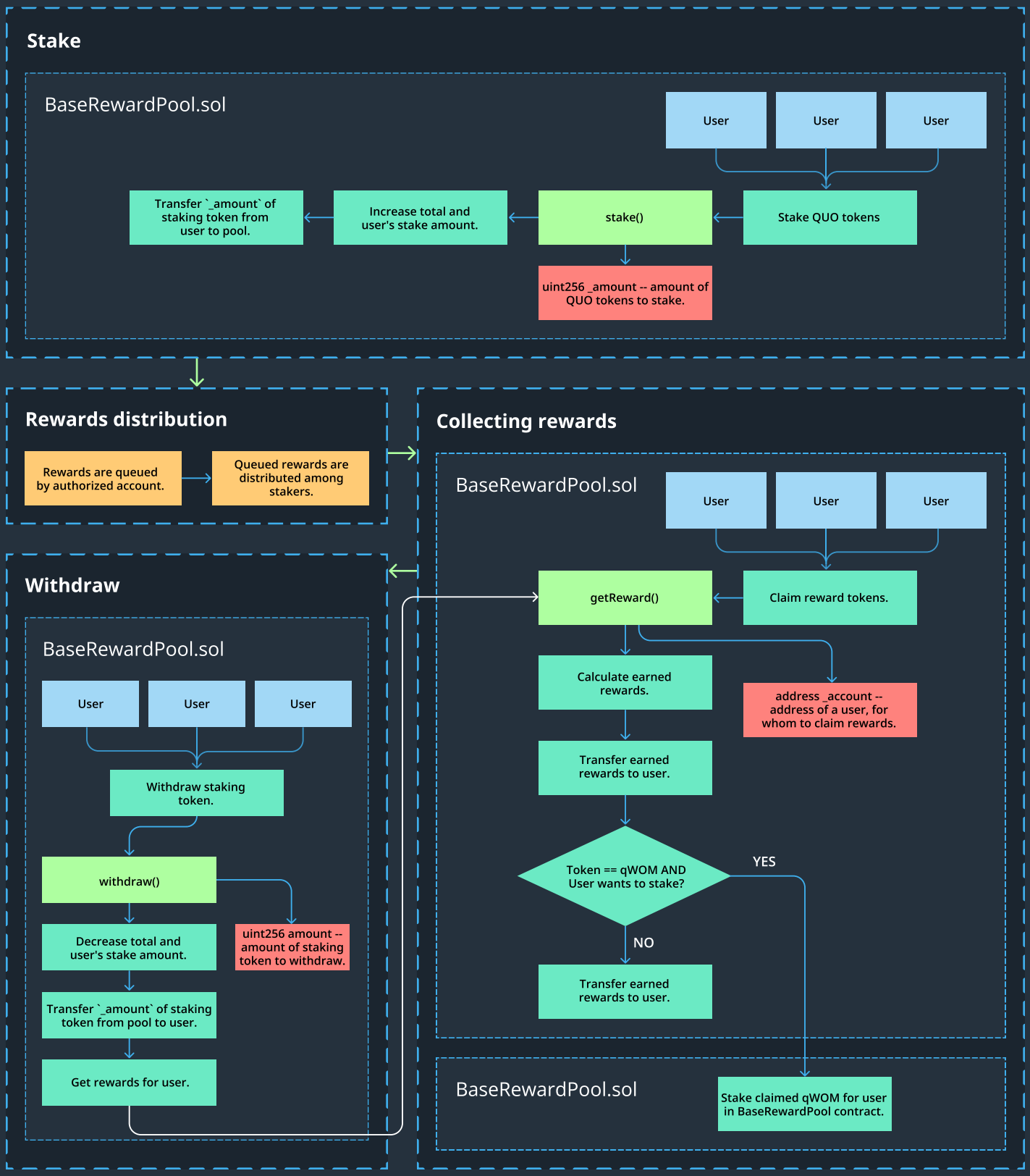
Withdraw flow in WombatBooster.sol



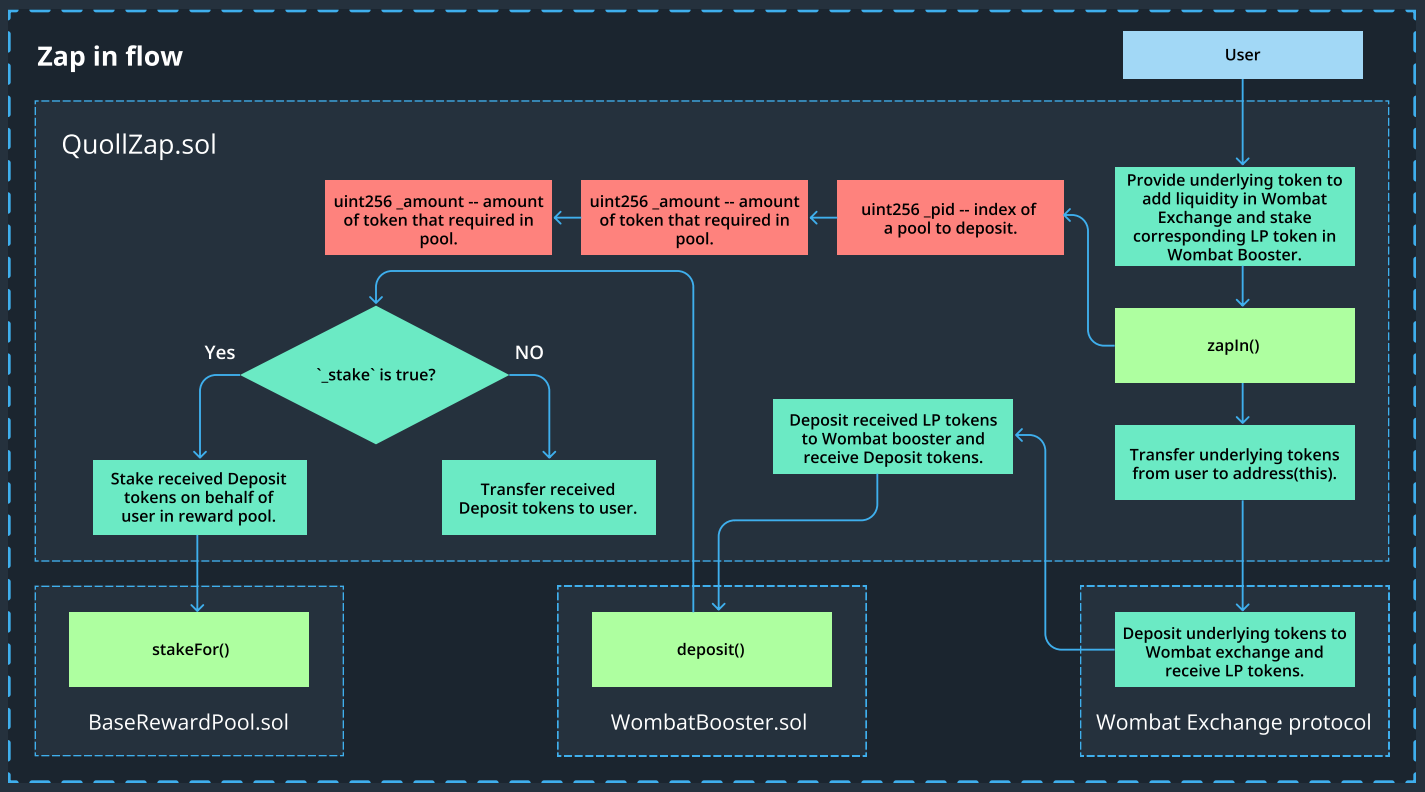
BaseRewardPool

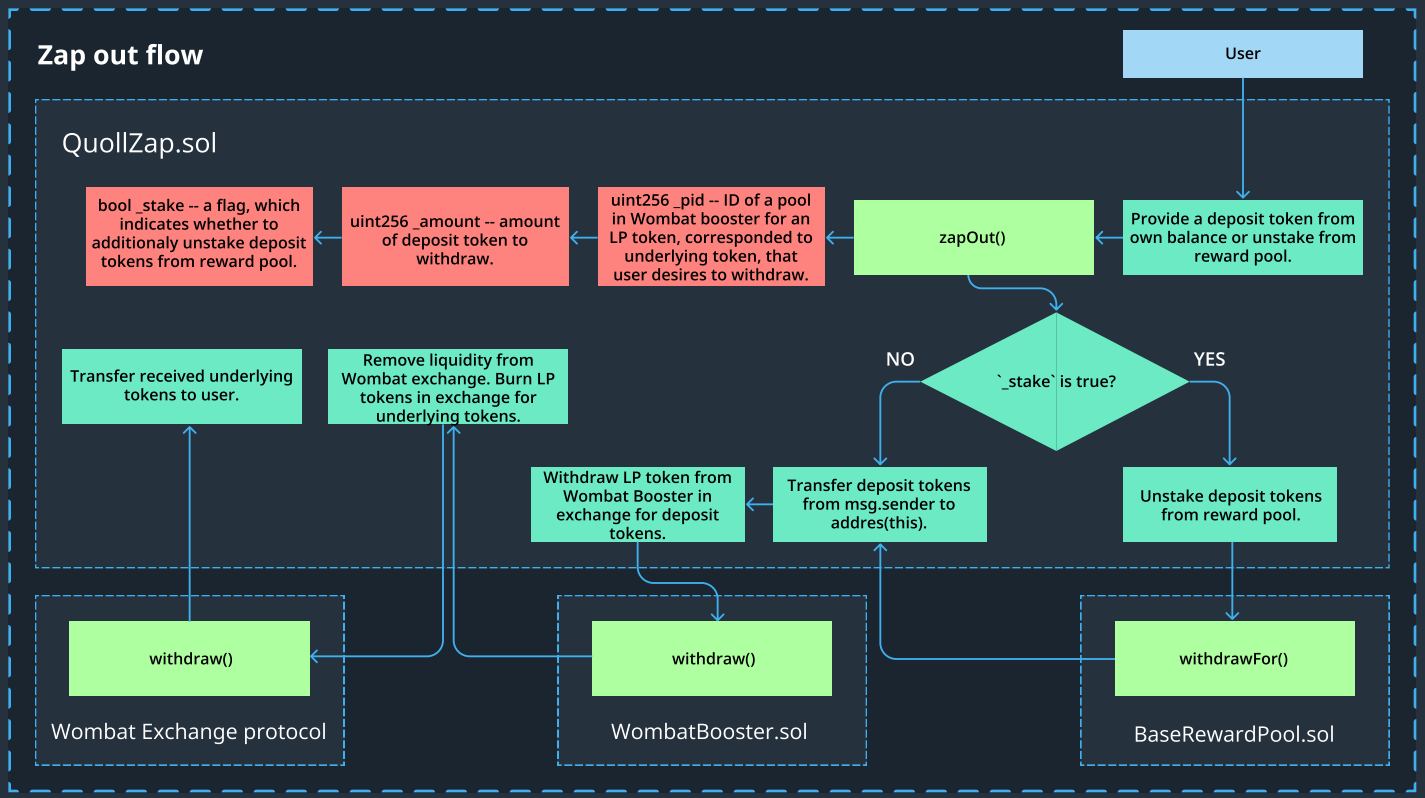


QuoRewardPool

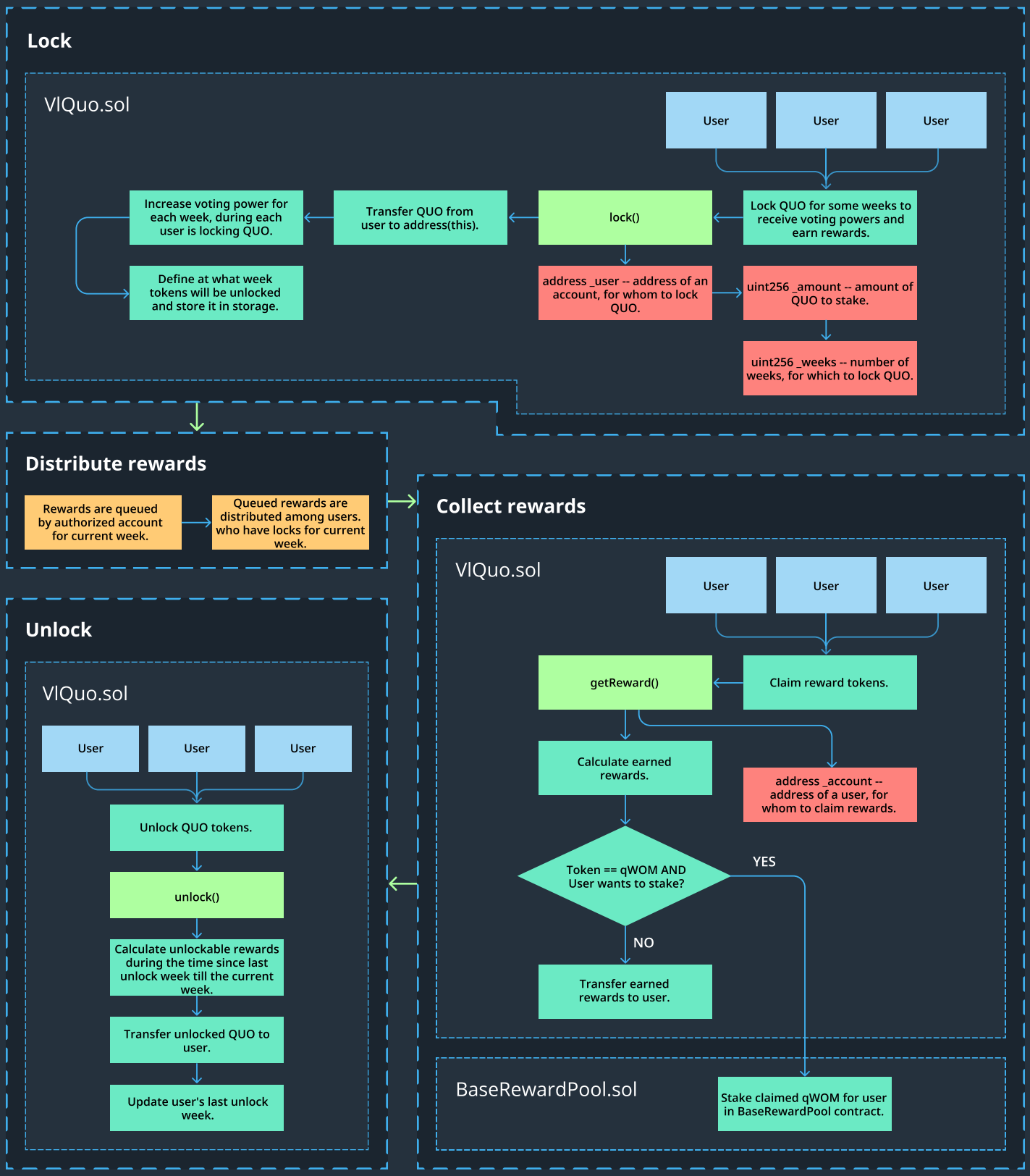


QuollZap

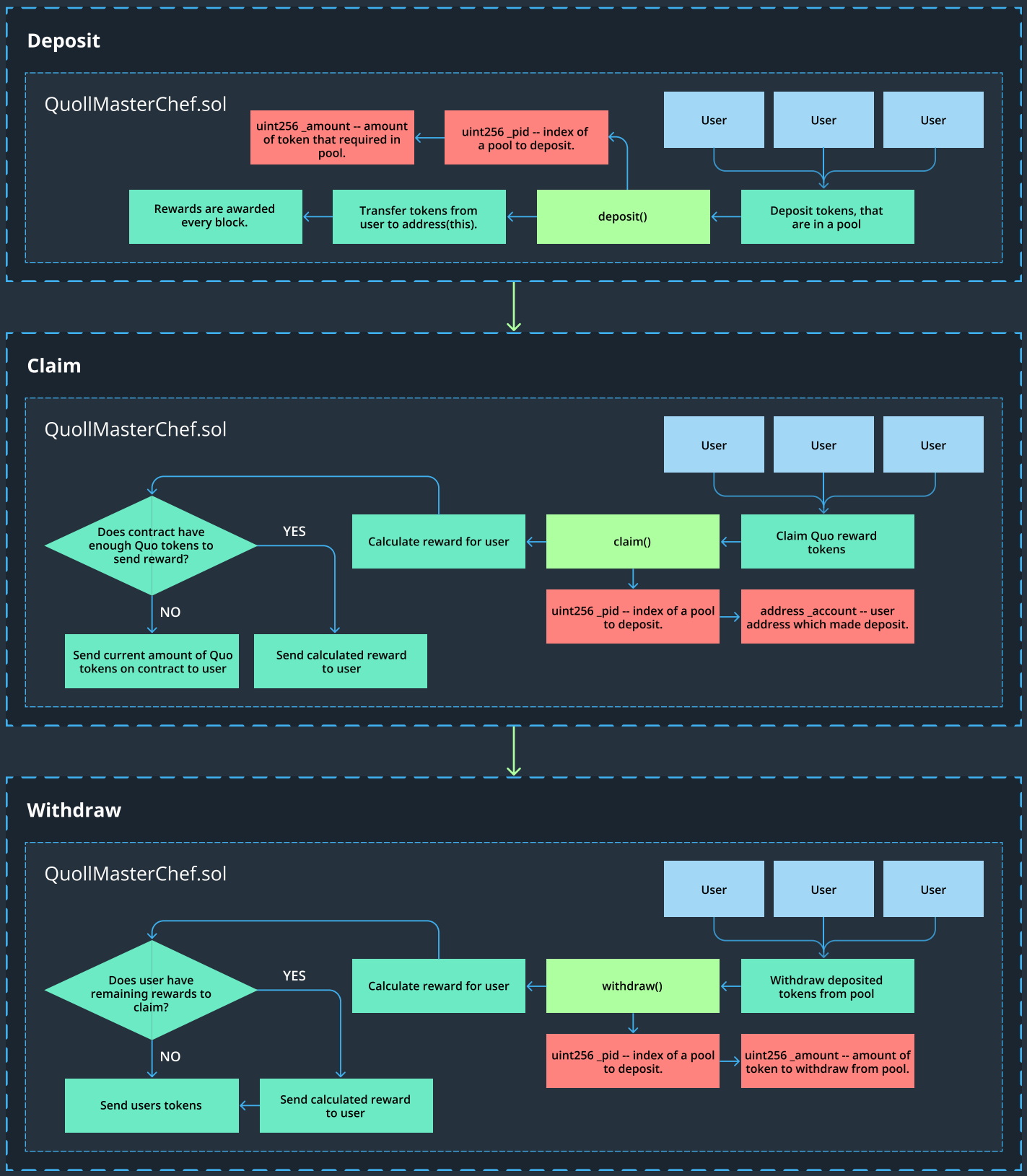

Zap in flow
User
QuollZap.sol
Provide underlying token to add liquidity in Wombat Exchange and stake corresponding LP token in Wombat Booster.
uint256 _pid -- index of a pool to deposit.
uint256 _amount -- amount of token that required in pool.
uint256 _amount -- amount of token that required in pool.
zapIn()
`_stake` is true?
Yes
NO
Transfer underlying tokens from user to address(this).
Deposit received LP tokens to Wombat booster and receive Deposit tokens.
Stake received Deposit tokens on behalf of user in reward pool.
Transfer received Deposit tokens to user.
Deposit underlying tokens to Wombat exchange and receive LP tokens.
stakeFor()
deposit()
BaseRewardPool.sol
WombatBooster.sol
Wombat Exchange protocol


Zap out flow
User
QuollZap.sol
Provide a deposit token from own balance or unstake from reward pool.
zapOut()
uint256 _pid -- ID of a pool in Wombat booster for an LP token, corresponded to underlying token, that user desires to withdraw.
uint256 _amount -- amount of deposit token to withdraw.
bool _stake -- a flag, which indicates whether to additionaly unstake deposit tokens from reward pool.
`_stake` is true?
NO
YES
Unstake deposit tokens from reward pool.
Transfer deposit tokens from msg.sender to addres(this).
Withdraw LP token from Wombat Booster in exchange for deposit tokens.
Remove liquidity from Wombat exchange. Burn LP tokens in exchange for underlying tokens.
Transfer received underlying tokens to user.
withdrawFor()
withdraw()
withdraw()
BaseRewardPool.sol
WombatBooster.sol
Wombat Exchange protocol

VlQuo


Lock
VlQuo.sol
User
User
User
Lock QUO for some weeks to receive voting powers and earn rewards.
lock()
Transfer QUO from user to address(this).
Increase voting power for each week, during each user is locking QUO.
Define at what week tokens will be unlocked and store it in storage.
address _user -- address of an account, for whom to lock QUO.
uint256 _amount -- amount of QUO to stake.
uint256 _weeks -- number of weeks, for which to lock QUO.
Distribute rewards
Rewards are queued by authorized account for current week.
Queued rewards are distributed among users. who have locks for current week.
Collect rewards
VlQuo.sol
User
User
User
Claim reward tokens.
getReward()
Calculate earned rewards.
address _account -- address of a user, for whom to claim rewards.
Token == qWOM AND User wants to stake?
YES
NO
Transfer earned rewards to user.
BaseRewardPool.sol
Stake claimed qWOM for user in BaseRewardPool contract.
Unlock
VlQuo.sol
User
User
User
Unlock QUO tokens.
unlock()
Calculate unlockable rewards during the time since last unlock week till the current week.
Transfer unlocked QUO to user.
Update user's last unlock week.

QuollMasterChef



STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. 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 it can lead to a significant loss, funds may be lost or allocated incorrectly.

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

HIGH-1 | RESOLVED

Withdrawals might get blocked

BaseRewardPool.sol, QuoRewardPool.sol: _withdraw().
During the withdrawal of the deposited tokens, the rewards are transferred as well (Line 233 for BaseRewardPool.sol, line 219 for QuoRewardPool.sol). However, if there are not enough reward tokens on the contract's balance, the withdrawal transaction will revert, preventing users from withdrawing their funds. The issue is marked as high since, in combination with the Low-1 issue, such a scenario might be possible.

Recommendation:
Make sure that paying rewards doesn't prevent users from withdrawals in case there are not enough reward tokens to pay.

Post-audit.
A mandatory transfer of the rewards from msg.sender to the reward pool was added so that there are always enough reward tokens on the balance.

HIGH-2 | RESOLVED

Possible frontrun via flash loan attack.
QuollZap.sol: function zapIn(), line 69; function zapOut(), line 127.
There are `minimumLiqudity` and `minimumAmount` parameters in deposit() and withdraw() functions of the Wombat Pool contract. These parameters are essential to protect users from big slippage and possible manipulations of the pool's reserves via frontrun. Which is why, it is highly recommended not to ignore these parameters and put an actual value instead of 0.

Recommendation:
Provide `minimumLiquidity` and `minimumAmount` when interacting with the Wombat Pool to protect users.
Post-audit.
The minimum amount parameter is passed to zapIn() and zapOut() functions now. Though it is still up to the Quoll team to provide necessary validations at the frontend side.

ETH might get stuck on the contract's balance.

QuollZap.sol: function zapIn(), line 54.
BaseRewardPool.sol: function donate(), line 266.
In case `underlyingToken` is a wrapped token of a native coin, it is validated that `_amount` is equal to the msg.value. Otherwise, it is not validated that there is no ETH sent to the function. As a result, any ETH sent to the function when `underlyingToken` is not a wrapped token will get stuck. The same situation is possible in BaseRewardPool in case `_rewardToken` is not a native coin, but ETH was sent to the function.

Recommendation:
Validate that there is no ETH sent when `underlyingToken` is not a wrapped token for QuollZap.sol. For BaseRewardPool.sol, validate that no ETH is sent when `_rewardToken` is not a wrapped token.

LOW-1 | RESOLVED

Unclear way of how reward tokens are sent to the pool address.
1) BaseRewardPool.sol, QuoRewardPool.sol: function queueNewRewards()**.**
It is not validated that there are enough reward tokens on the contract's balance to distribute provided `_rewards` and the necessary amount is transferred from msg.sender's balance during the execution of the function. As a result, there might occur difficulties when paying rewards.
2) QuollMasterChef.sol
There is no explicit transfer of reward tokens to the contract in the QuollMasterChef.sol code. Thus, it should be validated how rewards are sent to the contract (for example, by calling the transfer() function on the reward token contract).

Recommendation:
Verify how reward tokens are sent to the contract's balance. Ensure that there will be enough rewards to pay users.

Post-audit.
A mandatory transfer was added to BaseRewardPool.sol. It was verified by the Quoll team that the rewards are sent directly to the contract's balance.

LOW-2 | RESOLVED

An LP token can be added more than once.

QuollMasterChef.sol: function add(), line 111.
There is no validation that the pool for `_lpToken` is already added. Usually, it is not recommended to create more than one pool in master-chief-like stakings for the same token. Though this function can only be called by the owner and there is a warning saying that the same LP token should not be added more than once, it is recommended to add a restriction in the code in order to validate that the same LP token can't be added more than once.

Recommendation:
Add a restriction so that the same LP token can't be added more than once.

LOW-3 | RESOLVED

Parameters lack validation.

1) QuollToken.sol: function setAccess().
Parameter `_operator` should be validated not to be equal to zero address.
2) DepositToken.sol: function initialize().
Parameter `_operator` should be validated not to be equal to zero address.
3) BaseRewardPool.sol: function setParams().
Parameters `_booster`, `_stakingToken` should be validated not to be equal to zero addresses.
4) BaseRewardPool.sol, QuoRewardPool.sol: function addRewardToken().
Parameter `_rewardToken` should be validated not to be equal to zero address.
5) BaseRewardPool.sol, QuoRewardPool.sol: function stakeFor().
Parameter `_for` should be validated not to be equal to zero address.
6) BaseRewardPool.sol, QuoRewardPool.sol: functions grant(), setAccess().
Parameter `_address` should be validated not to be equal to zero address.
7) QuoRewardPool.sol, WombatBooster.sol, WombatVoterProxy.sol, WomDepositor.sol, QuollZap.sol,VlQuo.sol: setParams().
All function parameters should be validated not to be equal to zero addresses.
8) QuollMasterChef.sol: function setParams().
Parameter `_quo` should be validated not to be equal to zero address. Parameter `_startBlock` should be validated to be greater than current block.timestamp.
Parameter `_bonusEndBlock` should be validated to be greater than `_startBlock`.
9) QuollMasterChef.sol: function add().
Parameter `_lpToken` should be validated not to be equal to zero address.
10) VlQuo.sol: function lock.
Parameter `_user` should be validated not to be equal to zero address.
It is recommended to validate some of the function parameters before setting them in storage or executing further function code. For instance, parameters should be validated in functions such as setParams(), which can only be called once.

Recommendation:
Validate functions parameters.

Post-audit. All fixes were applied except point 8. According to the team, the validation of _startBlock and _bonusEndBLock is not necessary.

INFO-1 | VERIFIED

No reduction is applied until the total supply is equal to `reductionPerCli `.

QuollToken.sol
During the initialization of the contract, `reductionPerCli` is assigned to 1e24. (Line 31). Until the total supply of the token is equal to this value, the result of the division in line 56 will be equal to zero, and there won't be any reduction applied to the `_amount` parameter. The issue is marked as info in case this is an intended functionality.

Recommendation:
Verify that the cliff should be equal to 0 until the total supply is equal to 1e24.

From the client.
The Quoll team ensured that 50% of the total supply will be minted during the first mint.
Yet, the auditors team insists on adding transparent deployment scripts since there is no way to verify the deployment (and minting) process for now.

INFO-2 | VERIFIED

The operator can burn tokens from any addresses.

DepositToken.sol, QuollExternalToken.sol: function burn().
The address of the operator is set during the creation of tokens. Additionally, the owner of the QuollExternalToken can change the address of the operator with the setOperator() function. The operator can burn tokens from any account without the approval of the account. Though such functionality is an intended business logic, it should be mentioned in the audit report and verified by the Quoll team.

Post-audit
The Quoll team ensured that such functionality is intended and the only admin role will be delegated to the responsible contract. Yet, the issue is still present in the report due to the upgradeability of the contracts and the absence of deployment scripts to verify the role delegation process.

INFO-3 | RESOLVED

Call to an arbitrary address with the arbitrary data.

WombatVoterProxy.sol: function execute().
There is a call performed to an arbitrary address with an arbitrary calldata. Such a call might lead to unpredictable consequences in case it is performed to a malicious contract.

Recommendation:
Verify the `_to` parameter, to which call is performed. Consider adding a whitelist of the addresses that can be called.
Post-audit. The function was removed.

INFO-4 | RESOLVED

Contract's function can't be called.

WombatVoterProxy.sol: function execute().
There is a modifier on the execute() function, which verifies that msg.sender is a booster contract. However, the call of this function is not implemented in WombatBooster.sol. Therefore, currently, the function can't be executed. The issue is marked as informational since both contracts are upgradable and the implementation of the call can be added later.

Recommendation:
Add the execution of the execute() function inWombatBooster.sol.
Post-audit. The function was removed.

INFO-5 | UNRESOLVED

Storage constant should be used.

QuollMasterChef.sol
The `1e12` value is used across the contract multiple times (line 180, 183, 212, 224, 235, 258, 263, 287, 291). In order to increase the contract's readability, it is recommended to avoid the usage of numeric values directly and use a storage constant instead.

Recommendation:
Use storage constant.

INFO-6 | RESOLVED

Commented code.

QuollMasterChef.sol: function updatePool(), line 210.
Pre-production contracts should not contain commented or unimplemented logic.

Recommendation:
Remove commented code or finish the contract's logic commented to this code.

INFO-7 | RESOLVED

The wrong parameter is passed in the event.

QuoRewardPool.sol: function stakeFor(), line 204.
The msg.sender parameter is passed to the "Staked" event instead of the `_for` parameter.

Recommendation:
Pass the correct parameter to the event.

INFO-8 | VERIFED

Users might not receive rewards.

QuollMasterChef.sol: function safeRewardTransfer().
In case the contract doesn't have enough rewards on its balance, users will be paid as many tokens as possible, and it will be marked as if they received a full reward. Though such behavior is standard for Master-Chief-like staking, such a feature should be noted and verified by the Quoll team.

Recommendation:
Verify that users should not receive full reward in case there is not enough reward token on the contract's balance.

From the client.
The Quoll team ensured to keep track of an actual amount of rewards on the contract so that there are always enough reward tokens.

INFO-9 | UNRESOLVED

Outdated Solidity version.

Currently, the contracts use 0.6.0 Solidity version, which is considered deprecated. It is highly recommended to use the latest version of Solidity, which contains a lot of new features and bug fixes.
The issue is marked as Medium for several reasons:

- floating version is used ^0.6.0, not fixed
- there are no deploy scripts or Hardhat project files or a package file to verify that the last version will be used during the deployment
- it is recommended to use the latest stable version, which is 0.8.17.

Thus, using an outdated version (in combination with absent periphery scripts for the project) leaves the possibility of exploits in old compiler version, the possibility to use the version of the compiler with unfixed bugs by human mistake, and usage of unoptimized code (since latest versions provide several layers of optimizations).

Recommendation:
Update Solidity version to 0.8.17, provide Hardhat project files and deployment scripts

Post-audit:
Quoll team has updated Solidity version up to 0.6.12 which is the latest in the branch. While it is still considered as outdated, it is acceptable to have that version in order to decrease criticallity from Medium to Info.
Though, the project still lacks deployment scripts and Hardhat project structure in order to verify correctness of the deployment process.

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

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

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

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

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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

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

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

Contract: BaseRewardPool
User Interaction
✓ stake tokens
✓ withdraw tokens
✓ get rewards tokens
✓ donate tokens
Reverts
✓ If try to initialize again
✓ if msg.sender is not operator
✓ if params has been set
✓ if stake amount == 0
✓ if withdraw amount == 0
✓ if donate wrong token
✓ if do not have access to queue rewards
Contract: QuoRewardPool
Init
✓ sets reward token correctly
User Interaction
✓ stake tokens (63ms)
✓ withdraw tokens (62ms)
✓ get rewards tokens (146ms)
✓ donate tokens
Reverts
✓ If try to initialize again
✓ if msg.sender is not owner
✓ if params has been set
✓ if stake amount == 0
✓ if withdraw amount == 0
✓ if donate wrong token
✓ if do not have access to queue rewards

Contract: Quoll External Token

Init

✓ set operator to user

✓ mint/burn tokens

Reverts

✓ If try to initialize again

✓ if user is not authorized to mint/burn

✓ if caller is not owner

Contract: Quoll Master Chef

Init

✓ set params correctly

✓ add and update reward pool

Users interactions

✓ deposit to pool (81ms)

✓ withdraw/emergency withdraw from pool (92ms)

Reverts

✓ If try to initialize again

✓ If quo address == 0x0

✓ If caller is now owner

✓ no amount to withdraw

Contract: Quoll Token

Init

✓ set access to user

✓ mint tokens

Reverts

✓ If try to initialize again

✓ if user is not authorized to mint

✓ if caller is not owner

Contract: Quoll Zap

Zap in/out

✓ Should zap in underlying tokens in exchange of corresponding LP token (6013ms)

✓ Should zap in underlying tokens and stake deposit tokens (115ms)

✓ Should zap out deposit tokens in exchange of underlying tokens (4278ms)

✓ Should zap out staked deposit tokens in exchange of underlying tokens (219ms)

✓ Should rescue stuck tokens (413ms)

✓ Should zap in and out native coin (11493ms)

✓ Should not zap in native coin if amount != msg.value

Reverts

✓ If try to initialize again

✓ if caller is not owner

Contract: VlQuo

✓ Should lock QUO tokens for several weeks (51ms)
✓ Should lock QUO tokens more than one time (92ms)
✓ Should let lock on behalf of user (69ms)
✓ Should not let lock on behalf of other user if user forbade third party actions
✓ Should not let lock if number of weeks is 0
✓ Should not let lock if number of weeks exceeds maximum allowed
✓ Should not let lock if amount is 0
✓ Should unlock unlockable tokens (79ms)
✓ Should not unlock tokens from same week more than once (85ms)
✓ Should extend lock for full amount (130ms)
✓ Should extend lock for partial amount (197ms)
✓ Should not extend lock if weeks number is 0 (75ms)
✓ Should not extend lock if new weeks number exceeds maximum allowed (40ms)
✓ Should not extend lock if number of weeks is greater than new number of weeks (38ms)
✓ Should not extend lock if amount is 0 (39ms)
✓ Should not extend lock if amount is invalid (45ms)
✓ Should queue new rewards for current week (45ms)
✓ Should not queue rewards for current week if caller has no access
✓ Should queue rewards to be distributed later if current week total weight is 0
✓ Should donate rewards (65ms)
✓ Should queue donated rewards (63ms)
✓ Should not donate not reward token
✓ Should get reward for user (55ms)
✓ Should not let get reward if user hasn't locked (50ms)
✓ Should not let user claim same reward more than once (88ms)
✓ Should split reward correctly among users (177ms)
✓ Should stake collected qWOM rewards (62ms)
✓ Should collect to balance collected qWOM rewards (52ms)
✓ Should calculate 0 rewards if user hasn't locked (43ms)
✓ Should not let set params more than once
✓ Should return user's weight for current week (85ms)
✓ Should return user's current unlock amount
✓ Should get user's active locks (111ms)
✓ Should return zero unlockable if user hasn't locked
✓ Should return reward tokens number

Contract: WomDepositor

✓ Should deposit WOM and receive qWOM (509ms)
✓ Should deposit WOM and stake received qWOM (41ms)
✓ Should deposit all WOM
✓ Should not let deposit 0 WOM
✓ Should set new lock interval

```
Contract: WombatBooster
✓ Should deposit Wombat LP tokens (2780ms)
✓ Should deposit Wombat LP tokens with stake (65ms)
✓ Should deposit all Wombat LP tokens (45ms)
✓ Should withdraw deposit tokens (1714ms)
✓ Should withdraw all deposit tokens (127ms)
✓ Should mint QUO for claimed WOM (310ms)
✓ Should shutdown system (1873ms)
✓ Should shutdown pool (84ms)
✓ Should not let not reward pool execute rewardClaimed()
✓ Should not let set params more than once
✓ Should set new fees
✓ Should not set fees if total fee exceeds limit
✓ Should not set invalid fees
✓ Should set treasury
✓ Should get number of pools
✓ Should send rewards only to base reward pool and qWom reward pool if other fees are
set as 0 (313ms)
Contract: WombatVoterProxy
✓ Should not let set params more than once
✓ Should not let call booster functions non-booster accounts
✓ Should not let not depositor lock

109 passing (2m)
```

Note: there was no original test coverage provided by the Quoll team. Thus, all tests are prepared by the Zokyo Security team.

FILE	% STMTS	% BRANCH	% FUNCS
BaseRewardPool.sol	100	90	100
DepositToken.sol	100	50	100
QuoRewardPool.sol	100	88.89	100
QuollExternalToken.sol	100	100	100
QuollMasterChef.sol	89.76	75	100
QuollToken.sol	100	100	100
QuollZap.sol	100	94.44	100
VlQuo.sol	100	92.19	100
WomDepositor.sol	96	70	75
WombatBooster.sol	99.01	76.56	100
WombatVoterProxy.sol	89.09	73.53	80
All files	**97.62**	**82.78**	**95.91**

Zokyo Security has prepared a set of unit-tests on the BNB Smart Chain fork in order to test the contracts in conditions close to the production. Unfortunately, the part of the logic connected to lock/unlock functionality of WombatVoterProxy.sol could not be tested with the unit tests due to the whitelist functionality on the Wombat Exchange protocol. The correctness of this part of the code was validated during the manual audit and by manual tests performed by the team of auditors.

We are grateful for the opportunity to work with the Quoll Finance 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 Quoll Finance team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

ZOKYO.