RADIANT

SMART CONTRACT AUDIT

zokyo

March 6th 2022 | v. 1.0

Security Audit Score

PASS

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

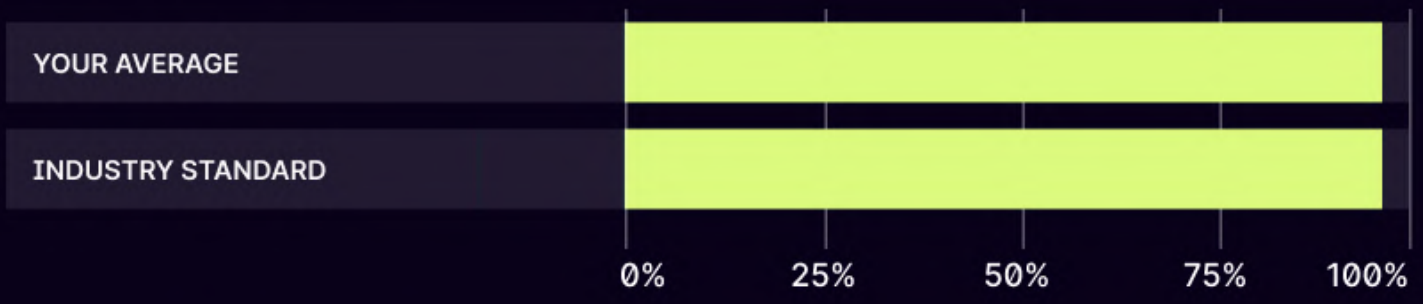
TECHNICAL SUMMARY

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

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

Contract Status

Testable Code



Contracts have a sufficient test coverage.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Radiant Capital 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 Radiant Capital repository:
https://github.com/radiant-capital/audit

Initial commit: 2ab06878c4b2498cdb4db81b9bd00f331920d993
Final commit: acd3e5284e5069ac23ee08edace5520e31957d58

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

- contracts\compounder\AutoCompounder.sol
- contracts\eligibility\Disqualifier.sol
- contracts\eligibility\EligibilityDataProvider.sol
- contracts\leverage\Leverager.sol
- contracts\oft\Migration.sol
- contracts\oft\RadiantOFT.sol
- contracts\price\PriceProvider.sol
- contracts\staking\ChefIncentivesController.sol
- contracts\staking\MFDstats.sol
- contracts\staking\MiddleFeeDistribution.sol
- contracts\staking\MultiFeeDistribution.sol
- contracts\staking\TokenVesting.sol
- contracts\stargate\StargateBorrow.sol
- contracts\zap\helpers\BalancerPoolHelper.sol
- contracts\zap\helpers\LiquidityZap.sol
- contracts\zap\helpers\UniswapPoolHelper.sol
- contracts\zap\LockZap.sol
- contracts\protocol\lendingpool\LendingPool.sol
- contracts\protocol\tokenization\AToken.sol
- contracts\protocol\tokenization\IncentivizedERC20.sol
- contracts\protocol\tokenization\StableDebtToken.sol
- contracts/bounties/BountyManager.sol
- contracts/oracles/

During the audit, Zokyo Security ensured 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, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

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

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

Executive Summary

During the audit, the Zokyo Security team reviewed the smart contracts the Radiant Capital team provided. Protocol represents a fork of the lending protocol with additional functionality such as rewards distribution via ChefIncentivesController, fee distribution, liquidity zapping, disqualification system, and cross-chain borrowing via Stargate and LayerZero. Thus, the goal of the audit was to verify the correctness of the forked lending protocol, validate the business logic of smart contracts, check the integration with 3rd part protocols, check the smart contract against the list of vulnerabilities, validate that code corresponds to Solidity best practices in terms of code quality and gas usage.

The Zokyo Security team carefully audited all the contracts during the manual audit. The team found several critical and high-severity issues. The first issue refers to the wrong boolean comparison in MultiFeeDistribution smart contract. Due to this, users were unable to withdraw staked assets. The second severity issue refers to role management, while the third one was connected to possible underflow in ChefIncentivesController. Other issues were connected to the ability of the owner to withdraw staking tokens, possible reverts during staking and withdrawals in MultiFeeDistribution, iteration through storage arrays, stuck ETH, absence of validations, and gas optimizations. Auditors also raised a price manipulation issue and some informational issues connected to the disqualification system. The Radiant Capital team has prepared new contracts connected to the disqualification system and TWAP price oracle to address these issues. Zokyo Security team has also carefully audited and tested new contracts and prepared several information issues about oracles. The radiant Capital team has successfully fixed them as well. TWAP oracles successfully withstand all price manipulations. However, it is recommended to update such oracles as frequently as possible.

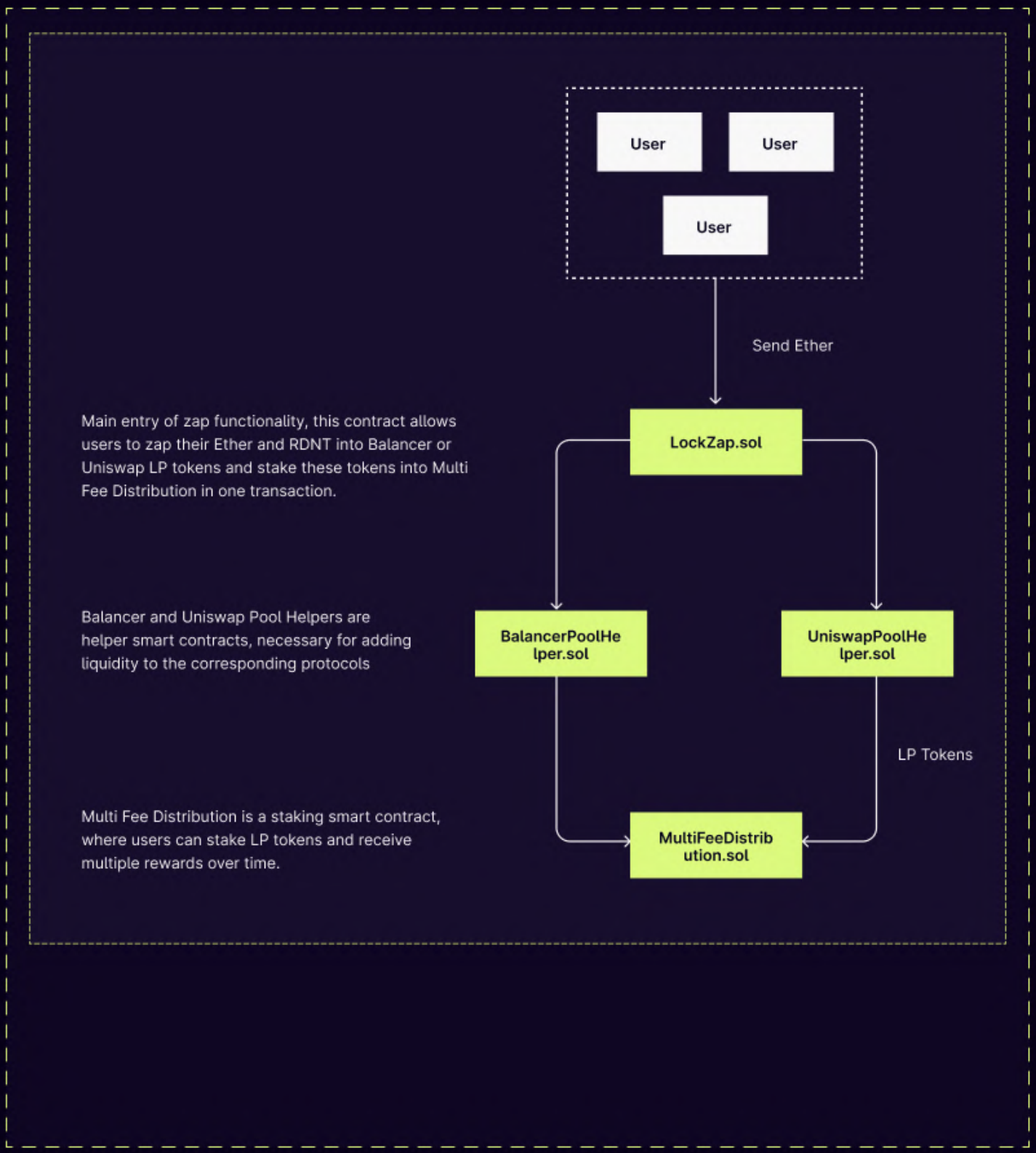
As a part of Zokyo Security's assistance, auditors have also prepared a set of unit tests to ensure the safety of smart contracts. All the crucial parts of the protocol were carefully tested. In order to validate the safety of interaction with other protocols, such as Stargate, auditors have prepared a fork test.

The Radiant Capital team successfully fixed most of the issues. Although the code of smart contracts is quite complicated, auditors have validated all the essential aspects of it.

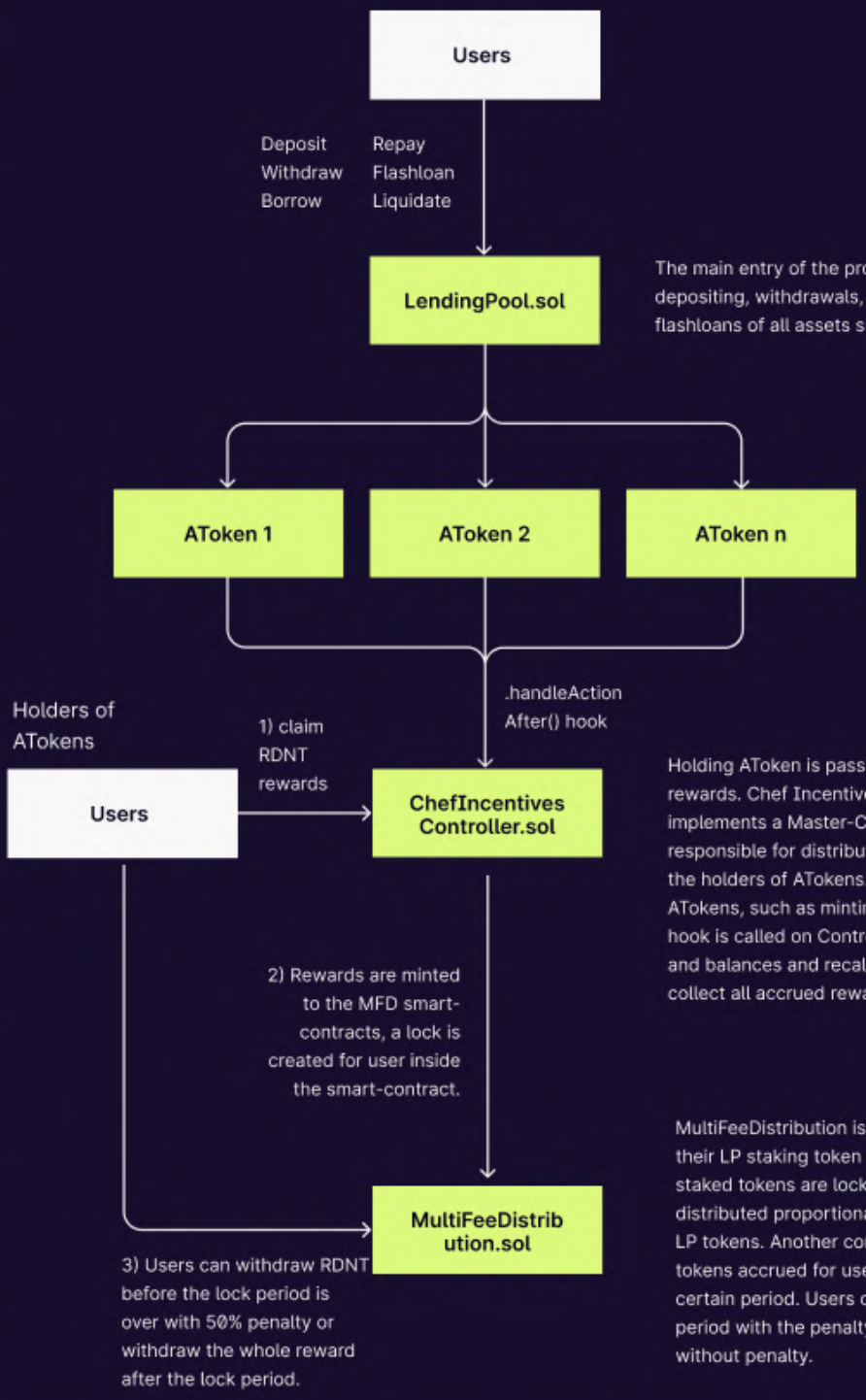
Executive Summary

The current mark reflects significant major changes in the protocol performed at the end of the audit, the lack of documentation for the protocol, high complexity and size of the codebase compared to the current level of readability and transparency, few inconsistencies for locks and staking verified by the team and left on their responsibility, lack of documentation for the oracles plugging-in. Despite the row of critical and high-risk issues being connected to all major security threats (flashloans, frontrun, overflow, unconvincing oracles system) - the team resolved all of them, so the contract corresponds to the necessary security level.

Protocol flow



Protocol flow



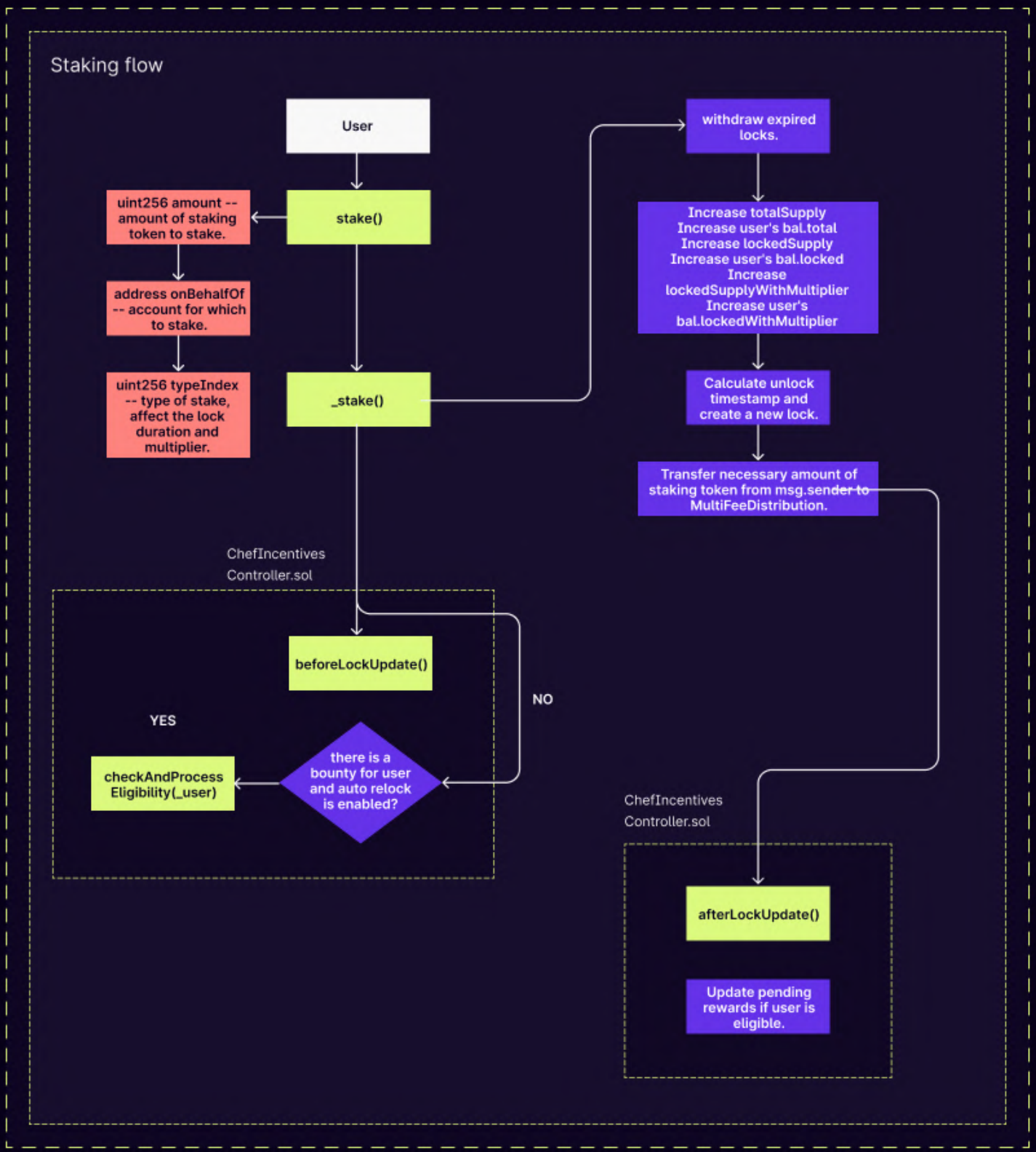
The main entry of the protocol, this contract is responsible for depositing, withdrawals, borrowing, repaying, and liquidation and flashloans of all assets supported by the Radiant Capital.

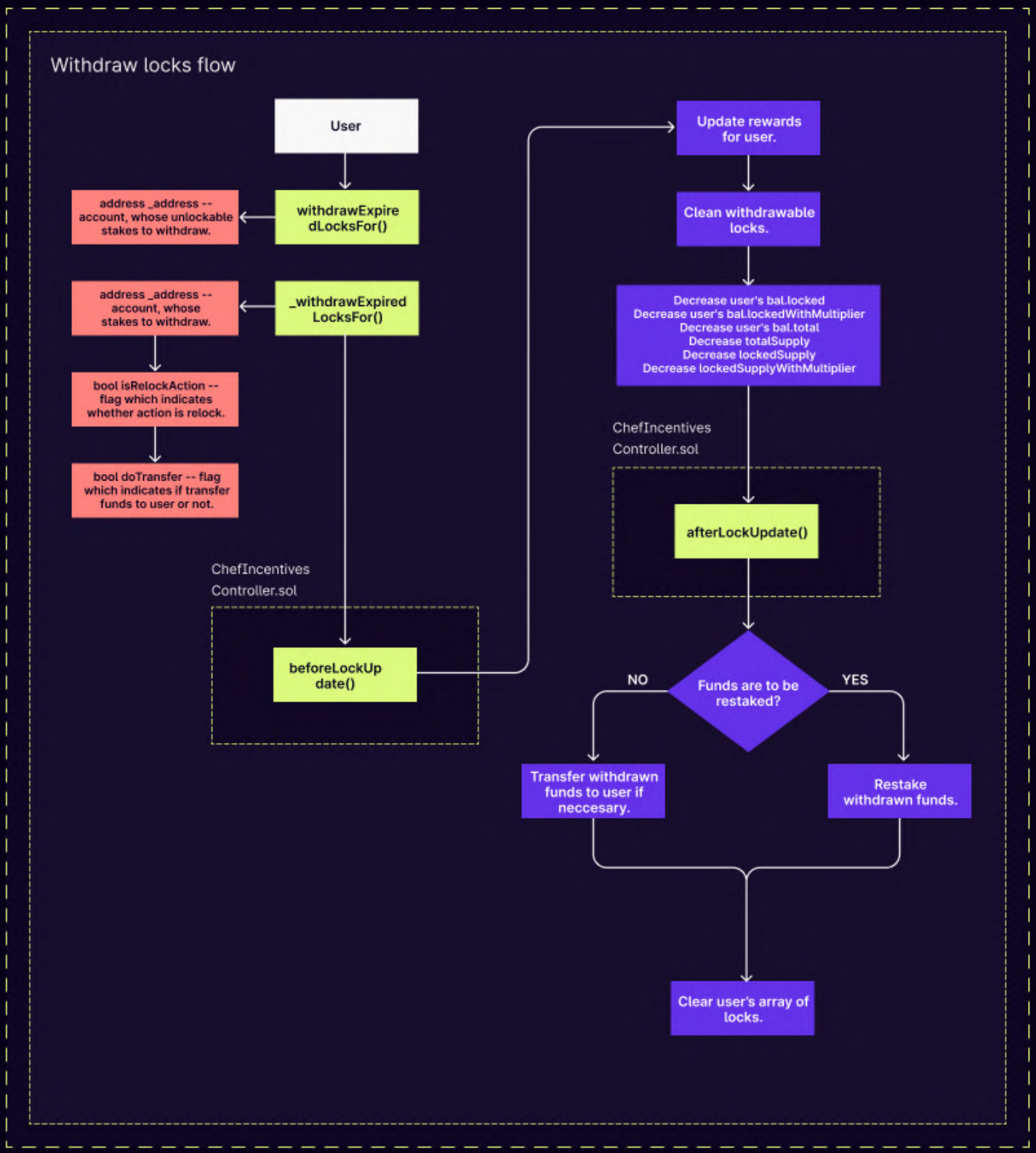
ATokens are necessary to prove that users have provided underlying tokens to the protocol. ATokens are ERC20, which can be obtained by depositing underlying assets into LendingPool and can be redeemed into underlying assets. It can only be minted and burned by the Lending Pool smart contract.

Holding AToken is passively incentivized by granting additional rewards. Chef Incentives Controller is a smart-contract that implements a Master-Chief-like rewards distribution system, responsible for distributing additional RDNT rewards amongst the holders of ATokens. Thus on each update of balances in ATokens, such as minting, burning, and transferring, a special hook is called on Controller, which marks the change of supply, and balances and recalculates rewards. Protocol users can then collect all accrued rewards from the Controller smart contract.

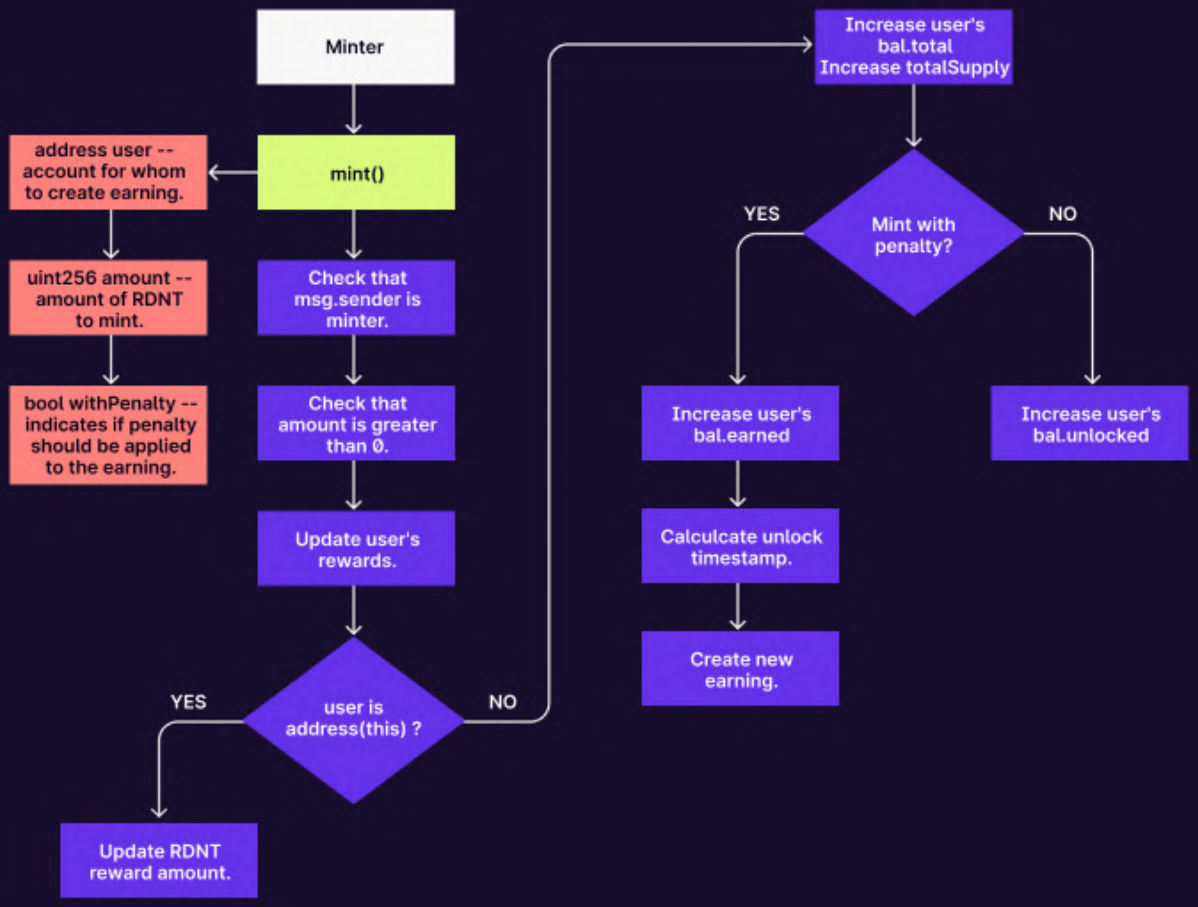
MultiFeeDistribution is a contract that allows users to stake their LP staking token and earn rewards in multiple tokens. All staked tokens are locked for a certain period, and rewards are distributed proportionally among all the users who have staked LP tokens. Another contract functionality is to lock RDNT tokens accrued for users in the Chef Incentives Controller for a certain period. Users can withdraw their rewards during this period with the penalty or withdraw RDNT after the lock period without penalty.

MultiFeeDistribution.sol

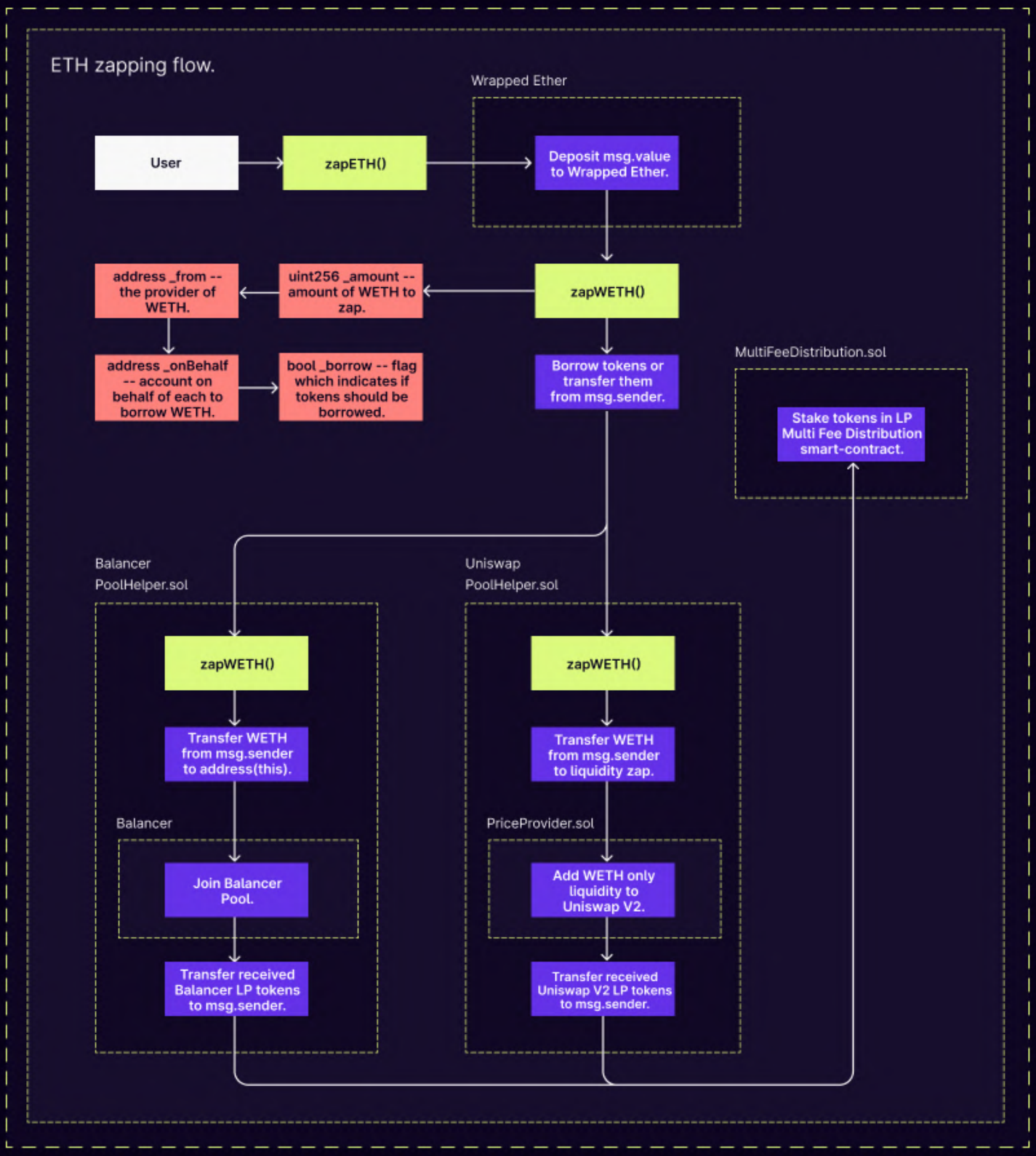



Withdraw locks flow
User
withdrawExpire dLocksFor()
address _address -- account, whose unlockable stakes to withdraw.
_withdrawExpired LocksFor()
address _address -- account, whose stakes to withdraw.
bool isRelockAction -- flag which indicates whether action is relock.
bool doTransfer -- flag which indicates if transfer funds to user or not.
ChefIncentives Controller.sol
beforeLockUp date()
Update rewards for user.
Clean withdrawable locks.
Decrease user's bal.locked
Decrease user's bal.lockedWithMultiplier
Decrease user's bal.total
Decrease totalSupply
Decrease lockedSupply
Decrease lockedSupplyWithMultiplier
ChefIncentives Controller.sol
afterLockUpdate()
Funds are to be restaked?
NO
YES
Transfer withdrawn funds to user if neccesary.
Restake withdrawn funds.
Clear user's array of locks.

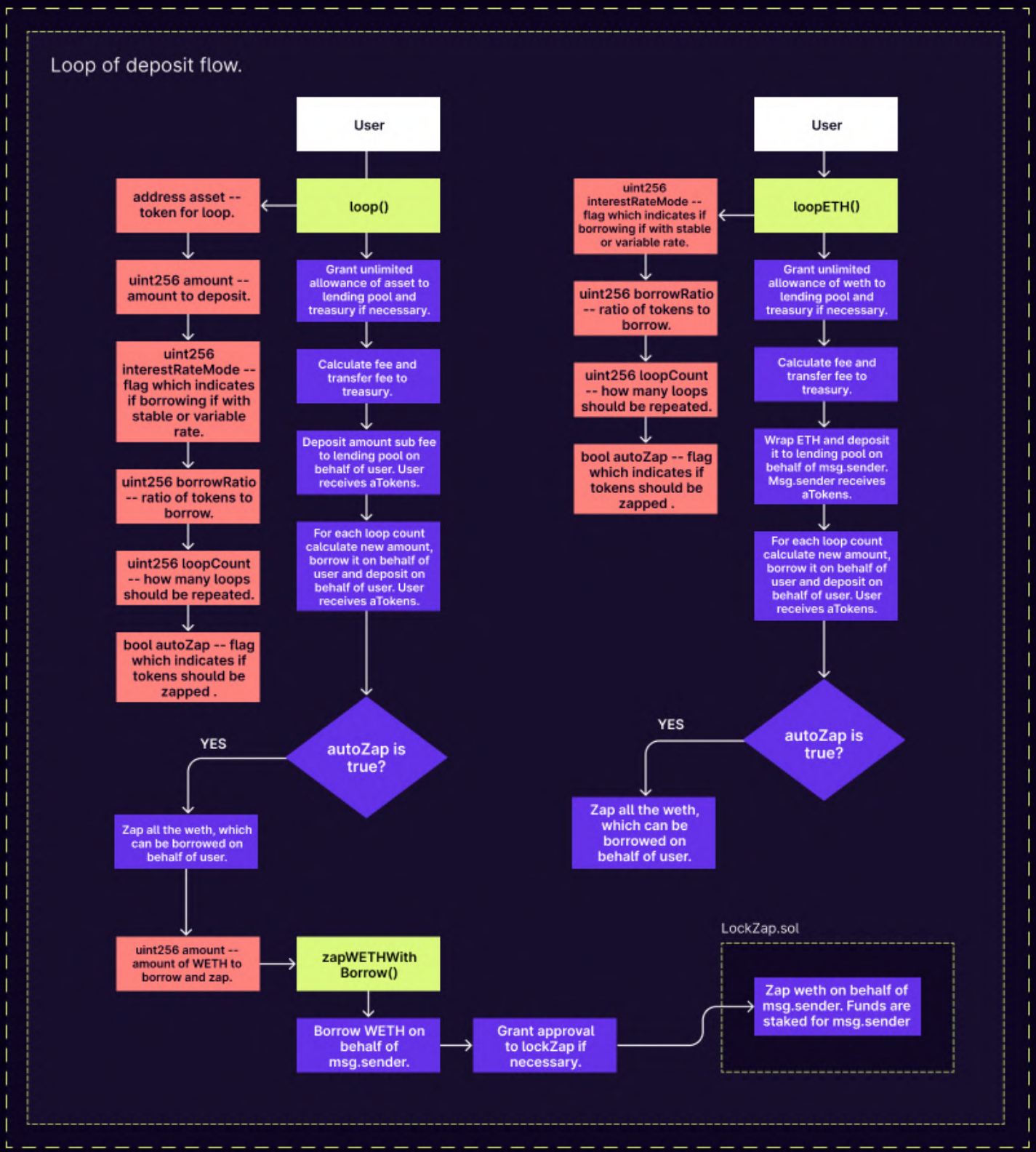
Mint earnings flow.



Withdraw earnings flow


ETH zapping flow.
Wrapped Ether
User
zapETH()
Deposit msg.value to Wrapped Ether.
address _from -- the provider of WETH.
uint256 _amount -- amount of WETH to zap.
zapWETH()
address _onBehalf -- account on behalf of each to borrow WETH.
bool _borrow -- flag which indicates if tokens should be borrowed.
Borrow tokens or transfer them from msg.sender.
MultiFeeDistribution.sol
Stake tokens in LP Multi Fee Distribution smart-contract.
Balancer PoolHelper.sol
Uniswap PoolHelper.sol
zapWETH()
zapWETH()
Transfer WETH from msg.sender to address(this).
Transfer WETH from msg.sender to liquidity zap.
Balancer
PriceProvider.sol
Join Balancer Pool.
Add WETH only liquidity to Uniswap V2.
Transfer received Balancer LP tokens to msg.sender.
Transfer received Uniswap V2 LP tokens to msg.sender.

Leverager.sol

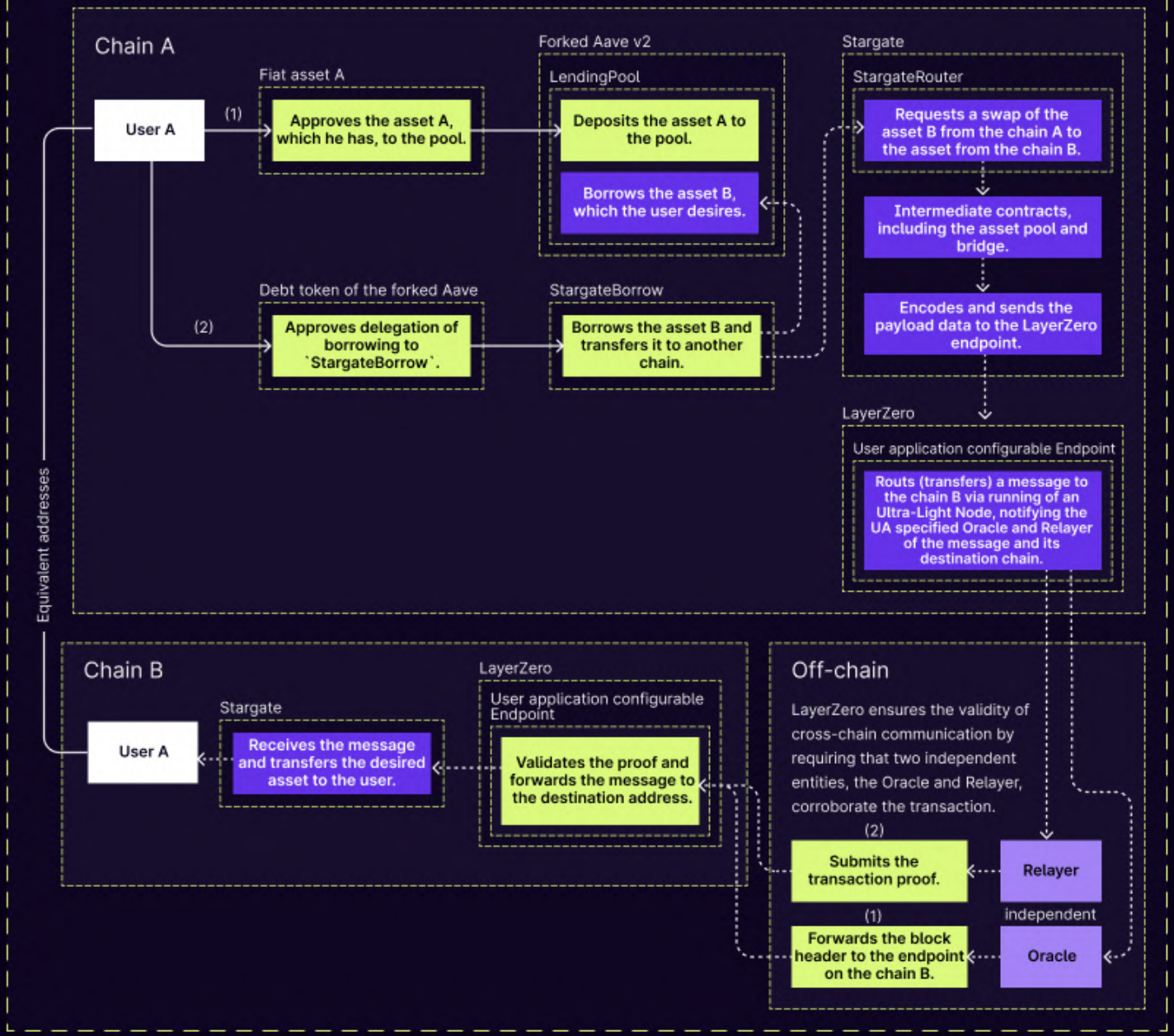


Stargate integration

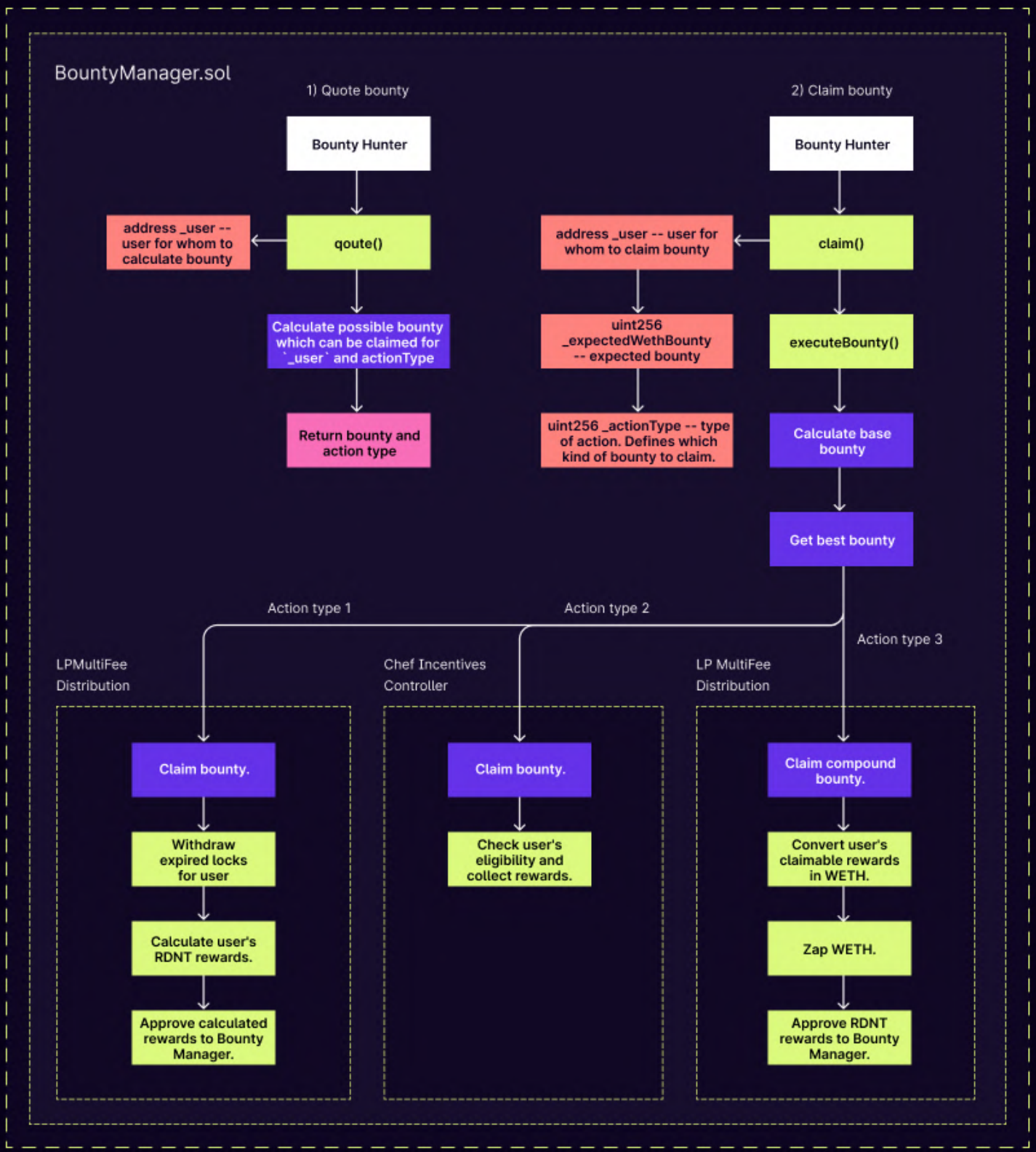
Stargate allows assets to be transferred from one chain to another and exchanged at the same time if necessary.

The Radiant Capital protocol allows a user to borrow desired assets, by reserving assets that he has, using its own Aave fork, and transferring them into another chain.

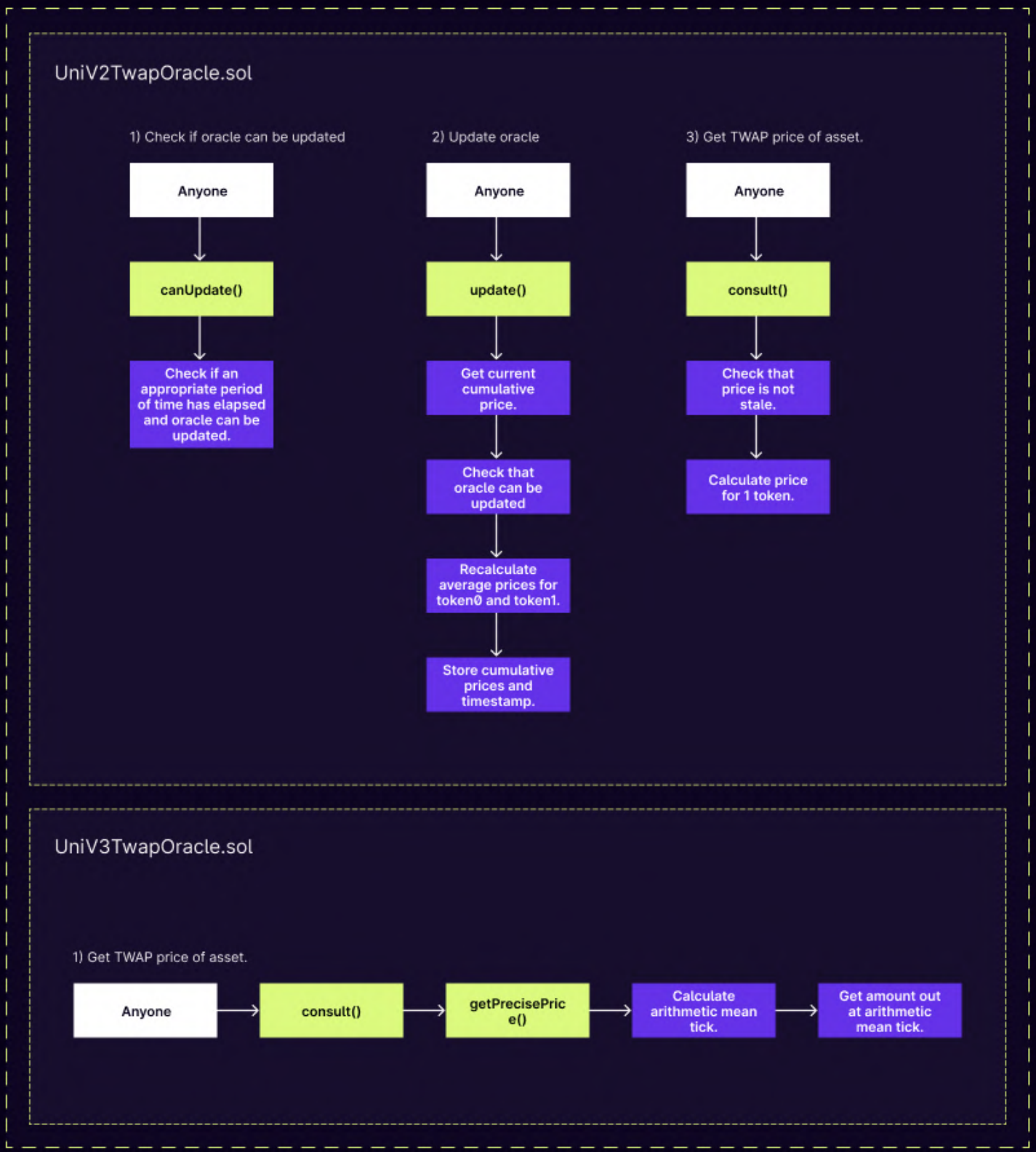
Note. Radiant may charge an additional fee for such a transfer as a percentage of the borrowed amount.



Radiant Capital scheme



Radiant Capital scheme



STRUCTURE AND ORGANIZATION OF THE DOCUMENT

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

Critical

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

High

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

Medium

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

Low

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

Informational

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

COMPLETE ANALYSIS

CRITICAL-1 | RESOLVED

Wrong comparison condition.

MultiFeeDistribution.sol: _withdrawExpiredLocksFor(), line 1147.
The "if " statement validates that in case relock is not disabled by the user during the relock action transaction, withdrawn funds should be restaked. However, the contract compares variable 'isRelockAction' to "false" instead of "true", which doesn't correspond to the logic of the contract. For example, if _withdrawExpiredLocksFor() is called within function withdrawExpiredLocksFor(), it will have parameter 'isRelockAction' set to "false" which means that funds should be transferred to the user, not restaked. However, due to the wrong condition, funds will be restaked. The issue is marked as critical since it is currently impossible for users to withdraw expired locks to their balances.

Recommendation:
Verify the comparison logic OR compare 'isRelockAction' to true instead.

Post-audit.
Comparison condition wasn't changed. Thus, function withdrawExpiredLocksFor() still performs a relock action, however another function was added to perform withdrawing without relock.

CRITICAL-2 RESOLVED

Functions are not restricted.

1) RadiantOFT.sol: setLpMintComplete(), setMinter().

Functions are not restricted, meaning anyone can call and stop the minting or set a minter address. While function setMinter() can be called only once, there are no deployment scripts to verify that the team will call this function right after the deployment. Thus, anyone can call setMinter() before the deployer and become a minter, and anyone can call setLpMintComplete() and stop minting at any time.

2) ChefIncentivesController.sol: handleActionAfter().

The function used to be restricted, so only valid RTokens or LP multiFeeDistribution can be called. However, at a commit 3df651a62b0446ff2113fe3f1a8506e3f7fff0f7 function is not restricted and can be called by anyone.

Recommendation:
Restrict the function, so that only certain addresses with specific role can call them.

Post-audit:
Functions were removed and the minting system was changed. Now function mint() can be called only once and is called in the deployment script.

CRITICAL-3 RESOLVED

Possible underflow when handling after-actions for token.

When ATokens are minted, transferred, or burnt, ChefIncentivesController.handleActionAfter() is called, and it updates balances and total supplies in mappings `userInfo` and `poolInfo`.
However, during disqualification, the contract calls ChefIncentivesController.disqualifyUser() when the user's balance is set to `0` (by calling _handleActionAfterForToken() where _balance = `0`).

However, ATokens are still present on the user's balance. When he tries to transfer or burn them, the contract calls handleActionAfter() and performs a subtraction from `0`. That might cause an underflow, granting users an enormous balance and affecting the correctness of rewards distribution.

Recommendation:
Do not set userInfo[user].amount to `0` during disqualification in order to avoid underflows when user transfers or burns his tokens.

Post-audit:
Subtraction is validated now. Also since user's current amount is subtracted from total supply, it won't cause an underflow.

HIGH-1 | RESOLVED

Owner is able to withdraw staking token.

MultiFeeDistribution.sol: recoverERC20().
The owner can directly access users' funds and withdraw their tokens anytime since the owner can't recover only reward tokens. As a result, in the case of the private key exploit (of an owner account), users' funds can be withdrawn directly from the contract. That's why it is recommended to validate that the provided `tokenAddress` is not a staking token, in order to exclude a centralization risk.

Recommendation:
Validate that `tokenAddress` is not a staking token in the function.

Post-audit.
Function recoverERC20() was removed.

HIGH-2 RESOLVED

Possible frontrun via flashloan attack.

UniswapPoolHelper.sol: swap(), line 188.
LiquidityZap.sol: addLiquidityWETHOnly(), line 113; addLiquidityETHOnly(), line 147.
BalancerPoolHelper.sol: swap(), line 311.
The contract does not check the actual `amountOut` value during the tokens swap. Usually, the swap in protocols such as Uniswap or Balancer requires a parameter `minimumLiquidity` and checks an actual output amount of token so that it will be not less than `minimumLiquidity`. This parameter is essential in order to protect users from too high slippage and possible manipulations of the pool's reserves through frontrun. It is highly recommended to check the `amountOut` and protect users from any possible pool manipulations.

Recommendation:
Pass `minimumLiquidity` as a function parameter and check that `amountOut` is greater or equal to `minimumLiquidity`.

Post-audit.
The main function of LockZap.sol contains a slippage check. As for other contracts, it was verified by the Radiant team, that users shouldn't interact with LiquidityZap, BalancerPoolHelper, UniswapPoolHelper directly.

HIGH-3 | RESOLVED

Price can be manipulated via flashloans.

PriceProvider.sol: update(), getTokenPrice(), getLpTokenPrice().
Since the contract calculates the price based on a single DEX (Uniswap or Balancer), it can be manipulated via flashloans. There are several parts of the protocol where price manipulation is profitable for an attacker.

1. Disqualifier.sol: getBaseBounty().

When a provided user can be disqualified, the contract calculates a bounty for him based on the price of RDNT. Though the calculated bounty can't exceed `maxBaseBounty`, the attacker can manipulate the price to receive the maximum allowed bounty.

2. RadiantOFT.sol: _getBridgeFee().

The contract utilizes price to calculate the fee for using the bridge. In this case, an attacker can manipulate the price to pay lower fees.

3. EligibilityDataProvider.sol: _lockedUsdValue().

Price is used to calculate the locked user's amount. In this case, an attacker may use flashloan to manipulate the value of the user's locked funds and affect the disqualification of user.
The issue is marked as high due to point 3, where users can be affected and disqualified due to price manipulation.

Recommendation:
Consider using off-chain oracles for price calculation OR use several sources for retrieving price (such as Uniswap, Balancer, Curve) and calculate a Volume Weighted Average Price OR calculate a historical price price based on a single source of price (TWAP). This can help reduce the chances of flashloan attack on the PriceProvider.

Post-audit:
TWAP oracles based on Uniswap V2 and Uniswap V3 were implemented.

HIGH-4 | RESOLVED

Transfer without enough token on contract balance.

Leverager: loop()
The function will revert with isBorrow parameter = true since it cannot transfer fee without any tokens on balance. Also it seems like it is charging funds twice: before the for() cycle and in it.
Recommendation:
Transfer fee when a contract has enough tokens on balance.

MEDIUM-1 RESOLVED

Staking and withdrawal operations might be blocked.

MultiFeeDistribution.sol: _stake(), line 644, _withdrawExpiredLocksFor(), line 1134.
During staking and withdrawing funds, a `beforeLockUpdate` hook is called on the Incentives Controller. This hook checks if a user is to be disqualified. For this purpose, the contract performs another external call to Disqualifier.sol, function processUser(). Inside this function, the contract calls an internal function of Disqualifier, _processUserWithBounty(). It has a "require" which will revert if the storage variable `DISABLED` is set to true. Thus due to this statement on Disqualifier.sol, staking and withdrawing of tokens on MultiFeeDistribution.sol might be blocked.
Further, in _processUserWithBounty(), an external call is performed to ChefIncentivesController.disqualifyUser(), where there are two checks which can also block the operations (Lines 489, 491).

Recommendation:
Do not revert a whole stake or withdraw transaction due to require statements in the Disqualifier.sol and ChefIncentivesController.sol.

Post-audit:
The team removed validations that might prevent staking and withdrawing. However, if a certain user is ineligible for rewards, a bounty should be claimed for him before staking or withdrawing.

MEDIUM-2 RESOLVED

User lock and earning might contain empty elements.

MultiFeeDistribution.sol: _cleanWithdrawableLocks(), line 1123.
MultiFeeDistribution.sol: withdraw(), line 775
After the lock withdrawal, the contract deletes it with "delete" operator. However, deleting elements of an array with this operator only sets the value of an element to `0`, and the element will still be present in the array. Due to this, the array `userLocks[user]` might contain empty elements, iteration through which will increase gas spending. This is why it is recommended to shift elements in the array to the left by 1 when any element is removed from the array. The issue is marked as medium-risk, since if there are too many elements in an array, a transaction might revert due to an "out of gas" error and even not fit in one block.

Recommendation:
Shift elements in the array when removing an element, so that there are no empty elements in the array.

Post-audit:
Both “userLocks” and “userEarnings” are cleaned now.

MEDIUM-3 | RESOLVED

Tokens are not approved to a new pool helper.

LockZap.sol: setPoolHelper().
After the LockZap.sol initialization, it requires granting an unlimited allowance to _poolHelper in WETH and RDNT tokens. However, when a new pool is set with the function setPoolHelper(), allowance is not granted to a new pool helper. Thus the protocol couldn’t operate with any new pool helper due to the lack of allowance.

Recommendation:
Consider granting allowance before each transfer **OR** grant unlimited allowance in the setter as well.

Post-audit:
Approval is performed before each transfer now.

MEDIUM-4 RESOLVED

Iteration though the whole storage array.

MultiFeeDistribution.sol: _cleanWithdrawableLocks(), line 1119; withdraw(), line 754.
After the withdrawal of locks or earnings, the contract performs an iteration through the whole storage array. For example, if a user creates a significant amount of locks, iteration might consume more gas than can fit in one transaction. Thus, the user's funds might get stuck. The issue is marked a medium-risk since it might affect certain users, not the whole protocol. However, it is still recommended not to allow users to create infinite elements in the array to mitigate this issue.

Recommendation:
Consider restricting the maximum amount of elements in storage arrays `userEarnings[user]` and `userLocks[onBehalfOf]` **OR** set a minimal amount of tokens, which can be locked or minted per one time, so that a user can't create too many locks with low values.
Post-audit:
Restriction was added in function _cleanWithdrawableLocks(). User will have to call function multiple times if he has a great amount of locks.

MEDIUM-5 | RESOLVED

ETH might get stuck on the contract's balance.

1) LiquidityZap.sol: standardAdd().
BalancerPoolHelper.sol: zapTokens().
Function is marked as payable, allowing users to send Ether when calling it. However this function never interacts with msg.value, thus sending Ether to it has no effect. Thus any Ether, sent with this function, will be stuck on the contract.
2) LockZap.sol: zapFromVesting(), line 118.
In case parameter `_borrow` is true, any sent Ether will get stuck on the contract's balance. Thus, in case `_borrow` is true, it should be validated that no Ether is sent with the function.

Recommendation:
1. Remove payable modifier OR add interaction with Ether in the function.
2. Add a validation that in case `_borrow` is true, msg.value is equal to 0.

MEDIUM-6 | RESOLVED

Unlimited allowance.

LockZap.sol: constructor(), lines 53-54; setLpMfd(), line 66.
Making an unlimited allowance of tokens might be potentially dangerous and lead to funds loss in case of exploitation. Though approval is performed on protocol contracts (UniswapPoolHelper.sol or BalancerPoolHelper.sol and MultiFeeDistribution.sol) these contracts are upgradeable. This is why approving tokens before each transfer on the amount necessary for a particular transfer is recommended.

Recommendation:
Approve tokens, necessary for transfer before each transfer operation instead of granting an unlimited allowance.

Post-audit.
Unlimited allowance was removed.

MEDIUM-7 | RESOLVED

Non-withdrawable locks can be withdrawn.

LockZap.sol: setPoolHelper().
After the LockZap.sol initialization, the contract grants an unlimited allowance to _poolHelper in WETH and RDNT tokens. However, when a new pool is set with the function setPoolHelper(), allowance is not granted to a new pool helper. Thus the protocol couldn't operate with any new pool helper due to the lack of allowance.

Recommendation:
Check the last element of the array with index `locks.length - 1` in "if".

Post-audit:
Last element of locks is not checked now. User has to call function several times to withdraw a large amount of locks.

LOW-1 | RESOLVED

Share variables lack validation.

Disqualifier.sol: setHunterShare().
EligibilityDataProvider.sol: setRequiredEthRatio().
MiddleFeeDistribution.sol: setLpLockingRewardRatio(), setOperationExpenses().
Leverager.sol: setFeePercent().
Leverager.sol: loop(), parameter `borrowRatio` (Should be validated not to exceed `10 ** BORROW_RATIO_DECIMALS`).
RadiantOFT.sol: setFeeInfo().
StargateBorrow.sol: setXChainBorrowFeePercent().
Some of the variables act as shares or use shares during calculations, however such variables lack validation, that they don't exceed 100%. For example, in Disqualifier.sol when setting HUNTER_SHARE, the contract should validate that the new value will not exceed 10000 (As division is performed with 10000 as a denominator in line 277). Issue is marked as low-risk since only owner can set these variables, though in case of invalid value it will affect the correctness of calculations in the protocol.

Recommendation:
Validate that share variable doesn't exceed 100%.

LOW-2 RESOLVED

Wrong hunter address is used.

Disqualifier.sol: processUserWithBounty().
Parameter `_hunter` is never used. Based on the logic of the function and a similar function processUsersWithBounty(), where parameter `_hunter` is passed to the internal function, the same should happen in processUserWithBounty().

Recommendation:
Pass parameter `_hunter` to the internal function instead of msg.sender OR remove parameter `_hunter`.

Post-audit:
Contract was removed.

Contracts can be initialized with wrong parameters.

MultiFeeDistribution.sol: initialize().
TokenVesting.sol: initialize(), _notifyUnseenReward()

1. MultiFeeDistribution.sol: No parameters are being validated in initialize(). This might be a serious problem as addresses "mfdStats" and "priceProvider" do not have setters themselves as the variables REWARDS_DURATION, REWARDS_LOOKBACK, DEFAULT_LOCK_DURATION do not have setter either and are set only in initialize() method.
2. TokenVesting.sol: rdntToken is not validated against zero address in initialize() method and there is no setter for rdntToken variable.
3. EligibilityDataProvider.sol, Disqualifier.sol, Leverager.sol, PriceProvider.sol: all address parameters should be validated against zero addresses in constructor and initializer
4. MultiFeeDistribution.sol: _notifyUnseenReward(): At line 965 subtraction REWARDS_DURATION - REWARDS_LOOKBACK might underflow in case REWARDS_LOOKBACK is greater than REWARDS_DURATION. Thus the initiaize() function should verify that subtraction won't result in an underflow.
5. RadiantOFT.sol: parameters _endpoint and _treasury in constructor should be validated against zero address. _fee should be validated not to exceed 1e4.
6. StargateBorrow.sol: all address parameters. `_xChainBorrowFeePercent` not to exceed 1e4.

Recommendation:
Validate parameters in initialization of contracts.

Post-audit:
All validations were added.

LOW-4 | RESOLVED

Vesting schedule can be overwritten and parameters are not validated.
TokenVesting.sol: addVest().

The owner is able to overwrite vesting, which is already ongoing. Any tokens from the previous vesting that weren't already claimed will get stuck on the contract. Though only the owner can call this function, already existing vestings can still be overwritten, leading to the loss of funds. Also, to prevent invalid vesting creation, it is recommended to validate parameters `_amount` and `_duration` against zero values, as in case `_duration` is zero, RDNT tokens still will be transferred. However, the claimer couldn't claim them due to validation in function claim(), line 53.

Recommendation:
Do not allow to overwrite vesting schedules which are already in progress and validate parameters `_amount` and `_duration` not to be zero values.

Post-audit:
Parameters are validated and active vesting can't be overwritten.

INFO-1 RESOLVED

Storage variables are never used.

AutoCompounder.sol: `rdntToken`, `lpTokenAddress`, `baseToRdnt`.
EligibilityDataProvider.sol: `baseTokenPriceInUsdProxyAggregator`.
Disqualifiers.sol: `treasury`.
MiddleFeeDistribution.sol: `minters`, `mintersAreSet`.
Variables are never used in the contract's logic, and some variables are even never set. `baseToRdnt` is initialized with zero addresses. The unused variable can mean that the contract is unfinished, so it is recommended not to have such variables.

Recommendation:
Remove unused variables **OR** finish contract's logic where variables will be used **OR** verify that these variables are necessary to other smart-contracts or the Dapp.

INFO-2 | RESOLVED

Lack of events.

In order to track the historical changes of essential storage variables, it is recommended to emit events in setters on every change of variable. Thus the following setter function should emit an event.

1. AutoCompounder.sol: addRewardBaseTokens(), setRoutes().
2. Disqualifier.sol, EligibilityDataProvider.sol, MiddleFeeDistribution.sol, Leverager.sol, RadiantOFT.sol, StargateBorrow.sol: all functions which start with "set".

Recommendation:
Emit events in setter functions.

INFO-3 | RESOLVED

Storage constants should be used.

Disqualifier.sol: function _processUserWithBounty(), line 184; bountyForUser(), line 277, value "10000".
EligibilityDataProvider.sol: requiredUsdValue(), line 159, value "1e4".
MiddleFeeDistribution.sol: mint(), line 171; forwardReward(), line 197, 204, value "1e4".
MFDstats.sol: addTransfer(), line 80, 90, value "1e4".
LockZap.sol: lines 105, 107, values "10000", "9500".
Leverager.sol: lines 107, 117, 141, 154, value "1e4".
RadiantOFT.sol: _getBridgeFee(), line 114, value "1e4".
StargateBorrow.sol: getXChainBorrowFeeAmount(), line 105, value "1e4".
In order to increase readability of the contracts, it is recommended to use storage constants instead of values directly.

Recommendation:
Use storage constants instead of values directly in code.

INFO-4 | RESOLVED

Same tokens can be added to eligible tokens array.

EligibilityDataProvider.sol: addToken().
The owner is able to set the same token as eligible multiple times. Thus the same token will be added to the array `eligibleTokens` multiple times. The issue is marked as info since it doesn't affect the contract, as array is not used in any of its logic. However, it might be crucial for other parts of the protocol that the array doesn't contain repeatable tokens, so it is recommended to restrict the same token to be added to the array multiple times.

Recommendation:
Validate that `token` is not already added to array of eligible tokens.

Post-audit.
Function was removed.

INFO-5 | RESOLVED

Comments don't correspond to the code.

1. MultiFeeDistribution.sol: initialize().

The commentary section stated that the first token in the array of rewards must be a staking token. As a function has a parameter '_stakingToken', one may assume this token must be the first one in the rewards array (based on comments). However, another token, '_rdntToken', is pushed to the array instead. Thus, this doesn't correspond to the comments and should be verified by the team.
EligibilityDataProvider.sol: lastEligibleTime().
The commentary section stated that the function returns locked RDNT and LP token values in ETH. However, the function's name and the return variable's name imply that the function returns the last eligible timestamp.

Recommendation:

1. Verify that '_rdntToken' should be the first one in the rewards array and update comments OR verify that '_rdntToken' and '_stakingToken' are the same token OR update the code and push '_stakingToken' to the rewards array.
2. Verify the correctness of the return value in function.

INFO-6 RESOLVED

Never used imports in multiple contacts.

Disqualifier.sol :
ILendingPool, LockedBalance, IUniswapV2Router02, IUniswapV2Factory, IUniswapV2Pair, IChainlinkAggregator, IAToken.
PriceProvider.sol: IERC20, SafeERC20.
Contracts are imported in files, however they are never used.

Recommendation:
Remove unnecessary imports OR use them if needed.

Post-audit.
All unused imports were removed.

INFO-7 | RESOLVED

Commented code.

MultiFeeDistribution.sol: _withdrawExpiredLocksFor(), line 1155.
LiquidityZap.sol: standardAdd(), line 189; _addLiquidity(), lines 235-236.
StargateBorrow.sol: setDAOTreasury(), line 94.
Pre-production smart-contract should not contain commented code as it might mean that part of the contract's logic is unfinished.

Recommendation:
Remove or uncomment commented code.

INFO-8 RESOLVED

Available borrows Ether is not calculated for an actual borrower. LockZap.sol: _executeBorrow().

Function _executeBorrow() has a parameter `_onBehalf` for which the availability of ETH to borrow is checked. However, the contract performs the actual borrow for msg.sender, which might mismatch with `_onBehalf` parameter in case the function is called within zapWETH(). Because the user can pass an arbitrary `_onBehalf` to zapWETH() function, removing this parameter and using msg.sender directly is recommended. The issue is marked as info, since LendingPool.sol still won't let borrow more than is available for msg.sender.

Recommendation:
Remove parameter `_onBehalf` and use msg.sender directly so that availability of Ether is checked for correct user.

Post-audit:
Availability is checked for `_onBehalf` address. Since liquidity is staked for `_onBehalf`, this might not be an issue that anyone can pass `_onBehalf` and zap his funds (In case `_onBehalf` allowed LockZap to perform borrow on his behalf.)

INFO-9 RESOLVED

User might have access to tokens, stored on contract's balance. LockZap.sol: zapWETH().

In case the user passes a parameter `_from` equal to the address of LockZap.sol, WETH won't be transferred from msg.sender to address(this), and the contract will use WETH stored on LockZap's balance. The issue is marked as info, since the contract isn't supposed to store funds, however, such functionality should be verified.

Recommendation:
Verify that users should be able to pass `_from` as the address of LockZap.sol and use funds which could be stored on contract's balance.

Post-audit:
address(this) can't be passed when zapWETH is called() now.

Disqualification mechanism should be verified.

Disqualifier.sol: _processUserWithBounty().
ChefIncentivesController.sol: disqualifyUser(), _isEligible().

In Disqualifier.sol (lines 167-168), the user will be disqualified from ChefIncentivesController.sol if the user's unlockable > 0 and either relock is disabled or the user's locked value is less than his collateral value in LendingPool.

Thus, the user is considered ineligible in case `lastEligibleTime > block.timestamp && !eligibilityProvider.isEligibleForRewards(_user)` (Disqualifier.sol, line 163.). However, in ChefIncentivesController.disqualifyUser(), the user is considered ineligible for rewards if `lastEligibleTime < block.timestamp OR !eligibilityProvider.isEligibleForRewards(_user)` (It is checked in line 491 by calling !_isEligible()). Due to this mismatch, the transaction might fail.

Also, disqualification may be called on ChefIncentivesController in case relock is disabled or in case chef bounty > 0 (line 180). However, in this case, relock from ChefIncentivesController is not taken into account. Thus, if the user is ineligible due to relock, it will not disqualify him on ChefIncentivesController, reverting the whole transaction due to check on line 491.

In Disqualifier.sol, line 191, eligibility is also checked after locks might be withdrawn, which can affect the result of eligibilityProvider.isEligibleForRewards(). Thus, it should be verified if such functionality is correct, or eligibility should be tracked before the locks are withdrawn.

Recommendation:

Verify the correctness of the current disqualification mechanism and that its implementation corresponds to business logic. Make sure that transaction won't revert due to mismatching validations on Disqualifier.sol and ChefIncentivesController.sol.

Post-audit:

Disqualification mechanism was updated. All logic, connected to disqualifications and bounty is executed in ChefIncentivesController or MFDPlus.

INFO-11 RESOLVED

Unnecessary validation checks and operations in Token Vesting.

TokenVesting.sol: initialize(), claim().
In TokenVesting.sol (line 29), the validation for msg.sender not being zero address is making no sense as zero address cant call any external methods.
In TokenVesting.sol (line 66), the statement if(v.duration == 0) will never become true as zero duration could be set because of validation "duration>0" in the method for adding vesting.

Recommendation:
Remove useless validation and if-statement.

Post-audit:
Contract was removed.

INFO-12 RESOLVED

Functions have the same implementation.

Disqualifier.sol: getBaseBounty() and getCompoundBounty().
Implementation of functions getBaseBounty and getCompoundBounty is absolutely identical. Thus only one function can be left.

Recommendation:
Consider removing or modifying one of the functions.

Post-audit:
Contract was removed.

ETH value is compared to USD value.

EligibilityDataProvider.sol: isEligibleForRewards(), line 205.
The function validates that the user's locked value >= user's required value (with functions lockedUsdValue() and requiredUsdValue() respectively).
The name of the function lockedUsdValue() implies, that it will return value in USD, the natspec section (line 152) states that value is returned in ETH. In fact, the value is returned in USD. Thus the variable `lockedValue` is in USD units.
The name of the function requiredUsdValue() implies, that it will return value in USD, the natspec section (lines 167-168) states that value is returned in ETH. In fact, the value is returned in ETH. Thus the variable `requiredValue` is in ETH units.
Thus when `lockedValue` and `requiredValue` are compared to each other, ETH value is compared to USD value, which is not a correct comparison. That leads to incorrect determining if user is eligible for rewards. Issue is marked as info, as it should be verified by the Radiant team if such functionality is valid.

Recommendation:
Correct the natspec or names of functions lockedUsdValue() and requiredUsdValue(). It should be clear in which units the value is returned. Verify the correctness of comparison or bring values to the same units.

Post-audit:
In function requiredUsdValue(), variable `totalCollateralETH`, taken from LendingPool, was renamed into `totalCollateralUSD`. However no changes were applied in LendingPool, thus LendingPool still returns value in ETH.

Post-audit:
According to the Radiant Capital team, when AaveOracle is deployed, its base currency will be set as USD. Thus functions such as getUserAccountData() in LendingPool will return values in USD, not ETH.

Verification of oracles' implementation.
UniV3TwapOracle.sol: getPrecisePrice(), lines 73, 79.
In case, value of `decimals1` or `decimals0` is greater than 18, an overflow will occur. Though such a case where a token has more than 18 decimals is rare, such functionality should still be verified.
1. UniV2TwapOracle.sol: consult(), line 118.
Price is requested for a value `1 ether`, thus in case token has any other number of decimals but 18, consult() will return an invalid price.
2. BaseOracle.sol
It should be noted that the owner of the contract can set any fallbackOracle at any time, thus affecting the price returned by oracle. Though it is not a security issue, it still should be noted in the final report.

Recommendation:
Correct the natspec or the name of functions lockedUsdValue() and requiredUsdValue() so that it is clear in which units the value is returned. Verify the correctness of comparison or bring values to the same units.

Post-audit:
In function requiredUsdValue(), variable `totalCollateralETH`, taken from LendingPool, was renamed into `totalCollateralUSD`. However no changes were applied in LendingPool, thus LendingPool still returns value in ETH.

Post-audit:
According to the Radiant Capital team, when AaveOracle is deployed, its base currency will be set as USD. Thus functions such as getUserAccountData() in LendingPool will return values in USD, not ETH.

INFO-15 RESOLVED

Event is never used.

MiddleFeeDistribution.sol: MintersUpdated()
Mint method was removed from the contract but the corresponding event remains present.

Recommendation:
Remove unused event.

Post-audit:
Radiant team has removed unused event.

INFO-16 RESOLVED

Duplicating Event.

RadiantOFT: mint()
Transfer event is duplicated since super._mint() uses same event.

Recommendation:
Remove duplicated event.

Post-audit:
Radiant team has removed duplicated event.

INFO-17 | UNRESOLVED

Unnecessary variable.

RadiantOFT: _mint()
Since contract can use _mint once per lifespan maxMintAmount is replacing maxSupply.

Recommendation:
maxMintAmount should be removed and replaced with maxSupply instead, also requires in this function cannot be reverted in the current contract version.

INFO-18 RESOLVED

Variable is initialized in storage in upgradable smart contract.

UniV3TwapOracle.sol: `lookbackSecs`.
Storage variables can't be initialized outside of functions in upgradable smart contracts. Thus variable `lookbackSecs` will be initially equal to `0`. Issue is marked as info, since smart contract has additional setter for this variable.

Recommendation:
Initialize variable in initializer() function.

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

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

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

	contracts\zap\helpers\UniswapPoolHelper.sol **contracts\zap\LockZap.sol** **contracts\protocol\lendingpool\LendingPool.sol** **contracts\protocol\tokenization\AToken.sol** **contracts\protocol\tokenization\IncentivizedERC20.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Radiant Capital in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.
The tests were based on the functionality of the code, as well as a review of the Radiant Capital contract requirements for details about issuance amounts and how the system handles these.

AutoCompounder
- ✓ Should swap rewards to LP for user and stake in LP Mfd
- ✓ Should revert if called not by provided user or Multi Fee Distribution
- ✓ Should not swap if there is no rewards for user
- ✓ Should swap WETH rewards without exchanging them to base token
- ✓ Should revert if reward base tokens or setting routes called not by owner

Disqualifier

Disqualifier
- ✓ getBaseBounty when base bounty is greater than max bounty
- ✓ getBaseBounty when base bounty is lower than max bounty
- ✓ processUsersWithBounty

setters
- ✓ setChefIncentivesController
- ✓ setDisabled
- ✓ setBaseBountyUsdTarget
- ✓ setHunterShare
- ✓ setTreasury
- ✓ setMaxBaseBounty
- ✓ setBountyBooster

EligibilityDataProvider

EligibilityDataProvider
- ✓ setChefIncentivesController
- ✓ setDisqualifierAddress
- ✓ setLPToken
- ✓ lp token cannot be set second time
- ✓ setRequiredEthRatio
- ✓ lockedUsdValue
- ✓ requiredUsdValue
- ✓ isEligibleForRewards
- ✓ setDqTime can be called only by diquvalifier
- ✓ setDqTime
- ✓ lastEligibleTime

PriceProvider

✓ getLpTokenAddress
✓ getTokenPrice the last requsted
✓ getTokenPrice the newest
✓ getLpTokenPrice the last requsted
✓ getLpTokenPrice the newest
✓ getTokenPriceUsd
✓ getLpTokenPriceUsd
✓ only owner can setRewardEligibilityProvider
✓ setRewardEligibilityProvider and update token prices
- not updating prices twice in the same block

Leverager

 # Constructor

✓ Should revert if lending pool is zero address
✓ Should revert if reward eligible provider is zero address
✓ Should revert if aave oracle is zero address
✓ Should revert if lock zap is zero address
✓ Should revert if weth is zero address
✓ Should revert if treasury is zero address

 # Setters

✓ Should set fee percent
✓ Should not set invalid fee percent
✓ Should set treasury

Zaps

✓ Should deposit tokens to lending pool and zap weth (177ms)
✓ Should deposit native to lending pool and zap weth (159ms)
✓ Should estimate zap
✓ Should revert loop if borrow ratio is invalid

RadiantOFT

✓ Should set new exchange rate
✓ Should exchange tokens
✓ Should withdraw tokens

RadiantOFT

Setters

✓ Should set mint complete
✓ Should set new minter
✓ Should revert new minter set second if one already exist
✓ Should set new fee info
✓ Should set new price provider
✓ Should pause bridge
✓ Should unpause bridge (41ms)

Send

✓ Should send from other user (51ms)
✓ Should estimate fee
✓ Should estimate fee with price provider set (2968ms)

Mint and burn

✓ Should burn tokens
✓ Should mint new tokens
✓ Should revert minting new tokens if it excess max mint supply
✓ Should revert minting new tokens if it excess max total supply (73ms)

Modifier

✓ Should revert call owner functions from third party user

StargateBorrow

Token borrowing

✓ Should borrow tokens (3242ms)
✓ Should borrow native (417ms)
✓ Should set new fee

LockZap

Zaps

✓ Should zap native (726ms)
✓ Should zap wrapped native without borrow (297ms)
✓ Should revert if slippage is too high (102ms)
✓ Should zap wrapped native without borrow and price provider (943ms)
✓ Should zap wrapped native with borrow (514ms)
✓ Should zap from vesting without borrow (56ms)
✓ Should zap from vesting with borrow (70ms)
✓ Should revert if available borrow amount is not enough

Utilities and helpers

✓ Should return debt token address
✓ Should set new pool helper
✓ Should set new liquidity zapper
✓ Should perform swap (42ms)
✓ Should add liquidity (408ms)
✓ Should return lp quote
✓ Should return quote amount for adding liquidity

Revert tests

✓ Should revert if initialize second time
✓ Should revert calls for onlyOwner functions from third party user
✓ Should revert if send not enough eth while zap from vesting

Balancer helper

✓ Should return price
✓ Should zap tokens (2777ms)
✓ Should zap weth (114ms)
✓ Should calculate quote

ChefIncentivesController

initialization and setters

1) # should revert when initializing with the wrong parameters
✓ # should revert when initializing twice
✓ # setDisqualifier
✓ should revert when not owner calls setDisqualifier

✓ # start
✓ should revert when calling start() for the second time
✓ should revert when not owner calls start
✓ # addPool
✓ should revert when not poolConfigurator tries to call addPool
✓ should revert when adding pool with existing pool-token
✓ # setRewardsPerSecond
✓ should revert when not owner calls setRewardsPerSecond
✓ # setEmissionSchedule
✓ # setEmissionSchedule (before start)
✓ should revert when not owner calls setEmissionSchedule
✓ should revert when calling setEmissionSchedule with length mismatch (50ms)
✓ should revert when calling setEmissionSchedule with length=0
✓ should revert when calling setEmissionSchedule with timeOffset = 0 (44ms)
✓ # batchUpdateAllocPoint (104ms)
✓ should revert when not owner calls batchUpdateAllocPoint
✓ # setOnwardIncentives (45ms)
✓ should revert when setting onwardInventive on unexisting pool-token
✓ should revert when not owner calls setOnwardIncentives
✓ should revert when not owner calls batchUpdateAllocPoint
✓ should revert when calling batchUpdateAllocPoint with params length mismatch
✓ should revert when calling batchUpdateAllocPoint with not existing pool
✓ # poolLength
✓ # registerRewardDeposit
✓ # setEligibilityEnabled
✓ should revert when not owner calls setEligibilityEnabled
✓ # mint (on MultiFee) (91ms)
✓ # pendingRewards (136ms)
✓ # claim (210ms)
✓ # claim (Eligibility Enabled, user is not eligible) (153ms)
✓ # claim (Eligibility Disabled, user is eligible) (124ms)
✓ # claimBounty (user is ineligible, marketDq, relock set to false) (240ms)
✓ # should not claim bounty when user is marketDq (327ms)
✓ # should not claim bounty when user autoRelockDisabled set to false (260ms)
✓ # should revert when not bounty manager calls claimBounty (208ms)
✓ # claimAll (203ms)
✓ # claim w/0 previously registering deposit (139ms)
✓ # claim (user not eligible for rewards) (152ms)
✓ # claim (not existing pool) (128ms)
✓ should revert when user calls handleActionAfter (72ms)
✓ # disqualifyUser (Eligibility Enabled, user inEligibleForRewards) (164ms)
✓ # disqualifyUser (Eligibility Enabled, user inEligibleForRewards) (164ms)

✓ should revert when disqualifyUser (Eligibility Disabled, user EligibleForRewards) (1229ms)
✓ should revert when disqualifyUser (Eligibility Disabled, user not EligibleForRewards) (324ms)
✓ should revert when not disqualifier calls disqualifyUser (363ms)
2) # claim (ELIGIBILITY_ENABLED = false)
✓ # earnedSince (Eligibility disabled) (78ms)
✓ # saveUserRewards (117ms)
✓ # recoverERC20
✓ should revert when not owner tries to recoverERC20

MFDstats

✓ # should revert when initializing with the wrong parameters (41ms)
✓ # should revert when initializing twice
✓ # addTransfer (treasury opExpenses as zero address, opExpensesRation = 0)
✓ # addTransfer (treasury opExpenses was set, opExpensesRation = 0)
✓ # addTransfer (treasury opExpenses as zero address, opExpensesRation > 0)
✓ # addTransfer (treasury opExpenses was set, opExpensesRation > 0) (50ms)
✓ # addTransfer (second transfer, treasury opExpenses was set, opExpensesRation > 0) (49ms)
✓ # addTransfer (second transfer 2 days after the first, treasury opExpenses was set, opExpensesRation > 0) (46ms)
✓ # getPriceDecimal
✓ # getTotal (before adding transfer)
✓ # getTotal (after adding transfer)
✓ # getLastDayTotal (after adding transfer)
✓ # getLastDayTotal (after 2 days w/o adding transfer)

MiddleFeeDistribution

initialization and setters

✓ # should revert when initializing with the wrong parameters
✓ # should revert when initializing twice
✓ should revert when not owner tries to call setMinters
✓ # setMinters
✓ should revert when setting minters for the second time (setMinters)
✓ # setLpLockingRewardRatio
✓ should revert when not owner tries to call setLpLockingRewardRatio
✓ # setLPFeeDistribution
✓ should revert when not owner tries to call setLPFeeDistribution
✓ # setOperationExpenses
✓ should revert when not owner or admin tries to call setOperationExpenses
✓ # addReward
✓ should revert when not owner or admin tries to call addReward
✓ # getMFDstatsAddress
✓ # getRdntTokenAddress
✓ # getLPFeeDistributionAddress
✓ # getMultiFeeDistributionAddress
✓ # lockedBalances

mint & forwardReward

✓ should revert when not minter tries to mint
✓ # forwardReward (operationExpenses ON (ratio>0), lpReward > 0, rdntReward > 0) (69ms)
✓ # forwardReward (operationExpenses ON (ratio=0), lpReward > 0, rdntReward > 0) (54ms)
✓ # forwardReward (operationExpenses OFF (ratio=0), lpReward > 0, rdntReward > 0) (66ms)
✓ # forwardReward (operationExpenses ON (ratio>0), lpReward = 0, rdntReward > 0) (65ms)
✓ # forwardReward (operationExpenses ON (ratio>0), lpReward > 0, rdntReward = 0) (63ms)
✓ should revert when not multiFee or lpFee tries to call forwardReward
✓ # recoverERC20
✓ should revert when not owner tries to recoverERC20

MultiFeeDistribution

initialization & setters

✓ # setIncentivesController
✓ # setLockTypeInfo
✓ should revert when not owner tries to set minters
✓ # setMinters
✓ should revert when setting minters for the second time
✓ # mint (notifyReward)
✓ should revert when not minter tries to mint
✓ # setIncentivesController
✓ # setMiddleFeeDistribution
✓ # setDisqualifier
✓ # setDAOTreasury
✓ # setRewardConverter
✓ # setLPToken (60ms)
✓ # setBURN
✓ # setRelock
✓ # setLockZap
✓ # setDefaultRelockTypeIndex
✓ # addReward

stake & earn & withdraw (standart)

✓ # stake (caller: user1) (49ms)
✓ should revert when stake under wrong conditions (caller: lockZap) (162ms)
✓ # stake (caller: lockZap) (54ms)
✓ # stake (autoRelock enabled) (61ms)
✓ user should be able to stake for the second time (51ms)
✓ should revert when staking 0 tokens
✓ should revert when staking with unexisting type index
✓ # totalBalance (when stakingToken != rdntToken) (45ms)

✓ # totalBalance (when stakingToken == rdntToken)
✓ # lockersCount (44ms)
✓ # claimableRewards (81ms)
✓ # earnedBalances (before mint) (101ms)
✓ # earnedBalances (before unlockTime) (117ms)
✓ # earnedBalances (after unlockTime) (105ms)
✓ # zapVestingToLp (before unlockTime) (41ms)
✓ # zapVestingToLp (after unlockTime)
✓ should revert when not lockZap calls zapVestingToLp
✓ # lockedBalances (before unlockTime) (73ms)
✓ # lockedBalances (after unlockTime, single staking) (49ms)
✓ # lockedBalances (before unlockTime, multiple stakings) (168ms)
✓ # bountyForUser (before unlockTime) (96ms)
✓ # bountyForUser (after unlockTime) (92ms)
✓ # withdraw w/ BURN=0 (106ms)
✓ # withdraw (minted w/ penalty, earnings time passed) (103ms)
✓ # withdraw (minted w/ penalty, no time passed) (136ms)
✓ # withdraw (minted(penalty=true) more than deposited) (117ms)
✓ should revert when user tries to withdraw more tokens than unlocked (114ms)
✓ # relock (120ms)
3) # withdrawExpiredLocks
✓ # withdrawExpiredLocksWithoutRelockFor (85ms)
✓ # withdrawExpiredLocksWithoutRelockFor (before unlockTime) (90ms)
✓ userLocks array should not contain empty elements (184ms)
✓ userEarnings array should not contain empty elements (475ms)
✓ # withdrawableBalance (when nothing is earned) (327ms)
✓ # withdrawableBalance (after mint, no time passed since staking) (187ms)
✓ # withdrawableBalance (after mint, reward time passed) (87ms)
✓ # getUsers (77ms)
✓ # claimFromConverter (reward > 0) (109ms)
✓ should revert when not rewardConverter calls claimFromConverter (108ms)
4) # individualEarlyExit (claimedUnlockTime > block.timestamp, claimedUnlockTime != systemUnlockTime)
✓ # individualEarlyExit (claimedUnlockTime > block.timestamp, claimedUnlockTime systemUnlockTime) (67ms)
✓ # individualEarlyExit (for the second time) (87ms)
✓ # individualEarlyExit (claimedUnlockTime < block.timestamp) (79ms)
✓ # individualEarlyExit (claimedUnlockTime < block.timestamp) (82ms)
✓ # exit (w/o claiming rewards) (126ms)
✓ # exit (w/ claiming rewards) (119ms)
✓ # getReward (102ms)
✓ # recoverERC20
✓ # disqualifyUser (no penalty) (123ms)

✓ should revert when not disqualifier calls disqualifyUser (88ms)
✓ # disqualifyUser (w/ penalty) (150ms)
✓ # getMFDstatsAddress
✓ # lockTypesLength

Wrong conditions

✓ should revert when initializing with wrong parameters (rdntToken) (625ms)
✓ should revert when initializing with wrong parameters (lockZap) (2578ms)
✓ should revert when initializing with wrong parameters (DAO Treasury) (110ms)
✓ should revert when initializing with wrong parameters (lockerList)
✓ should revert when initializing with wrong parameters (rewardDuration = 0)
✓ should revert when initializing with wrong parameters (rewardsLookback = 0) (63ms)
✓ should revert when initializing with wrong parameters (lockDuration = 0) (251ms)
✓ should revert when initializing with wrong parameters (BURN > WHOLE)
✓ should revert when initializing with wrong parameters (maxLockWithdrawnPerTxn = 0) (98ms)
✓ should revert when initializing with wrong parameters (rewardsLookback > rewardsDuration)
✓ should revert when initializing for the secons time
✓ should revert when locks and multipliers arrays length mismatch (setLockTypeInfo)
✓ should revert when not owner tries to execute owner-method
✓ should revert when setting LP token when it was already set (setLPToken)
✓ should revert when setting zero address as DAO treasury (setDAOTreasury)
✓ should revert BURN > WHOLE (setBURN)
✓ should revert when setting minters for the second time (setMinters)
✓ should revert when not minter tries to add new reward token (addReward)
✓ should revert when minter tries to add reward token which was already added (addReward)
✓ should revert when user tries to set relock type index which is out of bounds (setDefaultRelockTypeIndex)
7) should revert when owner tries to recover system token (recoverERC20)
✓ # mint (notifyReward)
✓ # mint (amount = 0)
✓ should revert when not minter tries to mint
✓ # should revert when withdrawing 0 tokens (91ms)
✓ should revert when not owner tries to call setIncentivesController
✓ should revert when not owner tries to call setMiddleFeeDistribution
✓ should revert when not owner tries to call setDisqualifier
✓ should revert when not owner tries to call setLPToken
✓ should revert when not owner tries to call setDAOTreasury
✓ should revert when not owner tries to call setBURN
✓ should revert when not owner tries to call setRewardConverter
✓ should revert when not owner tries to call setLockZap
✓ should revert when not owner tries to call recoverERC20
✓ should revert when owner calls recoverERC20 with reward token

TokenVesting

9) should revert when initializing with zero address as rdnt token
✓ should revert when initializing second time
✓ # setRdnt

✓ should revert when not owner sets Rdnt
✓ # addVest
✓ should revert when not owner tries to addVest
✓ should revert when addVest with duration=0
✓ should revert when addVest with amount=0
✓ should revert when adding vesting twice on the same user)
✓ # claimable (duration > 0, elapsedTime < duration)
✓ # claimable (duration > 0, elapsedTime = duration)
✓ # claimable (duration > 0, elapsedTime > duration)
✓ # claim (half of duration passed)
✓ # claim (claimTime = vestFinish)
✓ # claim (claimTime > vestFinish)
✓ # claim (second claim after claiming all amount) (44ms)

Leverager

✓ Should estimate zap as 0 (46ms)
✓ Should revert call deployer functions from third party user
✓ Should deposit tokens to lending pool and zap weth multiple times (232ms)
✓ Should deposit native to lending pool and zap weth multiple times (241ms)
✓ Should do nothing if amount to zap is zero (40ms)
✓ Should zap weth with borrow (94ms)

Migration

Modifiers

✓ Should revert call deployer functions from third party user
✓ Should revert when exchange is paused

RadiantOFT

Send

✓ Should estimate fee when price provider is address(0) (51ms)

Modifier

✓ Should revert call deployer functions from third party user
✓ Should revert when contract is on pause

Reverts

✓ Should revert when using wrong constructor parameters (66ms)
✓ Should revert when send fee is not covered (222ms)
✓ Should revert if new fee is too high

StargateBorrow

✓ Should revert if onlyOwner function is called from foreign uset
✓ Should revert if new XChain fee is not valid

LockZap

Zaps

✓ Should zap weth with lock duration (75ms)

Utilities and helpers

✓ Should return reserves on pair

✓ Should correct decimals
✓ Should return quote amount for adding liquidity
✓ Should return quote amount for adding liquidity

Revert tests

✓ Should revert if liquidity zap is already initialized
✓ Should revert if try to zap 0 eth
✓ Should revert if try to add liquidity with 0 amount
✓ Should revert if try to add liquidity on behalf of 0 address

Balancer helper

✓ Should return lp price (279ms)
✓ Should return valid reserves
✓ Should revert if pool is already initialized
✓ Should revert if contract is already initialized

UniV2TwapOracle

✓ Should check if oracle can be updated
✓ Should update price correctly (47ms)
✓ Test TWAP when price was manipulated at some point (855ms)
✓ Should not let update if period not passed yet
✓ Should not let consult if price is stale
✓ Should return stale price if stale price is allowed

UniV3TwapOracle

✓ Should return price in token0
✓ Should return price in token1

[Scenario] User borrows USDC tokens in Ethereum and transfer them to Polygon using Stargate

 # Token borrowing

✓ Borrows USDC tokens from Ethereum to Polygon (20268ms)

BountyManager

 # Initialization

✓ Initializes (969ms)
✓ Reverts on re-initialization (633ms)

 # Actions

 # Quote

✓ Quotes
✓ Quotes if the hunter share is zero
✓ Quotes when issues the basic bounty
✓ Reverts when quoting if an incorrect fee distributor which writes to the storage (39ms)

 # Claim

✓ Claims (74ms)
✓ Reverts when claiming if an expected bounty is zero (41ms)
✓ Reverts when claiming if too much slippage because of an expected bounty is exaggerated (55ms)
✓ Reverts when claiming if too much slippage because of zero total bounty (69ms)
✓ Claims when issuing a basic bounty if an enough reserve of Radiant tokens (55ms)
✓ Reverts when claiming if issues a basic bounty and the reserve of Radiant Reverts when claiming if issu tokens is empty (42ms)

```
✓ Claims when issuing a basic bounty with a zero total bounty (57ms)
✓ Claims a chef bounty (63ms)
✓ Claims a compound bounty (68ms)
# Bounty running
✓ Runs a bounty with execution (55ms)
✓ Runs a bounty without execution
# Getting of the basic bounty
✓ Gets
✓ Gets the maximum basic bounty if the calculated bounty is greater than
# Setting of parameters
✓ Sets the basic bounty USD target
✓ Prevents non-owners from setting the basic bounty USD target
✓ Sets the hunter share
- Reverts when setting the hunter share that is greater than 100%
✓ Prevents non-owners from setting the hunter share
✓ Sets the maximum basic bounty
✓ Prevents non-owners from setting the maximum basic bounty
✓ Sets the bounty booster
✓ Prevents non-owners from setting the bounty booster
✓ Sets the slippage limit
✓ Prevents non-owners from setting the slippage limit
# Transfer of ERC20 tokens to the owner
✓ Transfers
✓ Prevents non-owners from transfer
```

Zokyo Security team has prepared a set of unit tests to verify the correctness of work of all the contracts within the scope. Additionally, auditors have prepared a fork test to verify the correctness of interaction with 3rd part protocols such as Stargate, LayerZero, Balancer and Uniswap V2.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by the Radiant Capital team

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

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

Non-Elig CIC

- ✓ setOnwardIncentives
- ✓ can't start again
- ✓ registerRewardDeposit (515ms)
- ✓ recover ERC20 (39ms)

addPool requires

- ✓ should be callable by only pool configurator (98ms)

saveUserRewards

- ✓ zero address

handleActionAfter

- ✓ can't called from non-pool

batchUpdateAllocPoint

- ✓ length must match
- ✓ non existing pool (332ms)

setRewardsPerSecond callable

- ✓ should be callable by only owner (535ms)

check persist rewards per second in a single pool

- ✓ Deposit by User 1 (1374ms)
- ✓ claimable rewards should be calculated in seconds (41ms)
- ✓ claimable rewards should be updated with rewards per seconds (282ms)
- ✓ claim for middlefeedistribution (491ms)
- ✓ should claim rewards (972ms)
- ✓ withdraw all (115ms)

ChefIncentivesController Rewards Schedule and Manual Setting RPS.

- ✓ setEmissionSchedule before start (84ms)
- ✓ manually set rewards (260ms)
- ✓ scheulded rewards (410ms)

Ensure Emissions consistant

- ✓ user1 emission rate unchanged after user2 deposits (5689ms)

Refillable CIC
- Exhaust w/ 2 users & Refill

Default Relock
✓ relock (27639ms)

ChefIncentivesController Rewards Schedule and Manual Setting RPS.
✓ setEmissionSchedule before start (78ms)
✓ manually set rewards (269ms)
✓ scheulded rewards (2256ms)

Ensure Users can borrow after Eligibility change (no sub revert)
✓ Deposit and Lock by User 2 (1874ms)
✓ Wait for lock expire, user 2 borrow, ensure no revert (1545ms)

Base Bounty
✓ base bounty (2907ms)
✓ can scale up when price down (1555ms)
✓ can be boosted (1857ms)

Require Locked Value
✓ Deposit and borrow by User 2 + 3 (853ms)
✓ Lock new tokens, locked eth values are correct (1213ms)
✓ Earns RDNT on Lend/Borrow (78ms)
✓ Unlock all, earnings go to zero; but prev emissions are kept (1369ms)
✓ Lock again, locked eth values are correct (2558ms)
✓ Earns RDNT on Lend/Borrow (251ms)
✓ User2 can Vest RDNT (646ms)
✓ Can exit and get RDNT (360ms)

Interest split 70/30 between lockers and depositors
✓ lockers get 70% of interest (1227ms)

Reserve Factor
✓ 50% RV test (377ms)
✓ Can change to 70% (344ms)

Ensure lockers can claim Platform Revenue
✓ Lock LP (1481ms)
✓ Deposit USDC (473ms)
✓ Can borrow and claim (2766ms)
✓ Can borrow and claim many times (69241ms)

Compounding
✓ autocompound (28902ms)

MFDs split Platform Revenue
✓ Deposit and borrow by User 2 + 3 (1976ms)
✓ Earns RDNT on Lend/Borrow
✓ User2 can Vest RDNT (409ms)
✓ Both receives platform fees, also opEx receives interest (753ms)
✓ Can exit and get RDNT (180ms)

Individual Early Exits
✓ Check Individual Early Exit. (2063ms)
MultiFeeDistribution
✓ getMFDstatsAddress
✓ mintersArtSet
✓ setLPToken
✓ setDAOTreasury
✓ setBURN
✓ setDefaultRelockTypeIndex
✓ setLockTypeInfo
✓ addReward
✓ Add some radiant rewards
✓ recover ERC20 (53ms)
✓ mint & stake vlidation
✓ Withdraw expired locks (84ms)
✓ Different lock periods (194ms)
✓ Different reward amount per lock lengths (87ms)
✓ relock expired locks (131ms)
✓ autorelock (110ms)
- exit; validation
✓ exit; with penalty (141ms)
✓ Remove exit penalties; exit; withdraw full earnings (79ms)
✓ withdraw; empty earnings (154ms)
✓ Remove exit penalties; withdraw from unlocked (82ms)
✓ Remove exit penalties; Insufficient unlocked balance (91ms)
✓ Remove exit penalties; Insufficient balance (116ms)
✓ Remove exit penalties; with penalty (160ms)
✓ Remove exit penalties; withdraw (80ms)
✓ Vesting RDNT stop receiving rewards (72ms)
✓ Linear exit; day 1 (69ms)
✓ Linear exit; day 30 (67ms)
✓ Linear exit; last day (51ms)
✓ Linear exit; withdraw; day 1 (57ms)
✓ Linear exit; day 30 (56ms)
✓ Linear exit; last day (45ms)
- Individual early exit; validation
✓ Individual early exit; with penalty (152ms)
✓ Individual early exit; zero amount (74ms)
✓ cleanExpiredLocksAndEarnings; it should work fine (250ms)
✓ earnedBalances
✓ getReward; unknown token (74ms)
✓ getReward; notify after notify (194ms)

✓ different staking token and rdntToken (218ms)

MiddleFeeDistribution

✓ setMinters
✓ setLpLockingRewardRatio
✓ setLPFeeDistribution
✓ lockedBalances
✓ forwardReward with zero expense
✓ recover ERC20
✓ validation
✓ tokens minted on Middle go to MFDs
✓ tokens minted on Middle with custom lp ratio
✓ tokens minted on Middle go to user (65ms)

MFD Relocking

✓ Withdraw Expired Locks; disabling auto relock at first is saved (1583ms)
✓ Relock happens automatically at Withdraw (1913ms)
✓ Auto Relock doesn't happen when disabled (1679ms)
✓ Relock Expired Locks (2205ms)

Check MFD Stats via Deposit/Borrow Cycles

✓ Last Day total when empty
✓ Lock LP and Check Locked Supply (1427ms)
✓ Check Last Day Total (2298ms)
✓ Check Total Interest with Multiple Days (2356ms)
✓ Check Total Interest Per Asset Breakdown (5557ms)
✓ addTransfer with time advance (6742ms)
✓ addTransfer with zero expense (676ms)

Autocompound

✓ should add additional lock (24275ms)

Looping/Leverager

✓ autoZap test with slippage

Deposit/AutoZap

✓ autoZap test while deposit

Looping/Leverager

✓ returns debt token w/ getVDebtToken
✓ borrows & deposits correct amount, w/ correct fee
✓ autoZap while looping

Upgradeable Contracts

✓ Upgradeable ChefIncentivesController works.
✓ Upgradeable MiddleFeeDistribution works.
✓ Upgradeable MFD works.
✓ Upgradeable LPFD works.
✓ Upgradeable PriceProvider works.

Zapper
✓ setLiquidityZap
✓ setPoolHelper
✓ can zap into locked lp
✓ can zap from Vesting
✓ can zap WETH, and from Borrow
✓ can zap from Vesting w/ Borrow
✓ can early exit after zapping vesting w/ borrow
LiquidityZap
✓ initLiquidityZap again fails
✓ fallback
✓ zapEth validation
✓ zapEth validation
Price Provider
✓ returns initial token price (USD)
✓ returns initial token price (ETH)
✓ returns LP token price
✓ returns LP token price USD
Uni V2 TWAP
✓ can be updated
✓ returns price
✓ LP token change reflected in price after update
Uni V3 TWAP
✓ can be updated
✓ returns price
✓ LP token change reflected in price after update
Migration V1 → V2
✓ setExchangeRate
✓ Prepare reserve
✓ Migrate tokens
✓ withdrawTokens
✓ Pause action
Radiant token:
✓ mint fails for regular user
✓ can be burned
Radiant OFT:
✓ mint fails if exceeds max supply
✓ sendFrom()
✓ bridge()
✓ bridge fails if exceeds max supply
✓ pauseBridge()
✓ pauseBridge() - reverts if not owner

Stargate Borrow
✓ setDAOTreasury
✓ setXChainBorrowFeePercent
✓ Check X Chain Borrow Fee
✓ borrow eth
✓ invalid treasury
LiquidityZap
✓ initLiquidityZap again fails
✓ fallback
✓ zapEth validation
✓ zapEth validation
Price Provider
✓ returns initial token price (USD)
✓ returns initial token price (ETH)
✓ returns LP token price
✓ returns LP token price USD
Uni V2 TWAP
✓ can be updated
✓ returns price
✓ LP token change reflected in price after update
Uni V3 TWAP
✓ can be updated
✓ returns price
✓ LP token change reflected in price after update
Migration V1 → V2
✓ setExchangeRate
✓ Prepare reserve
✓ Migrate tokens
✓ withdrawTokens
✓ Pause action
Radiant token:
✓ mint fails for regular user
✓ can be burned
Radiant OFT:
✓ mint fails if exceeds max supply
✓ sendFrom()
✓ bridge()
✓ bridge fails if exceeds max supply
✓ pauseBridge()
✓ pauseBridge() - reverts if not owner

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

zokyo