Blaize.Security

September 1st, 2023 / V. 1.0

AraFi

ARAFI
SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **8.3**/10

Protocol's security is evaluated as **sufficiently secure**.

The **scope** of the project includes AraFi set of contracts:
contracts\aHUM\aHUM.sol
contracts\Fractals\AraEmissionDistributor.sol
contracts\Fractals\AraFractal.sol
contracts\Governance\AraLocker.sol
contracts\MasterChef\AraMasterChef.sol
contracts\MasterChef\AraMasterChefMultiReward.sol
contracts\Migrator\AraMigrator.sol
contracts\Migrator\FractalMigrator.sol
contracts\Strategy\HummusV2Strategy.sol
contracts\Strategy\HummusV3Strategy.sol
contracts\AraFiToken.sol

Repository:

https://github.com/aerariumfinance/arafi-v1-contracts

Branch: main

Initial commit:

- 582ca33b1cdb8a81d5d563deded83e537b4dbe58

Final commit:

- 4dd40b992a6d1acb65d7e19e4198c1ee916f095a

The code of the core protocol has later migrated into new repository:
https://github.com/AraFiEco/arafi-v1-contracts

The scope of the project includes Aerarium set of contracts:

contracts\LibOptions.sol
contracts\StakeWrapper.sol
contracts\TokenWrapper.sol
contracts\VeWrapper.sol

Repository:

https://github.com/aerariumfinance/arafi-v1-wrappers

Branch: main

Initial commit:

- c1d7ca3ab927d258c58eee89c45093d28a134a7e

Final commit:

- 37c421de924b818b9059cbeeedafa8c7c6f6802b

On the final stage of the audit, the code base migrated to:

https://github.com/AraFiEco/arafi-v1-contracts

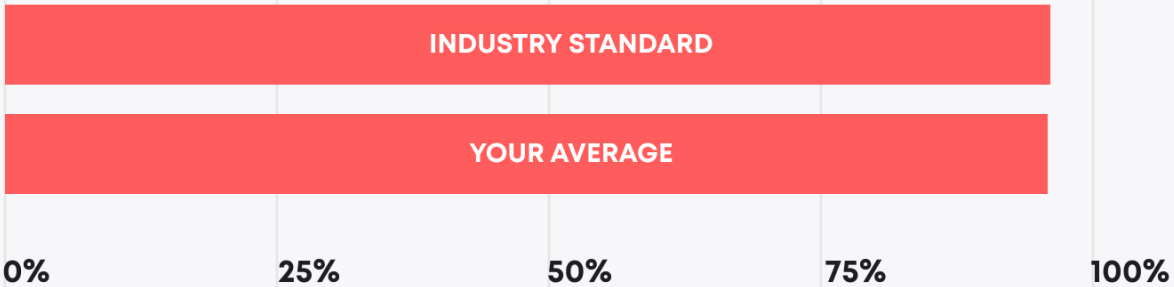
Branch: master
Final commit:

- 4dd40b992a6d1acb65d7e19e4198c1ee916f095a

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the AraFi protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **AraFi** smart contracts conducted between **May 16th, 2023** and **August 29th, 2023.**

Testable code



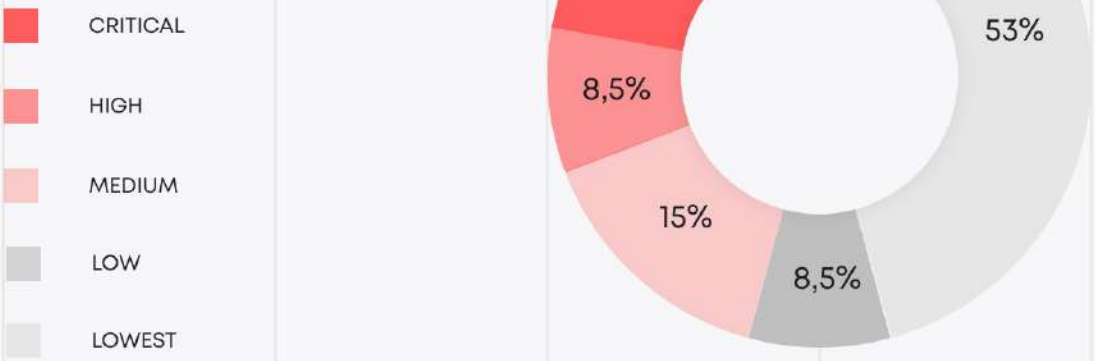
The testable code is sufficient to correspond the industry standard.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **AraFi** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **AraFi** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 47 problems were found. 40 issues were fixed or verified by the AraFi team.

	FOUND	FIXED/VERIFIED
Critical	7	7
High	4	4
Medium	7	7
Low	4	4
Lowest	25	18

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

Blaize.Security auditors start the audit by developing an **auditing strategy** - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

Manual audit stage

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

For advanced components:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

Testing stage:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

EXECUTIVE SUMMARY

Blaize Security has audited the AraFi Protocol, which consists of smart contracts for yield farming, master chefs for farming rewards, strategies based on the Hummus protocol, and an NFT for the voting process.

The audit aimed to examine the smart contracts for common vulnerabilities and the auditors' internal checklist, ensuring the security of users' funds involving LP tokens, HUM, and reward tokens, verify the accuracy of interaction with the Hummus protocol, and check the proper implementation of the business logic of the smart contracts.

The audit revealed various issues of differing severity, including some critical ones. The critical issues involved incorrect usage of storage pointers, improper loop iteration, problems with the compilation of smart contracts, incorrect distribution of reward tokens and inconsistent NFT ownership checks. The AraFi team has fixed all of the critical issues. The Complete Analysis section provides a full list of all the issues found.

High-risk issues included total allocation points not being updated, uninitialized rewardPerBlock mapping, wrong denominator usage, and lack of checks for duplicate LP token pools in contracts. All these issues were resolved. Medium issues involved transfers not being validated, transfer validated with assert, an array may contain empty elements, any user can deposit arbitrary new tokens, ERC721 is treated like a fungible token, harvester can add new harvesters, and users could get less Reward tokens. All these issues were resolved. Low issues included unused parameters and variables, same function name, redundant access checks, only pool with ID 0 can be set, unused public visibility, asserts are used for validating inputs, duplicating require-statements, and require message mismatch with code. All these issues were resolved.

Finally, the lowest issues included inaccurate version pragma, use of SafeMath library, lack of validation, comments mismatch with code, redundant initialization, assert are used for validating inputs, duplicating require-statements, require message mismatch with code, redundant calculation of multiplier, rewards distribution is finished on V2, wrong pool id is passed in the event and absence of pool id in the event, specific order of functions, and inconsistent reward transfer conditions in contracts. Some of these issues were resolved while others remain unresolved.

Blaize Security has prepared unit tests on the fork of the Metis network to verify the correctness of interaction with the Hummus protocol. Additional tests were written to check the general validity of the protocol. However, not all new features or fixes were tested due to extended timelines. The protocol seems to have sufficient security overall.

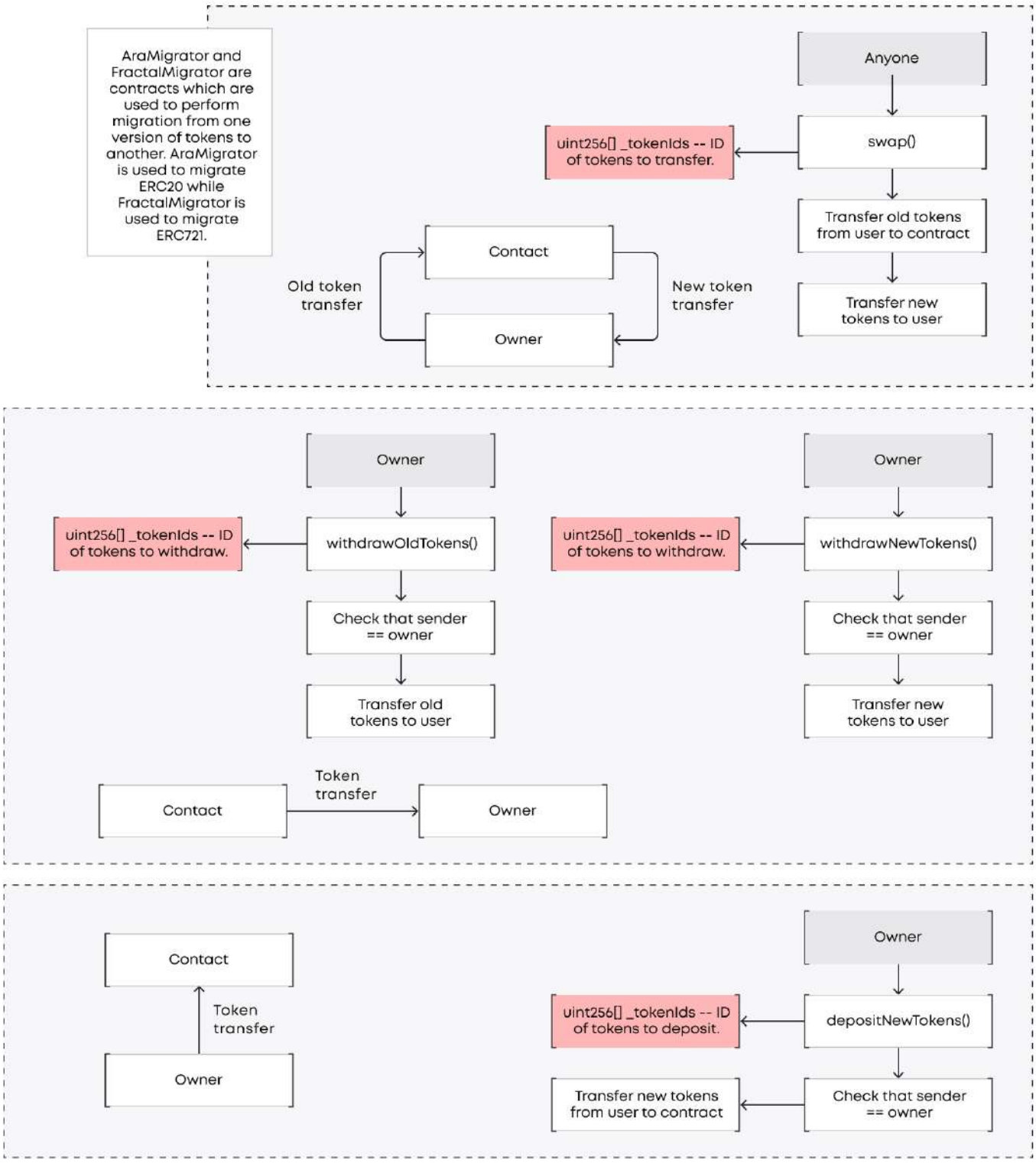
The contracts safely interact with users' funds and have necessary checks but lack the NatSpec documentation. In the process of re-audit, some new issues came up such as removal of necessary modifier or changing logic around aHUM strategy and emergency withdrawal. Thes changes are reflected in the lowest issues. The AraFi team has also removed the auto migration of funds during of replacing of strategies.

Despite most issues resolved and general status as "sufficiently secure", still there are factors which influence the final mark: absent documentation (so it is hard to compare implementation with desired logic), absent tests (so it is impossible to verify the integrity and secure further development), code quality with place for improvement, several unresolved issues (some with questions to business logic), unstructured delivery of patches and rapid logic changes). Therefore we recommend another round of crossreview and code quality update with best practices following and documentation.

	RATING
Security	8.5
Gas usage and logic optimization	9
Code quality	7
Test coverage	9
Total	8.3

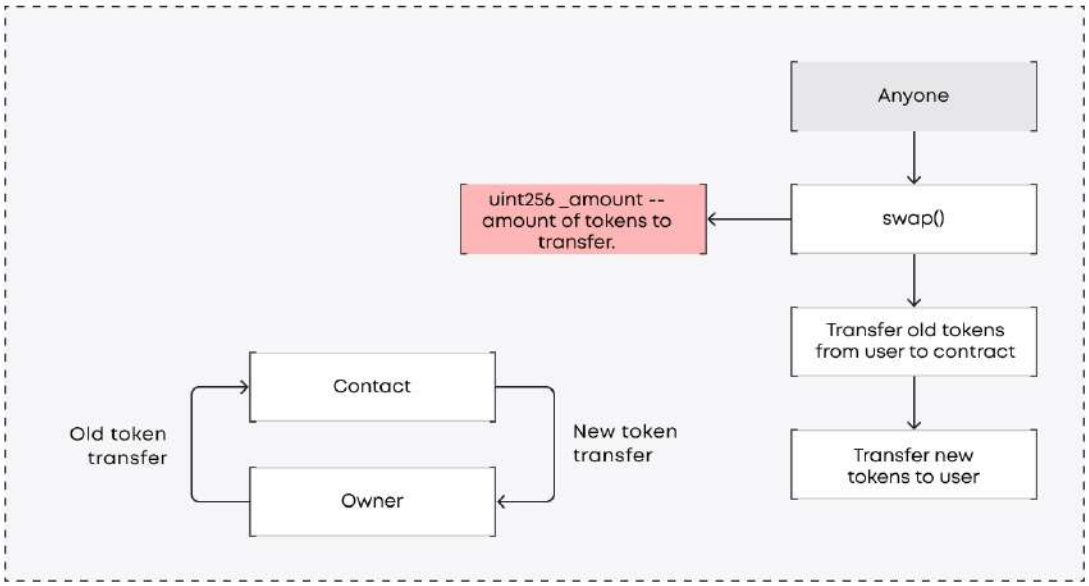
ARAFI SCHEME

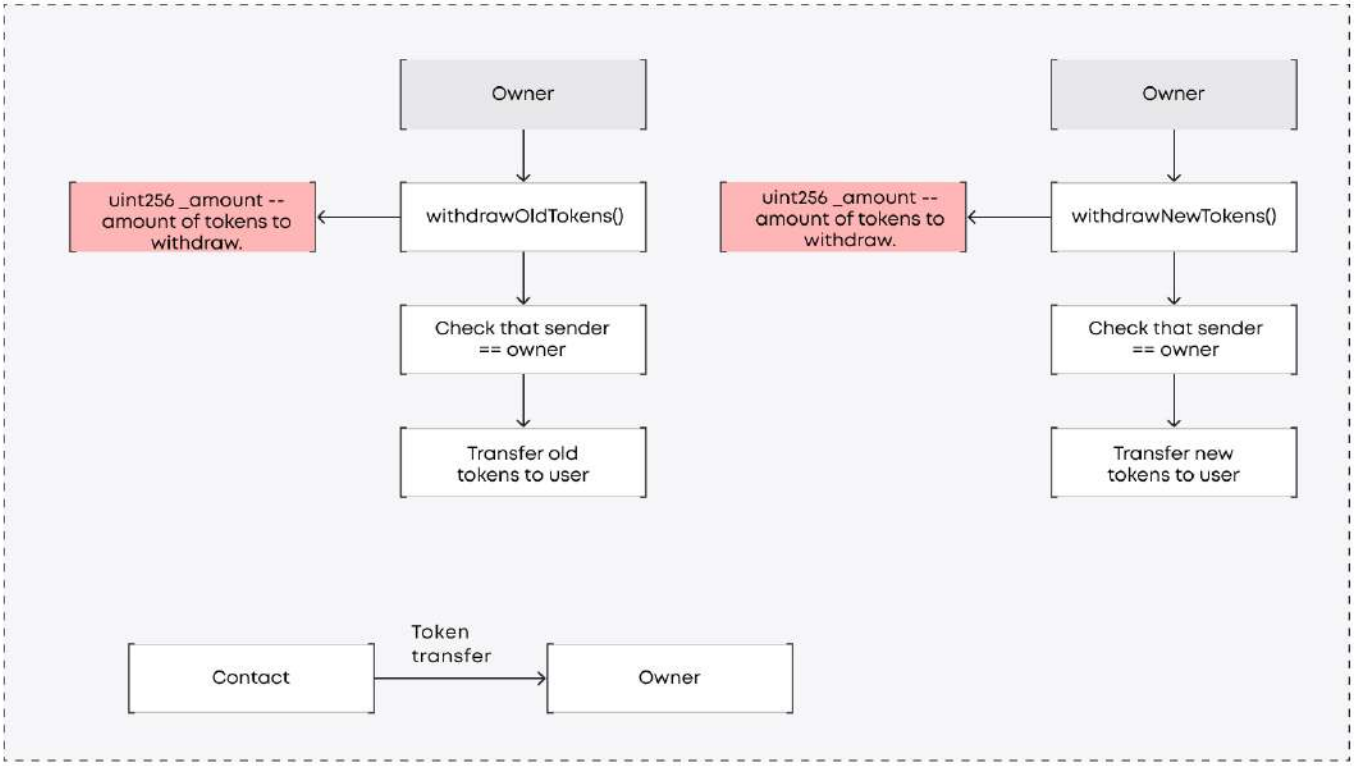
FractalMigrator.sol



ARAFI SCHEME

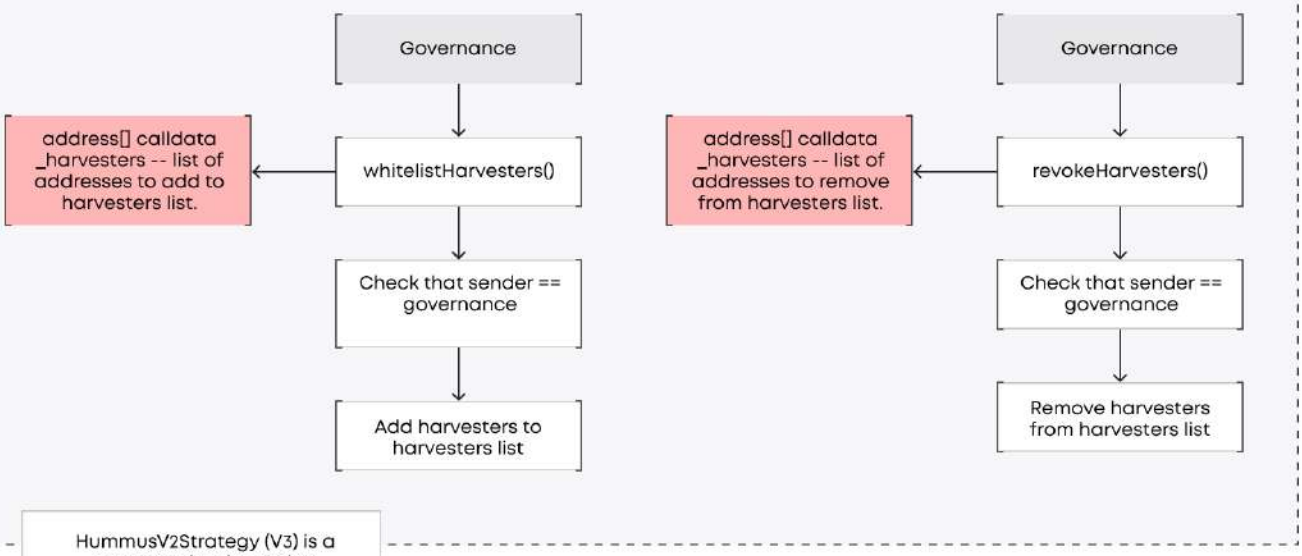
AraMigrator.sol



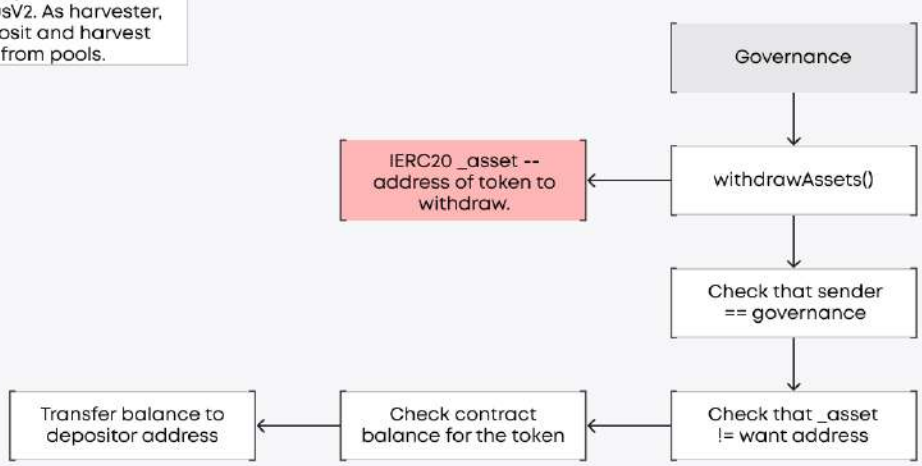


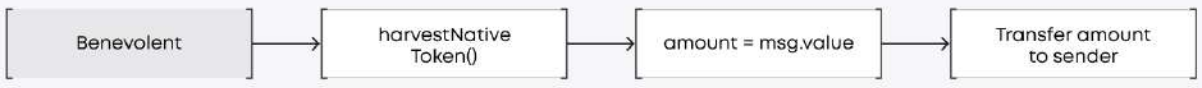
ARAFI SCHEME

HummusStrategy.sol (V2 and V3)



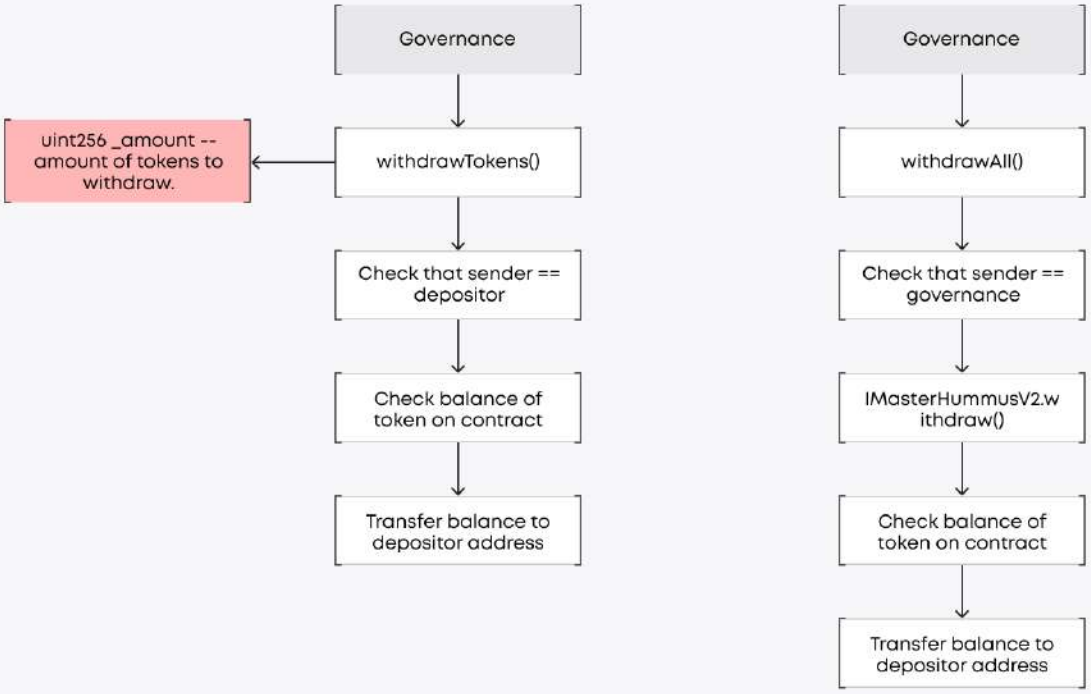
HummusV2Strategy (V3) is a contract that is used to manage pools on MasterHummusV2. As harvester, user can deposit and harvest rewards from pools.

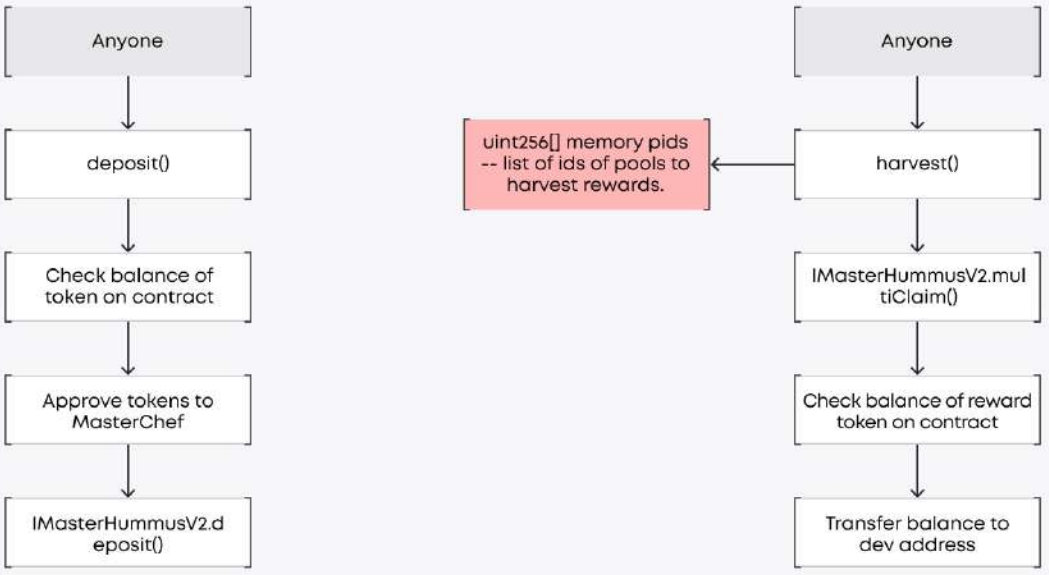




ARAFI SCHEME

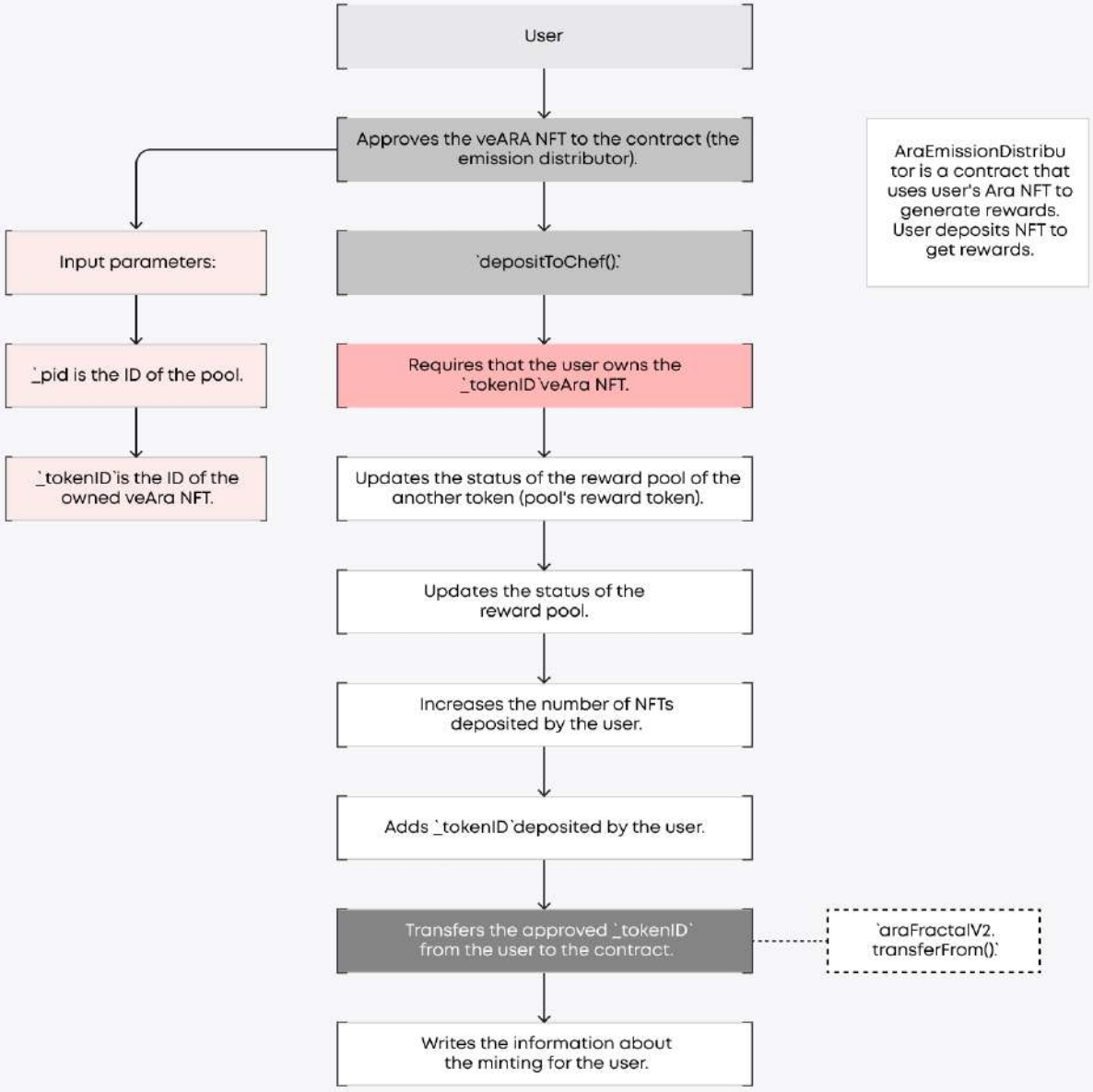
HummusStrategy.sol (V2 and V3)





ARAFI SCHEME

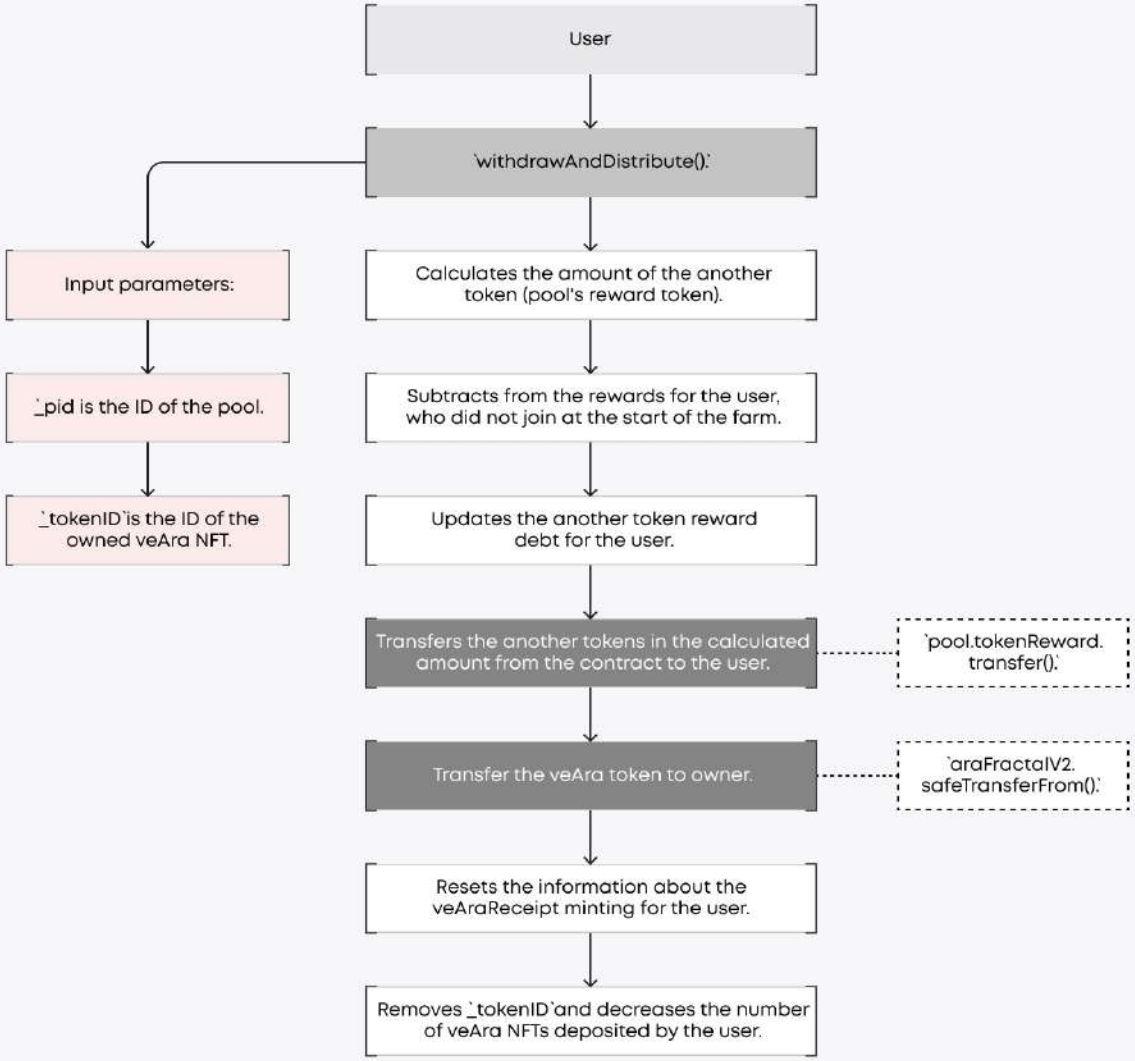
AraEmissionDistributor.sol



ARAFI SCHEME

AraEmissionDistributor.sol

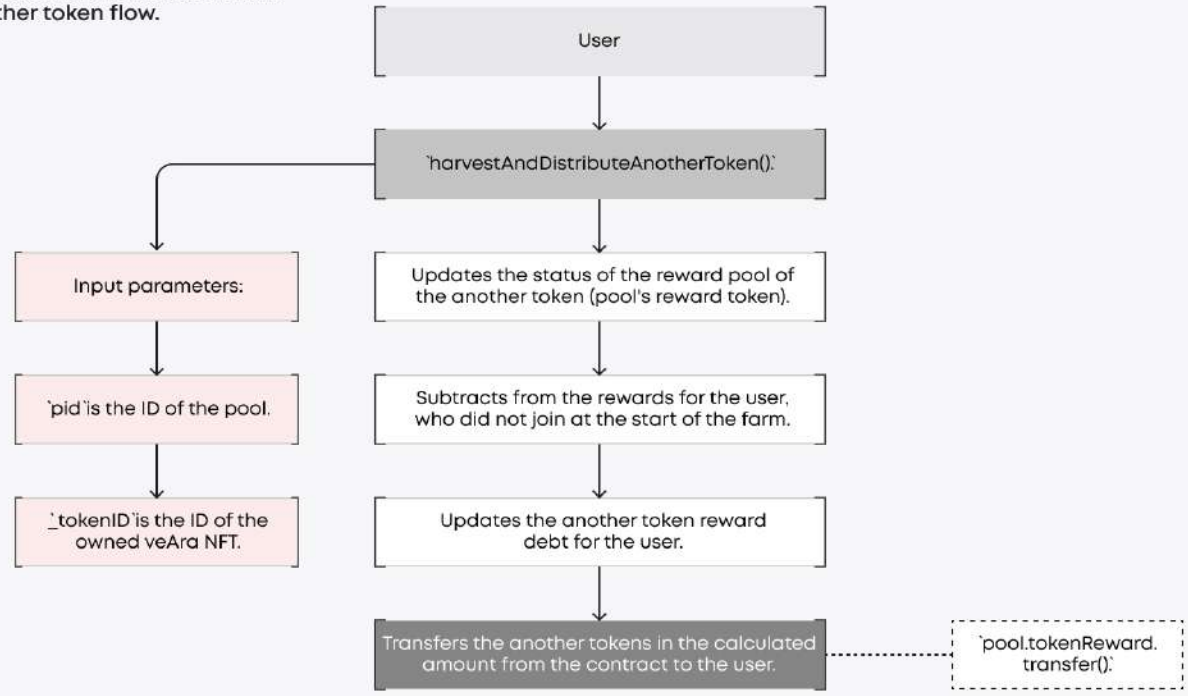
Withdrawal and distribution flow.



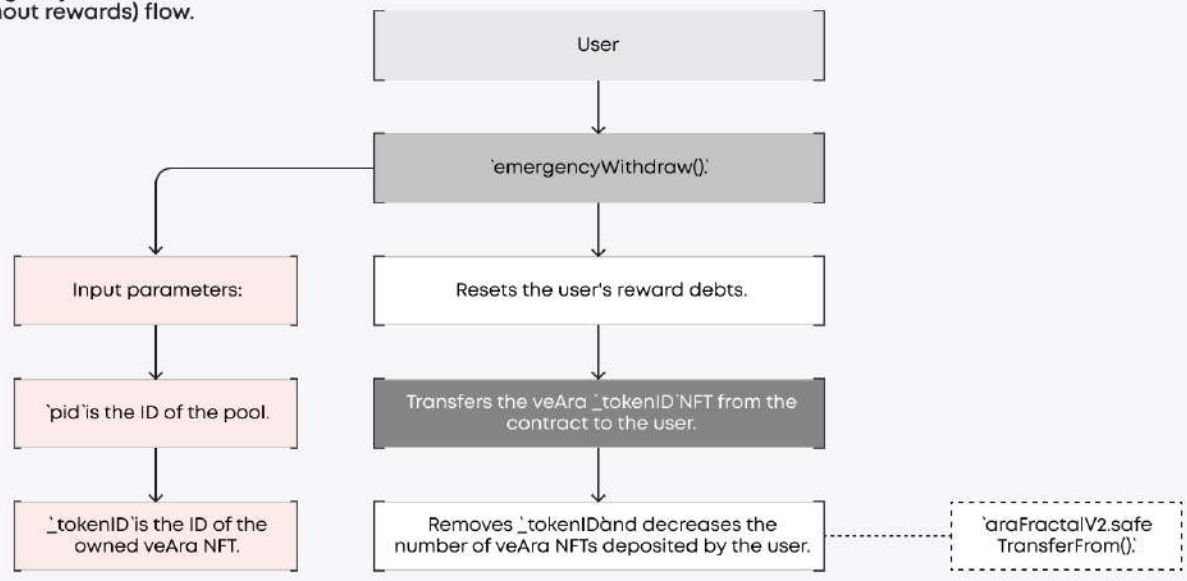
ARAFI SCHEME

AraEmissionDistributor.sol

Harvest and distribution of the another token flow.



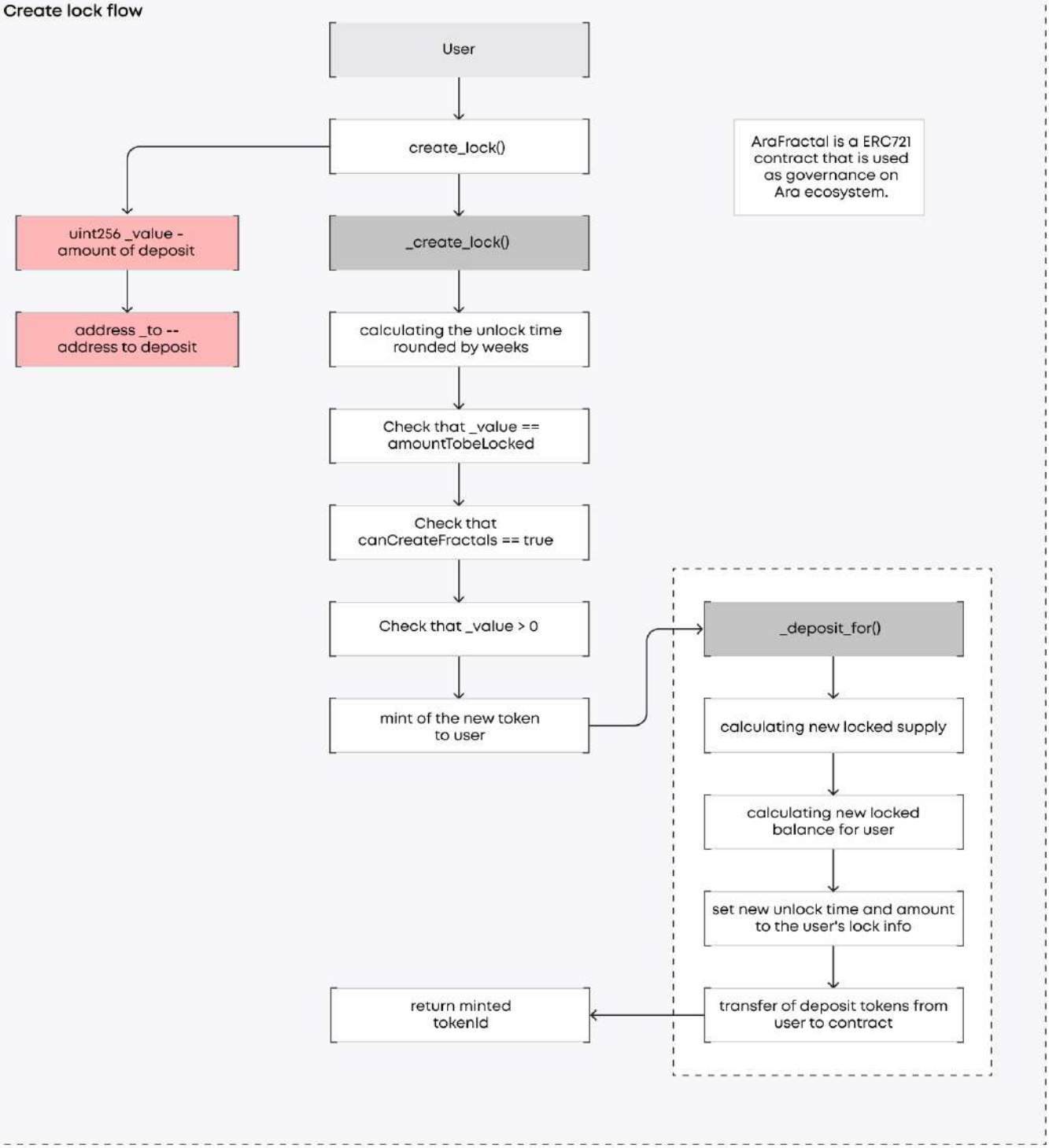
Emergency withdrawal (without rewards) flow.



ARAFI SCHEME

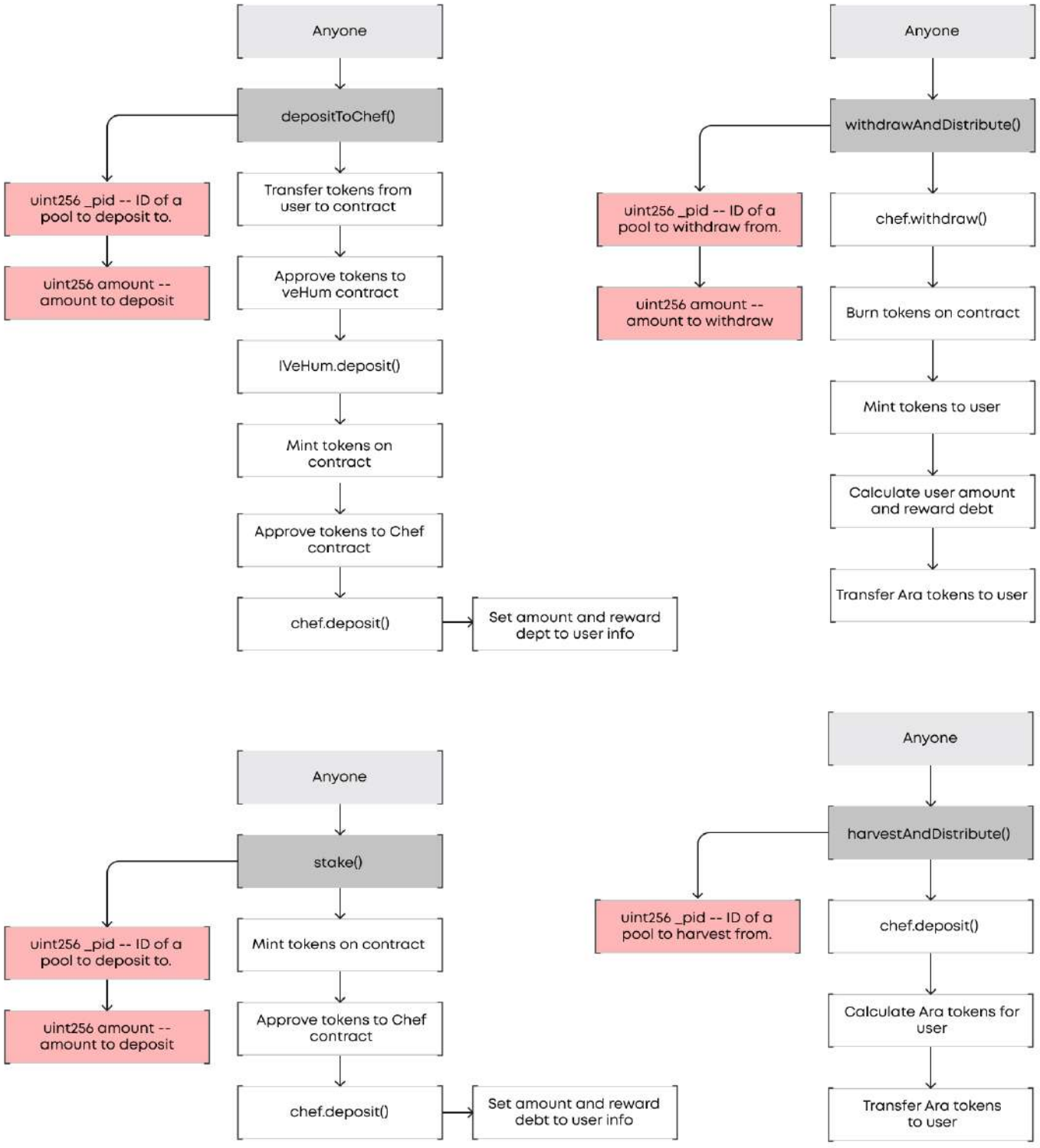
AraFractal.sol

Create lock flow



ARAFI SCHEME

aHUM.sol



ARAFI SCHEME

ARAFI WRAPPERS

TokenWrapper.sol

Description

The TokenWrapper contract is a Solidity smart contract that provides a framework for wrapping an ERC20 token with additional functionality. It inherits from multiple modules, such as Initializer, ReentrancyGuard, ERC20Facet, AccessControlFacet, PausableFacet, and SimpleBlacklistFacet. The contract allows users to deposit tokens, which are then processed according to the implemented logic. It also supports setting and updating options like treasury, fee, and rate, while ensuring proper access control, pausing, and blacklist enforcement.

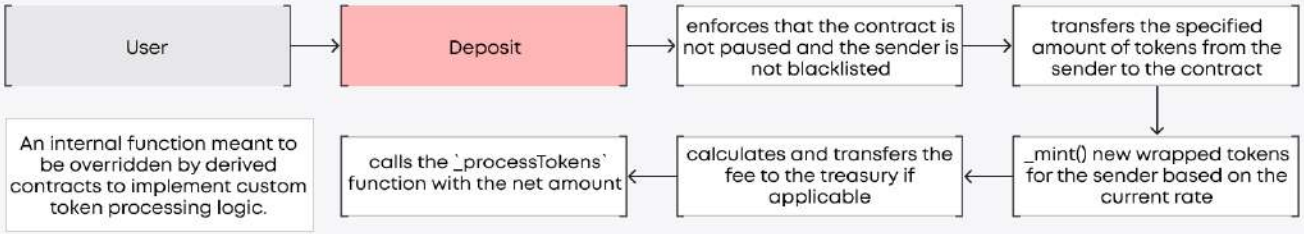
Owner

constructor()	Initializes the contract with the given ERC20 token address
_mint()	Mints new wrapped tokens for the specified account based on the given amount and rate.
setRate()	
setTreasury()	
setFee()	

Getters

getOptions()	Returns the current options (treasury, fee, and rate) from the contract's storage

User



LibOptions.sol

Description

The LibOptions library is a library that provides utility functions for managing the storage of options such as treasury, fee, and rate. It allows contracts to easily access and modify these values in a structured way.

Setters

- setRate()
- setTreasury()
- setFee()

Getters

- getStorage()
- getFee()
- getRate()
- getTreasury()

ARAFI SCHEME

ARAFI WRAPPERS

StakeWrapper.sol

Description

The StakeWrapper contract is a Solidity smart contract that extends the TokenWrapper contract, allowing users to deposit an ERC20 token and automatically stake the deposited tokens using a specified stake token contract. The contract supports initializing with custom parameters such as name, symbol, treasury, rate, and fee. It also overrides the `_processTokens` function to stake the deposited tokens.

Owner

constructor()	Inherits from the TokenWrapper contract and initializes the contract with the given ERC20 token and stake token addresses.
initialize()	- Grants the DEFAULT_ADMIN_ROLE to the specified owner. - Sets the name and symbol for the wrapped token using LibERC20. - Sets the treasury, rate, and fee values using LibOptions. - Approves the stake token contract to transfer the maximum possible amount of the wrapped token.
_processTokens()	- Overrides the `_processTokens` function from the TokenWrapper contract. - Calls the `stake` function of the stake token contract, staking the specified amount of tokens on behalf of the treasury.

VeWrapper.sol

Description

The VeWrapper contract is a Solidity smart contract that extends the TokenWrapper contract, allowing users to deposit an ERC20 token and automatically merge received NFTs from a specified voting escrow collection with the protocol voting escrow token. The contract supports initializing with custom parameters such as name, symbol, treasury, rate, fee, and veRate. It also overrides the `_processTokens` function to deposit tokens for the protocol voting escrow token ID.

Owner

constructor()	Inherits from the TokenWrapper contract and initializes the contract with the given ERC20 token, voting escrow collection, and protocol voting escrow token ID.
initialize()	- Grants the DEFAULT_ADMIN_ROLE to the specified owner. - Sets the name and symbol for the wrapped token using LibERC20. - Sets the treasury, rate, fee, and veRate values. - Approves the voting escrow collection contract to transfer the maximum possible amount of the wrapped token.
setVeRate()	

Getters

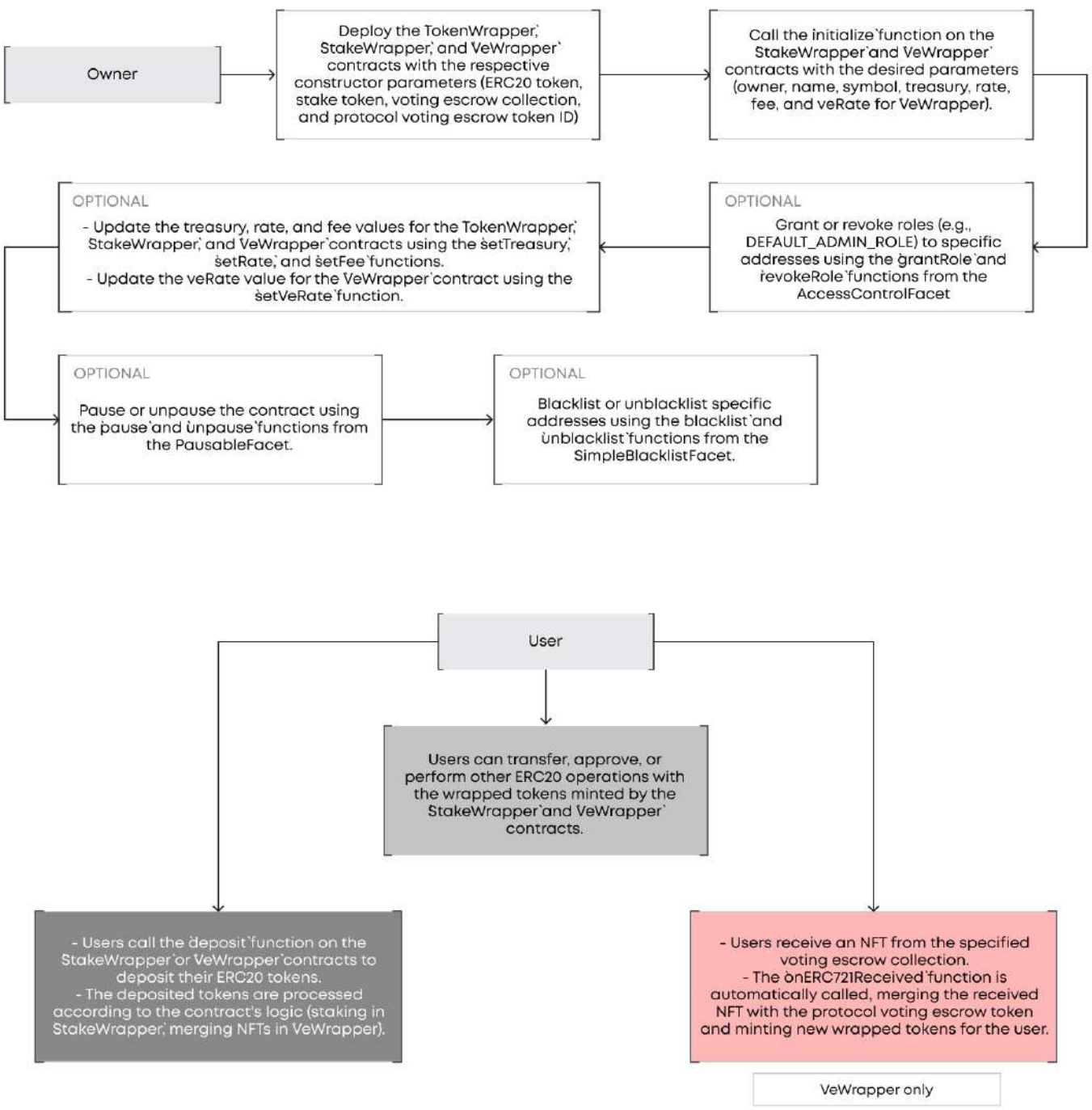
_getVeRateSlot()	Internal function that returns the storage slot for the veRate value.
getVeRate()	Returns the current veRate value from the contract's storage.

_processTokens()	- Overrides the `_processTokens` function from the TokenWrapper contract. - Calls the `deposit_for` function of the voting escrow collection contract, depositing the specified amount of tokens for the protocol voting escrow token ID.
onERC721Received()	- Enforces that the contract is not paused and the operator and sender are not blacklisted. - Checks if the received NFT is from the expected voting escrow collection, otherwise reverts with UnsupportedCollection error. - Retrieves the locked balance associated with the received NFT token ID. - Merges the received NFT with the protocol voting escrow token. - Mints new wrapped tokens for the operator based on the locked balance and the current veRate. - Emits a `DepositNft` event.

ARAFI SCHEME

ARAFI WRAPPERS

Flow of smart contracts



ARAFI SCHEME

ARAFI CONTRACTS

AraMasterChef.sol

Description

The AraMasterChef contract is a Solidity smart contract designed for managing liquidity pools and distributing rewards in the form of ARA tokens. Users can deposit LP tokens into the contract to earn ARA tokens, and the contract owner can add or update pools with different allocation points and deposit fees. The contract also supports integration with external strategies for optimizing yields and allows for updating emission rates and dev/fee addresses. Key features include:

1. Staking LP tokens to earn ARA rewards.
2. Adding and updating pools with different allocation points and deposit fees.
3. Integration with external yield-optimizing strategies.
4. Updating emission rates and dev/fee addresses.
5. Emergency withdrawal of LP tokens without rewards.
6. Safe ARA token transfer to prevent rounding errors.

Owner

constructor()	Setup reward token ara and it's emission rate, fee and dev addresses
add()	Add new pool
set()	Setup or update pool
updateEmmission Rate()	Update ara (reward token) per block
massHarvestFromStrategies(pids)	Call harvest on all strategies for provided pool ids
	Can be called by anyone

Getters

poolLength()	
getRewardMultiplier(from, to)	Return reward multiplier over the given _from to _to block.
pendingAra()	View function to see pending ARAs on frontend

Fee address

setFeeAddress()	Commissions are sent to the fee address

Dev address

dev()	Update dev address by the previous dev

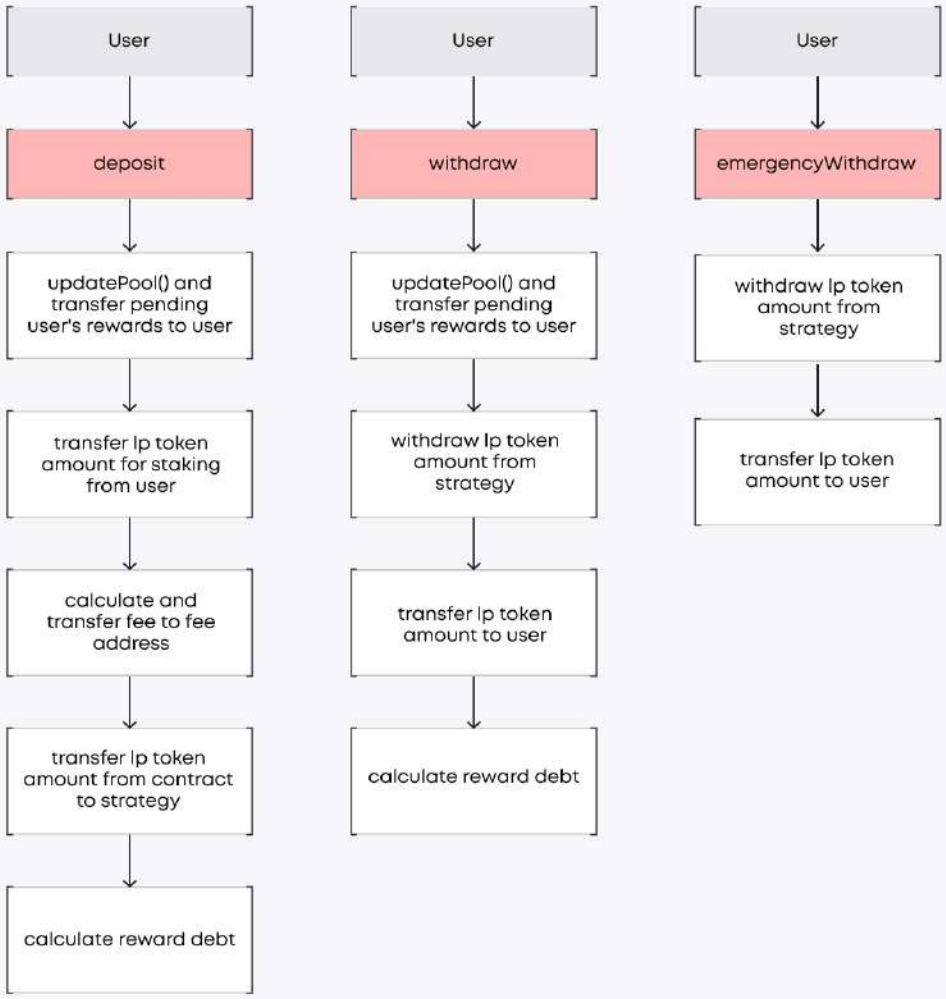
Receives a portion of the ARA rewards during the reward distribution process. A certain percentage (1/12th in this contract) of the rewards is transferred to the `dev`address

ARAFI SCHEME

ARAFI CONTRACTS

AraMasterChef.sol

User



ARAFI SCHEME

ARAFI CONTRACTS

AraMasterChefMultiReward.sol

Description

The second AraMasterChef contract is similar to the first one but with a few differences. The main functionality remains the same - managing liquidity pools and distributing rewards. However, this version supports multiple reward tokens for each pool, which allows users to earn multiple rewards simultaneously by staking LP tokens. The key differences and main functionalities are:

Differences:

1. Support for multiple reward tokens per pool.
2. The reward tokens and their respective reward per block are stored in separate mappings.
3. The UserInfo struct now has a mapping for rewardDebt instead of a single value.
4. The PoolInfo struct now has an array of reward tokens and a mapping for accRewardPerShare.

Main functionalities:

1. Staking LP tokens to earn multiple rewards simultaneously.
2. Adding and updating pools with different allocation points, deposit fees, and reward tokens.
3. Integration with external yield-optimizing strategies.
4. Updating reward tokens and their respective reward per block.
5. Emergency withdrawal of LP tokens without rewards.
6. Safe reward token transfer to prevent rounding errors.

Owner

add()	Add new pool
set()	Setup or update pool
updateReward()	Update reward token per block for specified token address
massHarvestFromStrategies(pids)	Call harvest on all strategies for provided pool ids
	Can be called by anyone
constructor()	Setup fee and dev addresses
addReward	Add a new reward token to an existing pool

Getters

poolLength()	
getRewardMultiplier(from, to)	Return reward multiplier over the given _from to _to block.
pendingReward()	View function to see pending user's rewards for provided pool ID and reward token on frontend

Fee address

setFeeAddress()	Commissions are sent to the fee address

Dev address

dev()	Update dev address by the previous dev

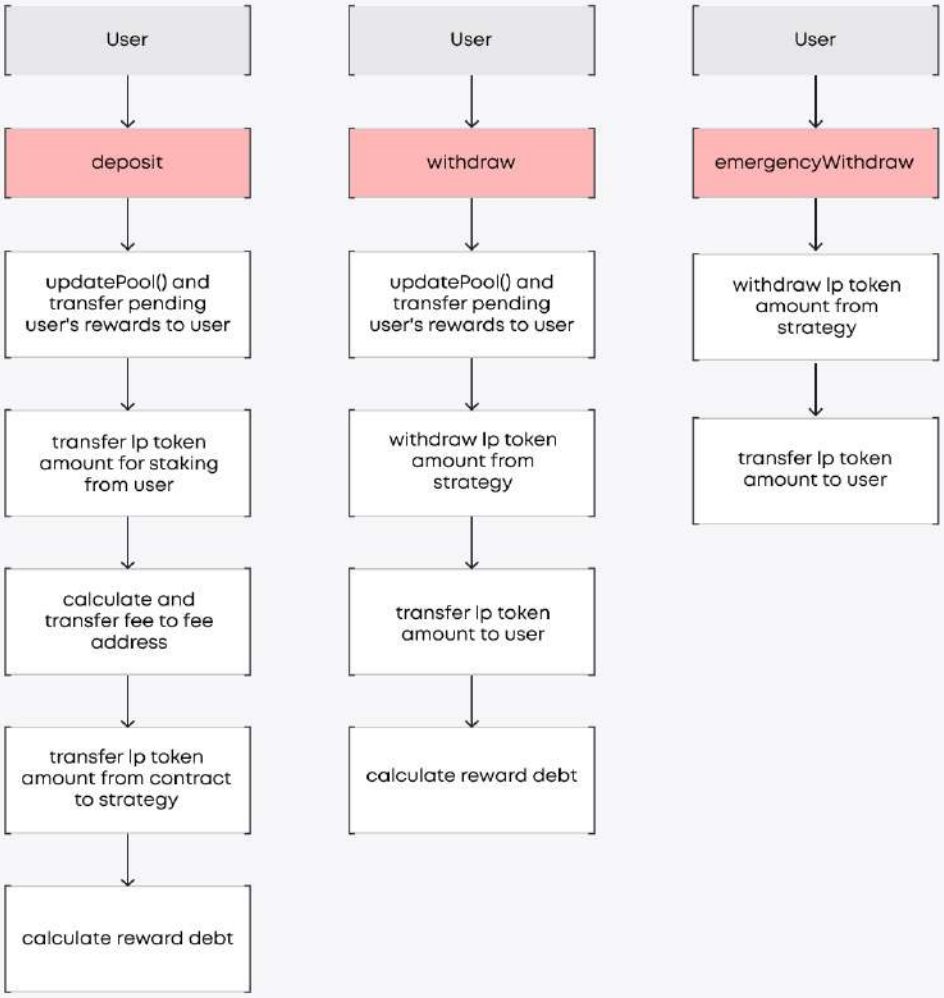
Receives a portion of the ARA rewards during the reward distribution process. A certain percentage (1/12th in this contract) of the rewards is transferred to the `dev` address

ARAFI SCHEME

ARAFI CONTRACTS

AraMasterChef.sol

User



ARAFI SCHEME

ARAFI CONTRACTS

AraLocker.sol

Description

This contract, named AraLocker, is a Solidity smart contract for a staking and rewards system. The main functionalities of this contract are as follows:

1. Staking: Users can lock their tokens (stakingToken) in the contract for a specified duration (lockDuration). The locked tokens are eligible to receive rewards from the system.

2. Rewards: The contract supports multiple reward tokens that can be distributed to stakers. The rewards are streamed over a defined period (rewardsDuration), and users can claim their pending rewards at any time.

3. Delegation: Users can delegate their voting power, which is based on their locked token balance, to another address. The delegation system uses checkpointing to keep track of the voting power for each delegatee over time.

4. Withdrawal: Users can withdraw their locked tokens after the lock duration has passed. They can also choose to re-lock their tokens to continue earning rewards. Additionally, there is a "kick" feature that allows other users to process expired locks for a user and earn a reward for doing so.

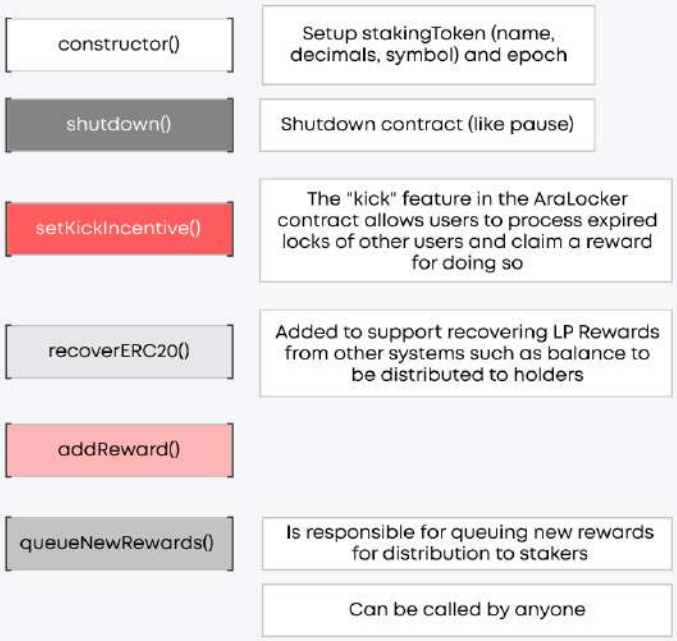
5. Blacklisting: The contract owner can blacklist certain addresses, preventing them from interacting with the contract.

6. Shutdown: The contract owner can shut down the contract, allowing users to withdraw their locked tokens without having to wait for the lock duration to pass.

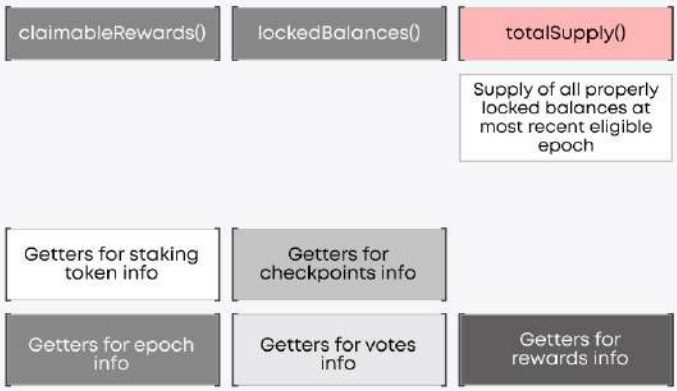
7. Reward Funding: External parties can fund the contract with new rewards, which will be distributed to stakers based on their locked token balance. The contract also supports queuing rewards and automatically distributing them when certain conditions are met.

Overall, the AraLocker contract is designed to incentivize users to lock their tokens in the contract and participate in the governance of the system by distributing rewards and allowing delegation of voting power.

Owner



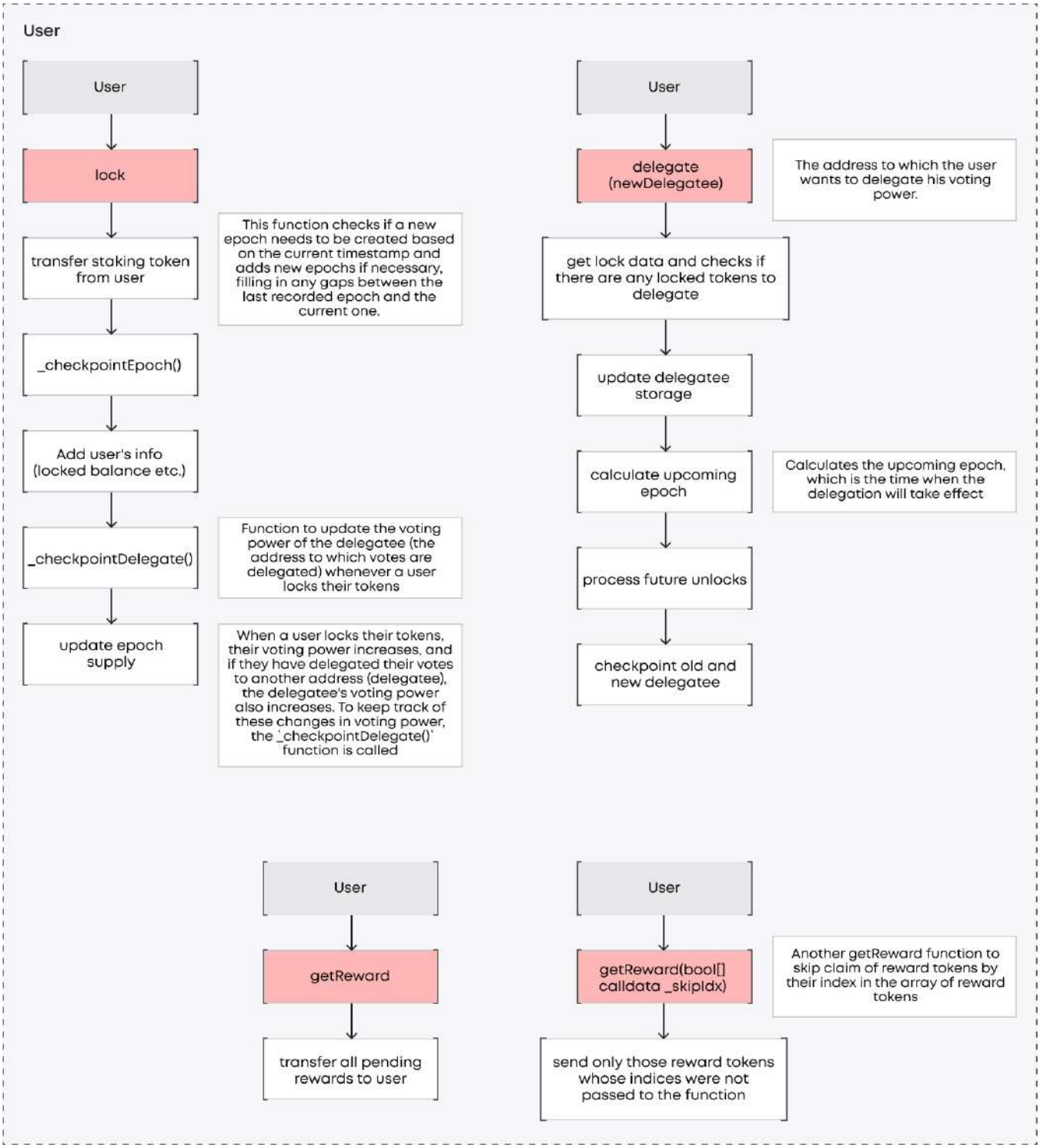
Getters



ARAFI SCHEME

ARAFI CONTRACTS

AraLocker.sol

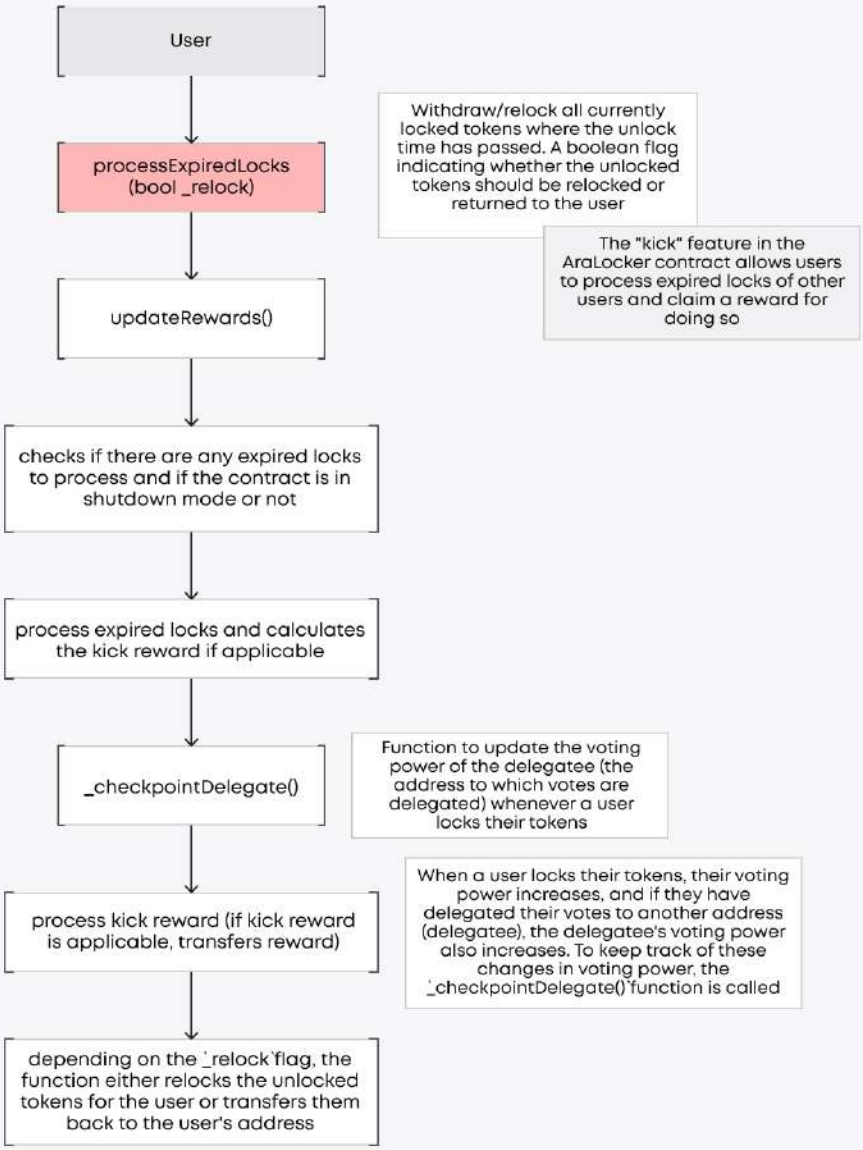


ARAFI SCHEME

ARAFI CONTRACTS

AraLocker.sol

User



COMPLETE ANALYSIS

CRITICAL-1	✔ Resolved

Incorrect deletion of a token ID in the user's array.

AraEmissionDistributor.sol: emergencyWithdraw().
In the emergencyWithdraw() function, the deletion of a token ID in the user's array (tokenIdsByCaller) is not handled correctly. The function uses a memory array for the tokenIdsByCaller array, which means that the changes made to the array are not reflected in the storage. As a result, the token ID is not properly deleted from the user's tokenIdsByUser mapping.

Recommendation:

Use a `storage` pointer instead of copying the array in the memory.

Post-audit:

Now, the storage pointer is used instead of copying the array into memory.

CRITICAL-2	✔ Resolved

Incorrect loop iteration in swap function leads to out-of-bounds error.

FractalMigrator.sol: swap(), lines 87-90.
The loop in the swap() function uses the index i`to obtain the new token while iterating from 0 to the length of the `_tokenIds`array. However, as the owned tokens array reduces in size during each iteration and the index i`simultaneously increases, the contract goes out of bounds and reverts. This could cause unexpected behavior and prevent the swap operation from succeeding, affecting the overall functionality of the contract.
The issue occurs due to the design of ERC721Enumerable.sol, where owned tokens are stored in mapping similarly to the array. Thus, the revert will happen in ERC721Enumerable.sol: tokenByIndex(), line 53.

Recommendation:

Modify the loop to iterate in reverse, starting from the last index of the `_tokenIds`array and decrementing the index i`during each iteration. This approach will ensure that the contract does not go out of bounds during the token swap process and will maintain the correct functionality of the swap operation. To implement this change, you can use a loop that iterates from `_tokenIds.length`to 0, decrementing the index i`at each step.

Post-audit:

The loop was modified so that the error doesn't occur.

CRITICAL-3

✔ Resolved

Contracts are not compiling.
After fixing contracts, most of them are not compiled. As a result, it cannot deploy or check contracts in tests. It is necessary to check if contracts can be compiled or not.

Recommendation:
Check and fix errors that prevent contracts compiling.

CRITICAL-4

✔ Resolved

Incorrect function call.
AraMasterChefMultiReward.sol, AraMasterChef.sol: withdraw(), _withdrawAllFromStrategy(), emergencyWithdraw().
In the contracts AraMasterChefMultiReward and AraMasterChef, the function withdraw(uint256 amount) of strategy contract (HummusV3Strategy or HummusV2Strategy) is called in withdraw(), _withdrawAllFromStrategy(), and emergencyWithdraw() functions. However, in the strategy contract, there is no function named withdraw with a uint256 parameter. It seems the contract intended to use the function withdrawTokens(uint256 amount) instead.
Due to this issue, the withdraw function in AraMasterChef.sol and AraMasterChefMultiReward might not work as intended, causing potential problems in the contract's functionality and user experience.

Recommendation:
Rename the function to match the function call and change the strategy interface considering this.

Post-audit:
Functions were renamed.

CRITICAL-5	✔ Resolved

Out-of-Bounds array access.
AraMasterChefMultiReward.sol: add().
There is an issue with out-of-bounds array access when creating a new pool. The variable `poolId` is assigned the length of the `poolInfo` array (line 1179), and then it is incremented before being used as an index to access the `poolInfo` array (line 1180). This results in accessing an element beyond the array's bounds, leading to unexpected behavior and potential security vulnerabilities.

Recommendation:
Instead of directly assigning the array's length to `poolId`, it should first push an empty PoolInfo struct to the `poolInfo` array and then access the newly added element using the correct index.

Post-audit:
Pushing an empty PoolInfo struct to the `poolInfo` array and accessing the newly added element using the correct index wasn't added.

CRITICAL-6	✔ Resolved

Incorrect token transfer amount.
AraEmissionDistributor.sol: safeAnotherTokenTransfer();
AraMasterChef.sol:safeAraTransfer();
AraMasterChefMultiReward.sol: safeRewardTransfer(),
aHUM.sol: safeARATransfer().
In the functions above, the contracts transfer the entire token balance to the recipient instead of the specified _amount. This could lead to unintended consequences, such as draining reward as the whole pool and causing potential loss of funds for other users.

Recommendation:
Replace the transfer of the token balance with the intended _amount.

Post-audit:
The transfer of the token balance with the intended _amount was replaced.

CRITICAL-7	✔ Resolved

Inconsistent NFT ownership checks in contracts.
AraEmissionDistributor.sol: withdrawAndDistribute(),
harvestAndDistributeAnotherToken(), emergencyWithdraw().
The contract checks if the NFT's owner is the same as before depositing the NFT for withdrawal and harvesting. However, when a user transfers the NFT, the ownership is also transferred. This could lead to issues where a user who no longer owns an NFT can still withdraw or harvest based on the previous deposit. But, it still needs some kind of NFT owner track because otherwise, anyone can withdraw someone else's NFT.

Recommendation:
Implement a mechanism to update the NFT ownership status in the contract whenever an NFT is transferred.

HIGH-1	✔ Resolved

Total allocation points are not updated.

AraEmissionDistributor.sol: setAnotherToken(), variable `totalAnotherAllocPoint`.
This value is not updated when the pool is set. Since the function for setting the pool can be called multiple times after the pool is added, the importance of totalAnotherAllocPoint`may be inaccurate, leading to the inaccurate distribution of rewards or any other unexpected behavior.

Recommendation:
Update totalAnotherAllocPoint`when the pool is set.

Post-audit:
The totalAnotherAllocPoint` is updated now.

HIGH-2	✔ Resolved

Uninitialized rewardPerBlock mapping.

AraMasterChefMultiReward.sol: mapping rewardPerBlock.
The rewardPerBlock mapping is never initialized, which may cause unexpected behavior when trying to access its values. For example, Reward per Share does not add rewards when updating the pool.

Recommendation:
Ensure that the rewardPerBlock mapping is properly initialized before being used in the contract. This can be done by initializing the mapping in the constructor or the addReward() function.

Post-audit:
Mapping is initialized with values > 0 in addReward() function.

HIGH-3	✔ Resolved

Wrong denominator is used.

AraEmissionDistributor.sol: pendingAnotherToken(), line 1292; updatePoolAnotherToken(), line 1331.
When the value accAnotherTokenPerShare is calculated, the denominator that should be used for the correct calculation must be equal to the sum of all users' amounts. However, the used value currently represents the total rewards stored on the contract, not the total amount of users' deposits. To keep the correct total supply of deposits per pool, it is recommended to add a variable totalSupply in the structure PoolInfoAnotherToken and update it during each execution of function depositToChef(), withdrawAndDistribute(), and emergencyWithdraw() so that it corresponds to the sum of all users' deposits in a certain pool.

For example, check the implementation of Sushi Swap MasterChef SC. When the rewards per share value is calculated, it uses the total supply of deposit tokens, not the total supply of reward tokens. https://github.com/1coinswap/core/blob/master/contracts/MasterChefV2.sol#L194
This issue is marked as high because the reward distribution might be inaccurate.

Recommendation:
Use the total supply of deposits in a pool as the denominator instead of the reward tokens' total supply.

Post-audit:
Total supply of deposits is used.

HIGH-4	✔ Resolved

Lack of checks for duplicate LP token pools in contracts.

AraMasterChef.sol, AraMasterChefMultiReward.sol.
When a pool is added in both the AraMasterChef and AraMasterChefMultiReward contracts, there are no checks to prevent the addition of a pool with an existing LP token. This could lead to issues during transfers if two or more pools have the same LP token.

Recommendation:
Implement a mechanism to check for duplicate LP tokens when adding a pool. This will prevent potential issues during transfers and ensure each pool has a unique LP token.

Post-audit:
The suggested mechanism was implemented.

MEDIUM-1	✔ Resolved

Transfers are not validated.

AraMasterChef.sol: updatePool(), line 1211-1212; safeAraTransfer(), line 1297, 1299.
AraEmissionDistributor.sol: depositAnotherToken(), line 916; safeAnotherTokenTransfer(), line 1119, 1122.
AraFractal.sol: withdrawUSDC(), line 809.
aHUM.sol: depositToChef(), line 2072; safeARATransfer(), lines 2194, 2196; withdrawErc20Tokens(), line 2216.
AraMigrator.sol.
Transferring is performed with a regular transfer() or transferFrom() method from the OpenZeppeling IERC20 interface without validation if the transfer is successful in almost all contracts. To ensure the security of the transfer, it is recommended to validate the success of the transfer call using SafeERC20, which contains all necessary security checks.

Recommendation:

It is recommended to use the SafeERC20 library and replace transfer()/transferFrom() with safeTransfer()/safeTransferFrom() functions.

Post-audit:

The transfers are validated now.

MEDIUM-2	✔ Resolved

Transfer is validated with assert.

AraFractal.sol: _deposit_for();
AraEmissionDistrubutor.sol: _deposit_for(), line 500;
TokenWrapper.sol: deposit(), line 63.
Transferring is performed with a regular transfer() method from the OpenZeppeling IERC20 interface with the validation by asserts for transfers to be successful. Still, the SafeERC20 library provides additional validations, which are important. To ensure the security of the transfer, it is recommended to use the OpenZeppelin SafeERC20 library.s

Recommendation:

It is recommended to use SafeERC20 library and replace transfer() with safeTransfer() functions.

Post-audit:

SafeERC20 is used now.

MEDIUM-3	✔ Resolved

An array may contain empty elements.

AraEmissionDistributor: withdrawAndDistribute(), line 944; emergencyWithdraw(), line 995.
When an NFT is withdrawn, it should be removed from an array of the user's staked NFTs, `tokenIdsByUser`. Though it is deleted with the operator `delete`, this operator doesn't actually delete the element from the array but sets its value to 0. To remove the element from the array, it is recommended to:

1. Swap the values of the element to remove the last element.
2. Pop the last element from an array.

This way, the array won't contain empty elements. The issue is marked as medium since empty elements increase gas consumption during iteration through the elements of the array and may even lead to an “out of gas” error, obstructing the users' withdrawal funds.

Recommendation:

Remove an element from the array so it doesn't contain empty elements.

Post-audit:

Elements are removed from the array now.

MEDIUM-4	✔ Resolved

Any user can deposit arbitrary `new tokens.`

FractalMigrator.sol: depositNewTokens().
The FractalMigrator`contract contains the depositNewTokens() function. This function includes access control ("Only the contract owner can deposit new tokens.") which prevents users without ownership from depositing 'new tokens'.
However, any user can deposit tokens via transferFrom directly to the contract.

Recommendation:

Verify that any user should be able to transfer tokens to the contract and remove the function because it is unnecessary **OR** protect the contract from receiving tokens from other users by implementing an onERC721Received() hook to validate the sender. However, note that users will still be able to transfer tokens directly with the transferFrom() function.

Post-audit:

depositNewTokens() now doesn't contain access control, so any user can transfer tokens to a contract.

MEDIUM-5	✔ Resolved

ERC721 is treated like a fungible token.

FractalMigrator.sol: swap().
In swap(), NFTs are exchanged by quantity without reference to their tokenID, parameters, etc. Any N 'oldTokens' will be exchanged for almost random N 'newTokens'. There are no rules for matching old and new tokens or exchange rules other than quantity.

Recommendation:

Verify that the contract's logic is correct.

Post-audit:

The Aerarium team has added rules for the exchange. Mapping tokenSwapMapping has been added in which it is set which NFT will be exchanged by token ID.

MEDIUM-6	✔ Resolved

Harvester can add new harvesters.

HummusV2Strategy.sol/HummusV3Strategy.sol:
whitelistHarvesters(), line 756.
If the sender is added as a harvester, he can add new harvesters that could be malicious users. Usually, it is an admin role that can grant other roles to users. This distribution in perks is more secure in case the account of any of the harvesters is compromised.

Recommendation:

Remove the ability for harvesters to add new harvesters and consider leaving it only to the governor.

Post-audit:

The ability to add new harvesters was left only to the governor.

MEDIUM-7	✔ Resolved

Users could get less Reward tokens.

aHUM.sol: safeARATransfer().AraMasterChef.sol: safeARAtransfer().
AraMasterChefMultiReward.sol: safeRewardTransfer().
AraEmissionDistributor.sol: safeAnotherTokenTransfer().
It is possible that the contract could run out of Reward tokens. In this case, users could have fewer tokens than they should when they try to withdraw. For example, the user has 100 Reward tokens to get, but the contract has only 10. Then, the user will have 10 instead of 100. The same goes for the following user, who will get 0 tokens. Consider checking of the amount of Reward still valid for the user which will be added in the next transaction.

Recommendation:

Add a variable to check for users' Reward amount to withdraw.

Post-audit:

Variable check was added.

LOW-1	✔ Resolved

Unused parameters and variables.

1. AraMasterChefMultiReward.sol: constructor();The constructor has an unused parameter _araPerBlock and _devaddr, which may lead to confusion and potential issues in the future.
2. AraLocker.sol: getReward().The getReward() function has an unused parameter _stake, which may lead to confusion and potential issues in the future.
3. AraMasterChefMultiReward.sol: `devaddr,` line 1063.
4. AraEmissionDistributor.sol: POOL_PERCENTAGE, line 842.Variable is initialized, however it is never used across the contract.
5. FractalMigrator.sol: _tokenIdGeneral.Variable is initialized, but never is used across the contract.

Recommendation:

Remove the unused parameters to avoid confusion and potential issues **OR** add logic in which these parameters will be used **OR** verify that they are needed for future functionality. Validate the necessity of unused storage variables (e.g., if it is necessary for frontend or other contracts) **OR** remove it.

LOW-2	✔ Resolved

Same function name.

HummusV2Strategy.sol/HummusV3Strategy.sol: withdraw(), lines 785, 793.
Contracts contain functions with the same name, which could confuse which function to use or the purpose of a particular function. It is better to create a name for a function to describe what it does. E.g., withdrawAssets/withdrawTokens.

Recommendation:

Change function names to avoid duplicated names.

Post-audit:
Function names were changed.

LOW-3	✔ Resolved

Redundant access checks.

HummusV2Strategy.sol/HummusV3Strategy.sol: withdraw(address)
Function has onlyBenevolent modifier, but also contains a check that msg.sender is a governance address.

Recommendation:

Remove the modifier or the requirement.

Post-audit:
The requirement was removed.

LOW-4	✔ Resolved

Only pool with ID 0 can be set.

aHUM.sol: set().
The function updates allocation points only for pools with ID 0. Thus, any other pools become immutable and can't be updated. While this may not be an issue in case other pools should be immutable by design, it is present in the report to verify the correctness of the function's logic.

Recommendation:

Verify that only pool with ID 0 should be updated, **OR** add parameter with pool ID to update a certain pool.

Post-audit:
Pool ID is passed as a function parameter now.

LOWEST-1	✔ Resolved

Inaccurate version pragma.

`pragma solidity ^>=0.6.0 <0.8.0; 0.6.12 >=0.6.0; 0.6.12; ^0.6.7; 0.8.0, 0.8.11` is used in contracts. The contract should be deployed with the same compiler version and options with which it has been most tested. Locking the pragma version helps ensure the contract is not accidentally deployed using a different version. In addition, older versions may contain bugs and vulnerabilities, and be less optimized in terms of gas. Using the latest version of Solidity and specifying the exact pragma is recommended.

Recommendation:

Specify the latest version of Solidity in the pragma statement.

Post-audit:
Solidity 0.8.18 is used now.

LOWEST-2	Unresolved

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:
Custom errors were to several smart contracts; however, some still utilize the required statements.

LOWEST-3	Unresolved

Lack of validation.

AraMasterChefMultiReward.sol: constructor(), add();
AraMasterChef.sol: constructor(), add();
AraLocker.sol: constructor();
AraEmissionDistributor.sol: withdrawAndDistribute(), harvestAndDistributeAnotherToken(), emergencyWithdraw();
TokenWrapper.sol, VeWrapper.sol, StakeWrapper.sol: setRate(), setFee(), initialize().
Either not all or none of the parameters passed to these functions are validated. This can lead to unexpected behavior in further use, which is why it is recommended to validate parameters. For instance, addresses should be checked to ensure they are not equal to zero addresses, and numeric parameters should be checked to ensure they are not equal to 0 or verify if that user owns veARA NFT.

Recommendation:

Add necessary validations.
Post-audit: Validations were added for all the contracts except wrappers.

LOWEST-4	✔ Resolved

Use of SafeMath library.

The contracts use the SafeMath library for arithmetic operations. While this is a good practice to prevent integer overflows and underflows, it is worth noting that Solidity 0.8.0 and later have built-in overflow and underflow checks, making using the SafeMath library unnecessary.

Recommendation:

Consider upgrading the contract to the latest version of Solidity and removing the SafeMath library to reduce gas costs and simplify the code.

LOWEST-5	Unresolved

Lack of events.

AraMasterChef.sol: set(), add(), setMultiplier();
AraMasterChefMultiReward.sol: setFeeAddress(), setMultiplier();
AraLocker.sol: addReward(), _checkpointEpoch();
AraEmissionDistributor.sol: depositAnotherToken(), updatePoolAnotherToken();
AraFractal.sol: setVoter(), voting(), attach(), detach();
TokenWrapper.sol, VeWrapper.sol, StakeWrapper.sol, LibOptions.sol: multiple functions.
aHUM.sol: updateEmissionRate(), claimVeHumRewards(), withdrawHumRewards(), withdrawErc20Tokens(), withdraw().
To keep track of historical changes of storage variables, it is recommended to emit events on every change in the functions that modify the storage.

Recommendation:

Emit events in these functions.

Post-audit:
Events were added for all contracts except wrappers.

LOWEST-6	✔ Resolved

Unused public visibility.

1) AraMasterChefMultiReward.sol: set(), add(), emergencyWithdraw(), setFeeAddress(), withdraw(), deposit(), updateReward(), addReward();
2) AraMasterChef.sol: add(), set(), deposit(), withdraw(), emergencyWithdraw(), dev(), setFeeAddress(), updateEmissionRate();
3) AraEmissionDistributor.sol: harvestAndDistributeAnotherToken(), emergencyWithdraw(), addAnotherToken(), setAnotherToken().
Functions are currently marked as public but are only called from outside the contract and do not need to be called internally. Marking these functions as external can save gas costs as external functions have lower gas consumption than public functions when called externally.

Recommendation:

Change the visibility of functions from public to external to optimize gas usage.

LOWEST-7	✔ Resolved

Inconsistent variables' naming.

AraFractal.sol.
Variables "user_point_history" and "user_point_epoch" do not correspond to their values. They are supposed to keep NFT's history and epoch as locks are no longer concentrated over user addresses but the token IDs.

Recommendation:

Change variables' naming according to their purpose.

Post-audit:
Variables' naming was changed.

LOWEST-8	✔ Resolved

5 DepositTypes declared in enum but 3 of them were not implemented.
AraFractal.sol: enum DepositType, line 16.
DEPOSIT_FOR_TYPE, INCREASE_LOCK_AMOUNT and INCREASE_UNLOCK_TIME deposit types were not implemented.

Recommendation:

Remove unused DepositTypes from enum **OR** implement corresponding functionality **OR** verify that it is created for future functionality.

Post-audit:
Unused DepositTypes weren't removed, corresponding functionality wasn't implemented and wasn't verified.

Post-audit:
Unused DepositTypes were removed.

LOWEST-9	✔ Resolved

Mappings can be used.

AraEmissionDistributor.sol: poolInfoAnotherToken, tokenInfo.
These arrays are only used to add new elements to the end of the sequence and read them. There is also no sequence processing specific to arrays and no getting functions for all sequence elements. In such cases, it is better to use mapping to reduce gas consumption.

Recommendation:

Replace these arrays with mappings.

Post-audit:
Arrays weren't replaced with mappings.

LOWEST-10	✔ Resolved

Pool ID is not explicitly verified.
AraEmissionDistributor.sol: depositToChef(), withdrawAndDistribute(), harvestAndDistributeAnotherToken(), emergencyWithdraw(), setAnotherToken().
AraMasterChef.sol, AraMasterChefMultiReward.sol: set(), updatePool(), withdraw(), deposit(), emergencyWithdraw().
aHUM.sol: depositToChef(), stake(), withdrawAndDistribute(), harvestAndDistribute(),
There is no explicit validation that the pool provided `_pid`exists. Though all the transactions with invalid pool ID will still revert when reading a non-existing element from arrays `poolInfo`or `poolInfoAnotherToken,` adding an explicit validation with a clarified error message is still recommended for better error clarification.

Recommendation:

Validate that pool with provided `_pid`exists like it is done in depositAnotherToken(), line 914.

LOWEST-11	✔ Resolved

Rewards for another pool may be calculated inaccurately.
AraEmissionDistributor.sol: updatePoolAnotherToken(), line 1095.
Value `anotherTokenRewardsForPool`is calculated using the formula: (anotherTokenRewards * DENOMINATOR) / DENOMINATOR. However, multiplying and dividing by the same value, DENOMINATOR is redundant and may only lead to a loss of accuracy.
There is also an assumption that the value should be multiplied by POOL_PERCENTAGE instead of DENOMINATOR, especially since POOL_PERCENTAGE is not used now.

Recommendation:

Verify the correctness of the formula **OR** fix it.

Post-audit:
Formula was fixed.

LOWEST-12	✔ Resolved

Comments mismatch with code.
AraEmissionDistributor.sol: `tokenInfo`, line 833.
Comments don't correspond to the variables and code they describe.
Recommendation:

Fix comments.

Post-audit:
Comments were fixed.

LOWEST-13	✔ Resolved

Redundant initialization.
AraEmissionDistributor.sol: `totalPidsAnotherToken`, `totalAnotherAllocPoint`; lines 830, 835.
aHUM.sol: `totalAllocPoint`, `totalAmountOfSupplyStaked`; lines 2023, 2025.
Variables are set to 0 during deployment explicitly, which increases the gas cost for the deployment. Since all values are initialized with zero values by default, initializing it explicitly to zero is redundant.

Recommendation:

Remove explicit initialization to 0.

Post-audit:
Redundant initialization was removed.

LOWEST-14	Unresolved

Asserts are used for validating inputs.
AraFractal.sol: multiple functions.
StakeWrapper.sol: initialize(), line 46;
VeWrapper.sol: initialize(), line 105.
According to Solidity docs, the “assert” function should only be used to test for internal errors and to check invariants. The “require” function should ensure valid conditions, such as inputs or contract state variables, are met or validate return values from calls to external contracts. For the latest Solidity versions, it is recommended to use custom errors instead to save gas.

Recommendation:

Replace the asserts with require **OR** use custom errors.

Post-audit:
The asserts were replaced in all contracts except wrappers.

LOWEST-15	✔ Resolved

Duplicating require-statements.
AraFractal.sol: attach(), detach(), voting(), setVoter().
Four identical require-statements are used for validating the msg.sender to be the Voter address. A single modifier should be used as a best practices pattern.

Recommendation:

Use modifiers instead of the require-statements.

Post-audit:
Duplicated require-statements were replaced with modifiers.

LOWEST-16	✔ Resolved

Require message mismatch with code.
FractalMigrator.sol: withdrawNewTokens(), line 101;
AraMigrator.sol: withdrawOldTokens(), line 122.
Require messages don't correspond to variables and code they describe.
In addition, these required messages can be replaced with a single modifier, or OpenZeppelin's Ownable can be used because it is the industry standard and contains proven and advanced functionality for access control.

Recommendation:

Fix require messages and replace require messages with modifier **OR** use OpenZeppelin's Ownable.

Post-audit:
Require messages were fixed, and ownership validation differs from standard OpenZeppelin.

LOWEST-17	✔ Resolved

Solidity style guide violation.

1) AraFractal.sol, AraMasterChefMultiReward.sol, AraMigrator.sol, FractalMigrator.sol, HummusV2Strategy.sol, HummusV3Strategy.sol. Files names and contractsnames mismatch.

Recommendation:

Name contract file and contract itself with the same name.

2) AraMasterChef.sol, AraMasterChefMultiReward.sol.
The smart contract contains several magic numbers and hardcoded values that may not be easily understood or changed. These magic numbers can make the code harder to read, maintain, and modify.

Recommendation.

Replace magic numbers with named constants or variables to improve code readability and maintainability.

Post-audit:

Files names and contracts names still mismatch. Magic numbers weren't replaced with named variables.

LOWEST-18	Unresolved

Anyone can withdraw native tokens.

HummusV2Strategy.sol/HummusV3Strategy.sol: harvestNativeToken().
The function does not have a check on who is invoking the function, so anyone can withdraw tokens from the contract. The issue is marked as info since all this function does is receive ETH from msg.sender and send it back to msg.sender, which makes the function redundant or unfinished.

Recommendation:

Clarify the necessity of the function or finish logic behind it.

Post-audit:

Function was restricted, though its destination remains unclear.

LOWEST-19	Unresolved

Insufficient documentation and comments in the code.

TokenWrapper.sol, VeWrapper.sol, StakeWrapper.sol, LibOptions.sol; aHUM.sol, AraMigrator.sol, FractalMigrator.sol.

The contracts and library have limited inline documentation and comments, making it difficult for developers to understand the code's functionality and intent. Proper documentation ensures maintainability, readability, and ease of collaboration with other developers.

Recommendation:

Add detailed inline comments and documentation to the contracts and library, explaining the purpose and functionality of each function, as well as any assumptions or constraints. This will help future developers understand the code more efficiently and reduce the risk of introducing errors during updates or modifications.

Post-audit:

Comments and documentation weren't added.

LOWEST-20	✔ Resolved

Redundant calculation of multiplier.

AraMasterChef.sol, AraMasterChefMultiReward.sol.
The Solidity smart contracts code contains a redundant multiplier calculation in the `getMultiplier` function. The BONUS_MULTIPLIER` constant is set to 1, meaning the multiplier analysis does not affect the result. This issue also affects other parts of the code that use the getMultiplier`function, such as the `updatePool` function.

Recommendation:

Remove the redundant calculation of the multiplier and update the affected parts of the code accordingly **OR** make this variable non-constant, add a setter for it, and then this functionality can be used in the future.

Post-audit:
Variable was made non-constant and a setter for it was added.

LOWEST-21	✔ Verified

Rewards distribution is finished on V2.

HummusV2Strategy.sol.
Strategy is connected to contract on mainnet where rewards are not currently distributed. Also, according to the transaction, some sort of migration is in progress. https://andromeda-explorer.metis.io/address/0x9cadd693cDb2B118F00252Bb3be4C6Df6A74d42C

Recommendation:

Verify that V2 Strategy is relevant. - FOR NOW IS NOT

From client:
V2 Strategy will not be used.

LOWEST-22	✔ Resolved

Wrong pool ID is passed in the event and absence of pool ID in the event.

aHUM.sol: depositToChef(), stake(), withdrawAndDistribute(), add(), set().

In all the functions, 0 is passed in events instead of the actual pool ID in which deposit/withdraw or pool addition was performed. As a result, events may provide wrong historical data. Also, the event in function set() has no parameter for pool ID, which may also be confusing.

Recommendation:

Pass the correct ID in the event. Add parameter for pool ID in event LogSetPool().

Post-audit:

The correct ID passed in the event. Parameter for pool ID in event LogSetPool() was added.

LOWEST-23	Unresolved

Specific order of functions.

AraLocker.sol: addReward(), queueNewRewards(), claimableRewards().

In AraLocker.sol, the distribution of rewards will only work correctly if the addReward() function is called, and immediately after that, queueNewRewards for the same reward token. Otherwise:

- If you call queueNewRewards() and then addReward(), you receive an error Reward already exists`, line 1065 from addReward(), and then you can't add reward in rewardTokens array, so subsequent use of this array in other functions will revert.
- If you call addReward() and then not immediately (after some time) call queueNewRewards(), rewards will not display correctly with claimableRewards().

Recommendation: Verify that the described logic is correct for the contract.

LOWEST-24	✔ Resolved

Strategy for aHUM should be verified.

When a user executes the function depositToChef() on aHUM.sol, aHUM tokens are deposited to either AraMasterChef or AraMasterChefMultiReward, where aHUM tokens are deposited into the strategy, which deposits tokens to the Hummus. However, since Hummus doesn't have the necessary pool for aHUM, using this strategy is impossible. The Master Chef contracts suggest that the strategy may remain unset by passing zero addresses to it. Thus, the tokens won't be deposited into the strategy. However, due to the implementation of functions add() and set() on both Master Chef smart contracts, it is impossible to set the strategy as zero address.

Recommendation:
Verify which strategy will be used for aHUM and whether Hummus will add a pool for aHUM.

Post-audit:
The AraFi team has verified that HummusStrategyV3 will be used, though aHUM pool is unlikely to be added to the Hummus protocol. Also, it is possible now to set zero address for strategy on AraMasterChefMultiReward. However, it is still not possible to do this on AraMasterChef.

LOWEST-25	✔ Resolved

Tokens are not withdrawn from strategy during the emergency withdrawal.
AraMasterChefMultiReward.sol: emergencyWithdraw().

If the strategy is set for a certain pool in AraMasterChefMultiReward, the execution of the emergencyWithdraw() function will revert since the necessary amount of tokens is not withdrawn from the strategy. As a result, trying to transfer the amount of the required tokens will fail due to the lack of balance of Master Chef. This issue is marked as info since the necessary mechanism for withdrawing tokens is already implemented in the emergencyWithdraw() of AraMasterChef.sol. Thus, the implementation should be copied to the AraMasterChefMultiReward as well.

Recommendation:
Withdraw tokens from the strategy if the balance of AraMasterChefMultiReward is insufficient.

Post-audit:
The AraFi team has implemented the necessary withdrawal of tokens from the strategy.

		VeWrapper.sol **StakeWrapper.sol** **TokenWrapper.sol** **LibOptions.sol** **AraLocker.sol** **AraMasterChef.sol**
✔	Re-entrancy	Pass
✔	Access Management Hierarchy	Pass
✔	Arithmetic Over/Under Flows	Pass
✔	Delegatecall Unexpected Ether	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

		AraMasterChefMultiReward.sol **aHUM.sol** **AraEmissionDistributor.sol** **AraFractal.sol** **AraFiToken.sol** **AraMigrator.sol**
✔	Re-entrancy	Pass
✔	Access Management Hierarchy	Pass
✔	Arithmetic Over/Under Flows	Pass
✔	Delegatecall Unexpected Ether	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

FractalMigrator.sol
HummusV2Strategy.sol
HummusV3Strategy.sol

✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	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, PREPARED BY BLAIZE SECURITY TEAM

AraEmissionDistributor

Deployment
- ✓ Reverts when initialization if the Ve token is the zero address

Setting of the farming pool ID
- ✓ Deposits fund to pool (90ms)

Deposit of a veARA NFT to the Ara master chef
- ✓ Deposits (94ms)
- ✓ Reverts when deposit if the user does not own the veARA NFT (49ms)

Deposit of the another token
- ✓ Deposits
- ✓ Reverts when deposit if an incorrect pool ID
- ✓ Prevents non-owners from deposit

Withdrawal and distribution
- ✓ Withdraws and distributes (92ms)
- ✓ Withdraws and distributes if the pool for the another token is closed (98ms)
- ✓ Withdraws and distributes if there are two deposits from the user (170ms)
- ✓ Reverts when withdrawing and distributing if the user does not have enough tokens (61ms)

Harvest and distribution of the another token
- ✓ Harvests and distributes (49ms)
- ✓ Does not transfer rewards when harvesting and distributing if no rewards
- ✓ Only updates the pool when harvesting and distributing if it is closed (54ms)

Emergency withdrawal
- ✓ Emergency withdraws (77ms)
- ✓ Emergency withdraws if there are two deposits from the user (148ms)

Adding of a new pool of the another token
- ✓ Adds
- ✓ Prevents non-owners from adding

Setting of the parameters of the pool for the another token
- ✓ Sets
- ✓ Prevents non-owners from setting

Update of a pool
- ✓ Updates rewards

- ✓ Does not update rewards if the last reward block
- ✓ Updates rewards for the another token
- ✓ Does not update rewards for the another token if the last reward block

Getters

- ✓ Returns pending rewards of a user for the another token (70ms)

AraFractal

veAra: lock creation

- ✓ should create lock [first lock]
- ✓ should create lock [second lock] (43ms)

veAra: token ownership

- ✓ should return totalSupplyAt at given block [first epoch]
- ✓ should return totalSupplyAt at given block [second epoch]
- ✓ should return totalSupplyAt at given block [epoch < currentEpoch] (53ms)
- ✓ should revert when calling totalSupplyAt with given block > current block
- ✓ should return correct token URI (142ms)
- ✓ should revert when calling tokenURI for nonexistent token
- ✓ should return balanceOfNFT for a given token
- ✓ should return user point history ts for a given token
- ✓ should return tokenOfOwnerByIndex (62ms)
- ✓ should return approved address for token (56ms)
- ✓ should return true if owner calls isApprovedOrOwner (40ms)
- ✓ should return true if approved address calls isApprovedOrOwner (43ms)
- ✓ should return false if not owner nor approved calls isApprovedOrOwner
- ✓ should return block number
- ✓ should setApprovalForAll to true
- ✓ should setApprovalForAll to false
- ✓ should revert when approving non-existent token
- ✓ should revert when approving to self
- ✓ owner of NFT should be able to transfer token (47ms)
- ✓ owner of NFT should be able to transfer token (48ms)
- ✓ approved address of for all NFTs should be able to transfer token (53ms)
- ✓ approved address of for particular NFT should be able to transfer token (54ms)
- ✓ should revert when not owner not approved address tries to transfer NFT (40ms)
- ✓ should revert when owner tries to transfer NFT which was attached (41ms)

veAra: attachments

- ✓ should set new voter
- ✓ should revert when not voter tries to set new voter

- ✓ should revert when not voter tries to detach NFT
- ✓ should cast vote for particular NFT
- ✓ should cast vote for particular NFT

aHUM

Main

- ✓ Add pool
- ✓ Deposit to chef (85ms)
- ✓ Harvest (105ms)
- ✓ Claim veHUM rewards (88ms)
- ✓ Withdraw HUM from veHUM contract (95ms)

StakeWrapper

Initialize

- ✓ Should initialize the contract (432ms)
- ✓ Should not initialize the contract if already initialized (213ms)

Constructor

- ✓ Should correctly set the token and stakeToken (80ms)

Deposit

- ✓ Should deposit tokens and stake them (490ms)
- ✓ Should not deposit tokens if paused (131ms)
- ✓ Should not deposit tokens if blacklisted (173ms)
- ✓ Should not deduct fee if fee is 0 (439ms)

Setters

- ✓ Should set treasury (88ms)
- ✓ Should not set treasury if not admin (198ms)
- ✓ Should set fee (85ms)
- ✓ Should not set fee if not admin (222ms)
- ✓ Should set rate (81ms)
- ✓ Should not set rate if not admin (284ms)

VeWrapper

Initialize

- ✓ Should initialize the contract (204ms)
- ✓ Should not initialize the contract if already initialized (57ms)

Constructor

- ✓ Should correctly set the token and stakeToken (98ms)

Deposit

- ✓ Should deposit tokens and stake them (445ms)
- ✓ Should not deposit tokens if paused (131ms)

- ✓ Should not deposit tokens if blacklisted (161ms)
- ✓ Should not deduct fee if fee is 0 (329ms)

Receive NFT

- ✓ Should receive NFT and merge them (218ms)
- ✓ Should not receive NFT if paused (155ms)
- ✓ Should not receive NFT if blacklisted (270ms)
- ✓ Should not receive NFT if not from voting escrow (68ms)

Setters

- ✓ Should set treasury (73ms)
- ✓ Should not set treasury if not admin (212ms)
- ✓ Should set fee (42ms)
- ✓ Should not set fee if not admin (145ms)
- ✓ Should set rate (55ms)
- ✓ Should not set rate if not admin (157ms)
- ✓ Should set veRate (56ms)
- ✓ Should not set veRate if not admin (133ms)

AraLocker contract

Modify black list

- ✓ Should emit event 'BlacklistModified' (45ms)
- ✓ Should reverted with 'Ownable: caller is not the owner' (44ms)
- ✓ Should reverted with 'Must be contract'

Add reward token

- ✓ Should add new reward token to array (53ms)
- ✓ Should reverted with 'Ownable: caller is not the owner'
- ✓ Should reverted with 'Reward already exists'
- ✓ Should reverted with 'Max rewards length' (114ms)

Set kick incentive

- ✓ Should emit event 'KickIncentiveSet'
- ✓ Should reverted with 'Ownable: caller is not the owner'
- ✓ Should reverted with 'over max rate'
- ✓ Should reverted with 'min delay'

shutdown the contract

- ✓ Should emit event 'Shutdown'
- ✓ Should reverted with 'Ownable: caller is not the owner'

Added to support recovering LP Rewards
✓ Should emit event 'Recovered' (88ms)
✓ Should reverted with 'Ownable: caller is not the owner' (52ms)
✓ Should reverted with 'Cannot withdraw staking token' (70ms)
✓ Should reverted with 'Cannot withdraw reward token' (70ms)
Locked tokens
✓ Should emit event 'Staked' (93ms)
✓ Should emit event 'Staked' (147ms)
✓ Should revert with 'blacklisted' (81ms)
✓ Should revert with 'Cannot stake 0' (43ms)
✓ Should revert with 'shutdown' (71ms)
getReward
✓ Should emit event 'RewardPaid' (220ms)
✓ Should revert with '!arr'
✓ Should revert with string 'RewardPaid' (248ms)
Checkpoint epoch
✓ Check stored epochs date
Withdraw/relock all currently locked tokens
✓ Should emit 'Withdrawn' (255ms)
✓ Should revert with 'no exp locks' (74ms)
✓ Should revert with 'no locks'
kick expired locks
✓ Should emit 'Withdrawn' (102ms)
✓ Should revert with 'no exp locks' (75ms)
✓ Should revert with 'no locks'
Withdraw without checkpointing
✓ Should emit 'Withdrawn' (167ms)
✓ Should change receiver balance (163ms)
✓ Should revert with 'Nothing locked'
✓ Should revert with 'Must be shutdown'
Delegate votes from the sender
✓ Should emit 'DelegateChanged' (120ms)
✓ Should emit 'DelegateChanged' (99ms)
✓ Should emit 'DelegateCheckpointed' (72ms)
✓ Should emit 'DelegateCheckpointed' with checkpoint pass (85ms)
✓ Should emit 'DelegateChanged' in further also (78ms)
✓ Should revert with 'Must delegate to someone' (69ms)

- ✓ Should revert with 'Must choose new delegatee' (81ms)
- ✓ Should revert with 'Nothing to delegate'

Get delegating address

- Should emit 'DelegateChanged'

Gets the current votes

- ✓ Should return current vote

Get the `pos`-th checkpoint

- ✓ Should return checkpoint (93ms)

Get number of checkpoints

- ✓ Should return number of checkpoint

Get the number of votes for `account

- ✓ Should return number of votes

Get the totalSupply`at the end of `timestamp

- ✓ Should return totalSupply data (88ms)
- ✓ Should return with 'ERC20Votes: block not yet mined' (92ms)

Check balance of an account

- ✓ Should return correct balance

Return information on a user's locked balances

- ✓ Should return locked balance

Return supply of all properly locked balances

- ✓ Should return total supply

Return supply of all properly locked balances

- Should return total supply

Return an epoch index based on timestamp

- ✓ Should return epoch index

Epoch count

- ✓ Should return epoch length

Get decimals

- ✓ Should return decimals

Get name

- ✓ Should return name

Get symbol

- ✓ Should return symbol

Get claimable amount

- ✓ Should return correct amount (180ms)

Get last time reward applicable

- ✓ Should return correct last time reward

Get reward per token
✓ Should return correct last time reward
Get queue ner rewards
✓ Should emit 'RewardAdded' (143ms)
✓ Should emit 'RewardAdded' before 7 days (143ms)
✓ Should emit 'Withdrawn' (145ms)

AraMasterChef contract

Add pool
✓ Should add pool data in storage (40ms)
✓ Should add pool data in storage when 'withUpdate' parameter is equal 'false'
✓ Should revert with 'Ownable: caller is not the owner'
✓ Should revert with 'add: invalid deposit fee basis points'
Set pool
✓ Should change pool data in storage (67ms)
✓ Should change strategy and transfer balances (548ms)
✓ Should change pool data in storage when 'withUpdate = false' (44ms)
✓ Should revert with 'Ownable: caller is not the owner'
✓ Should revert with 'add: invalid deposit fee basis points'
deposit
✓ Should emit event 'Deposit' (309ms)
✓ Should emit event 'Deposit' when amount is 'zero' (68ms)
withdraw
✓ Should emit event 'withdraw' (283ms)
✓ Should revert with 'withdraw: not good' (200ms)
emergencyWithdraw
✓ Should emit event 'EmergencyWithdraw' (232ms)
setFeeAddress
✓ Should emit event 'SetFeeAddress'
✓ Should revert with 'setFeeAddress: FORBIDDEN'
updateEmissionRate
✓ Should emit event 'UpdateEmissionRate'
✓ Should revert with 'Ownable: caller is not the owner'
dev
✓ Should emit event 'SetDevAddress'
✓ Should revert with 'dev: wut?'

get Multiplier
✓ Should return number with BONUS_MULTIPLIER
Get pool length
✓ Should change pool length after add new pool
get pendingAra
✓ Should return number of Ara (43ms)
Withdraw rewards
✓ Should change receiver balance (354ms)

AraMasterChefMultiReward contract

Add pool
✓ Should emit event 'PoolAdded'
✓ Should add pool data in storage (38ms)
✓ Should add pool data in storage when 'withUpdate' parameter is equal 'false' (43ms)
✓ Should revert with 'Ownable: caller is not the owner'
✓ Should revert with 'add: invalid deposit fee basis points'
✓ Should revert with 'addPool: invalid reward token address'
Set pool
✓ Should emit 'SetPool' (83ms)
✓ Should change pool data in storage (84ms)
✓ Should change pool data in storage when 'withUpdate = false' (63ms)
✓ Should revert with 'Ownable: caller is not the owner' (51ms)
✓ Should revert with 'set: invalid deposit fee basis points' (54ms)
✓ Should revert with 'setPool: invalid reward token address' (45ms)
deposit
✓ Should emit event 'Deposit' (286ms)
withdraw
✓ Should emit event 'withdraw' (262ms)
✓ Should revert with 'withdraw: not good' (149ms)
emergencyWithdraw
✓ Should emit event 'EmergencyWithdraw' (238ms)
setFeeAddress
✓ Should emit event 'SetFeeAddress'
✓ Should revert with 'setFeeAddress: FORBIDDEN'
get Multiplier
✓ Should return number with BONUS_MULTIPLIER

Get pool length
✓ Should change pool length after add new pool (42ms)
Add rewar
✓ Should emit 'AddReward' (71ms)
✓ Should revert with 'addReward: invalid reward token address' (47ms)
✓ Should revert with 'addReward: invalid reward per block' (38ms)
✓ Should revert with 'Ownable: caller is not the owner' (41ms)
Update reward
✓ Should emit 'UpdateReward' (73ms)
✓ Should revert with 'Ownable: caller is not the owner' (41ms)
✓ Should revert with 'updateReward: invalid reward token address'
✓ Should revert with 'uupdateReward: invalid reward per block'
Pending Reward
✓ Should return pending rewards (447ms)
Transfer from strateg
✓ Should change receiver balance (584ms

AraFiToken

Deployment
✓ Transfers 1 mln to the contract deployer
✓ operator is the contract deployer
Operator
✓ operator can be changed

AraMigrator

Deployment
✓ Old token and new token set correctly
✓ Owner set correctly
✓ Receiver set correctly
Balance Getter
✓ Balance getter returns correct balance
Swap
✓ Swap works correctly
✓ Swap fails if allowance is not enough
✓ Swap fails if old token transfer fails
✓ Swap fails if new token transfer fails
Withdraw New Tokens
✓ Withdraw new tokens works correctly

- ✓ Withdraw new tokens fails if not owner
- ✓ Withdraw new tokens fails if new token transfer fails

Withdraw Old Tokens

- ✓ Withdraw old tokens works correctly
- ✓ Withdraw old tokens fails if not owner
- ✓ Withdraw old tokens fails if old token transfer fails

FractalMigrator

Deployment

- ✓ Old token and new token set correctly
- ✓ Owner set correctly

Balance Getter

- ✓ Balance getter returns true if the user owns the token
- ✓ Balance getter returns false if the user doesn't own the token
- ✓ Balance getter reverts if the token doesn't exist

Swap

- ✓ Swap works correctly
- ✓ Swap reverts if the user doesn't approve tokens
- ✓ Swap reverts if the user doesn't own the tokens
- ✓ Swap reverts if the contract doesn't have enough new tokens (67ms)

Withdraw Old Tokens

- ✓ Withdraw old tokens works correctly (62ms)
- ✓ Withdraw old tokens reverts if the user isn't the owner
- ✓ Withdraw old tokens reverts if the contract doesn't have enough tokens (55ms)

Withdraw New Tokens

- ✓ Withdraw new tokens works correctly (63ms)
- ✓ Withdraw new tokens reverts if the user isn't the owner
- ✓ Withdraw new tokens reverts if the contract doesn't have enough tokens (56ms)

Deposit New Tokens

- ✓ Deposit new tokens works correctly (69ms)
- ✓ Deposit new tokens reverts if the user isn't the owner
- ✓ Deposit new tokens reverts if the user doesn't have enough tokens (60ms)
- ✓ Contract won't allow deposits via transferFrom

HummusV2Strategy

governance 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266

Deployment
- ✓ Constructor parameters set correctly

Deposit
- ✓ Should deposit correctly
- ✓ Deposit works if want == 0

BalanceOfPool
- ✓ Should return correct balance

GetHarvestable
- ✓ Should return correct harvestable

GetHarvestableBonusToken
- ✓ Should return correct harvestable bonus token

Harvest
- ✓ Should harvest correctly

Withdraw
- ✓ Should withdraw correctly
- ✓ Should withdraw if specified amount lower than balance
- ✓ Should withdraw only by depositor

Withdraw other
- ✓ Should withdraw other correctly
- ✓ Should withdraw other by harvester correctly
- ✓ Should not withdraw wan
- ✓ Should not allow non-governance to withdraw

Withdraw all
- ✓ Should withdraw all correctly
- ✓ Should not allow non-governance to withdraw

Setters
- ✓ Should whitelist harvesters correctly
- ✓ Should revoke harvesters correctly
- ✓ Should not allow non-governance to whitelist harvesters
- ✓ Should set governance correctly
- ✓ Should not allow non-governance to set governance
- ✓ Should set depositor correctly
- ✓ Should not allow non-governance to set depositor

HummusV3Strategy

Deployment
✓ Constructor parameters set correctly
Deposit
✓ Should deposit correctly
✓ Deposit works if want == 0
BalanceOfPool
✓ Should return correct balance
GetHarvestable
✓ Should return correct harvestable
GetHarvestableBonusToken
✓ Should return correct harvestable bonus token
Harvest
✓ Should harvest correctly
Withdraw
✓ Should withdraw correctly
✓ Should withdraw if specified amount lower than balance
✓ Should withdraw only by depositor
Withdraw other
✓ Should withdraw other correctly
✓ Should withdraw other by harvester correctly
✓ Should not withdraw want
✓ Should not allow non-governance to withdraw
Withdraw all
✓ Should withdraw all correctly
✓ Should not allow non-governance to withdraw
Setters
✓ Should whitelist harvesters correctly
✓ Should revoke harvesters correctly
✓ Should not allow non-governance to whitelist harvesters
✓ Should not allow non-governance to revoke harvesters
✓ Should set governance correctly
✓ Should not allow non-governance to set governance
✓ Should set depositor correctly
✓ Should not allow non-governance to set depositor

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.