Radiant V2 Audit

OPENZEPPELIN SECURITY | OCTOBER 18, 2023

Security Audits

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2023-04-24
: To 2023-06-09

Languages
: Solidity

Total Issues
: 82 (75 resolved)

Critical Severity Issues
: 1 (1 resolved)

High Severity Issues
: 5 (5 resolved)

Medium Severity Issues
: 15 (13 resolved)

Low Severity Issues
: 41 (36 resolved)

Notes & Additional Information
: 20 (20 resolved)

Scope

In scope were the following contracts:

```
./contracts/radiant
├── accessories
│   ├── Compounder.sol
│   ├── Leverager.sol
│   └── StargateBorrow.sol
├── eligibility
│   ├── BountyManager.sol
│   └── EligibilityDataProvider.sol
├── libraries
│   ├── AddressPagination.sol
│   └── LockerList.sol
├── oracles
│   ├── BaseOracle.sol
│   ├── ChainlinkV3Adapter.sol
│   ├── PriceProvider.sol
│   ├── UniV2TwapOracle.sol
│   ├── UniV3TwapOracle.sol
│   └── wstethOracle.sol
├── staking
│   ├── ChefIncentivesController.sol
│   ├── MiddleFeeDistribution.sol
│   └── MultiFeeDistribution.sol
├── token
│   ├── Migration.sol
│   └── RadiantOFT.sol
└── zap
    ├── LockZap.sol
    └── helpers
        ├── BalancerPoolHelper.sol
        ├── DustRefunder.sol
        ├── LiquidityZap.sol
        └── UniswapPoolHelper.sol
```

collateral and borrow against it while keeping a healthy collateralization ratio. It features cross-chain borrowing by leveraging LayerZero's novel interoperability protocol between different blockchains.

Protocol revenue is shared with depositors and additionally, RDNT token rewards are distributed to eligible stakers. Stakers can become eligible by providing liquidity in the form of RDNT and the wrapped native token of each specific blockchain, and then locking it for a given period of time. The longer the lock time, the larger the reward multiplier. This incentivizes protocol participants to provide deep liquidity for the governance token.

The audited codebase is comprised of a few different packages that manage staking, incentives, eligibility, price oracles, a bounty system, looping (leveraging positions in a single transaction), zapping assets into locked LP tokens, cross-chain borrowing and the new Radiant OFT V2 token.

The system is currently deployed on Arbitrum and Binance Smart Chain (BSC). Locked liquidity is provided in a Balancer weighted pool (80% RDNT and 20% WETH) in Arbitrum, while a PancakeSwap V2 pool is used on BSC with 50% RDNT and 50% WBNB.

Staking, Reward Emissions and Eligibility

Staking on Radiant V2 means providing liquidity and locking the LP tokens for a certain period of time. In order to become eligible for reward emissions, users must always maintain a certain amount of locked LP tokens, respective to their total deposited collateral value in USD terms. These tokens can be locked for different periods of time, ranging from 1 month up to 12 months at the moment of writing. The longer the period, the larger the multiplier applied to accrued rewards.

By default, auto-relocking is enabled, which means that after a certain lock expires, it will get locked again automatically for the preferred lock period in order to not lose eligibility for rewards. Alternatively, the LP tokens can be withdrawn on lock expiration.

RDNT reward emissions can be claimed into vesting, which imposes a 90-day period before users can withdraw their full earnings. If a user decides to withdraw their rewards before vesting is complete, they will incur a linear penalty over that time ranging from 90% down to 25%. The

Eligible depositors and borrowers are entitled to receive RDNT emissions within the money market. Emissions can follow either a linear pattern with a fixed number of tokens distributed per second, or be linked to a predefined schedule that continuously updates the distribution at specific times.

The aforementioned vesting period can be bypassed and rewards instantly claimed for their total amount under the condition that they are zapped into locked LP tokens by pairing all claimed RDNT with the wrapped native token of the specific chain.

Compounding

Eligible depositors will generate rewards in `rTokens` such as rUSDC or rWBTC, which are the deposited, yield-generating equivalent of their underlying asset. These rewards can be claimed directly as `rTokens` into the user's wallet or, alternatively, they can be compounded into locked LP tokens in order to generate even more rewards.

This can be achieved via the `Compounder` contract, in a single transaction. Users can self-compound their rewards at any time or, they can opt-in for auto-compounding and rely on bounty hunters (see below) to do so on their behalf once a day in exchange for a small fee taken out of their claimed rewards.

Bounty System

In order to keep the protocol both decentralized and working as expected, a system of bounties has been set up so that third parties (called hunters) are incentivized to perform specific actions on behalf of the protocol. There are three different actions that hunters can perform in order to receive a bounty for their efforts:

1. **Withdrawing expired locks for users:** Expired locks should not keep generating rewards, so hunters are paid a fixed amount (the base bounty) in order to timely withdraw any user's expired locks. Depending on user settings, this lock will be withdrawn as LP tokens directly into their wallet or relocked into the protocol for their preferred amount of time.
2. **Stopping emissions for ineligible users:** Due to natural market fluctuations, user locks might change in USD value and go below the necessary threshold to maintain eligibility.

- **Auto-compounding user rewards**: Users can opt-in for auto-compounding their rewards into locked LP tokens. These rewards correspond to their share in protocol revenue and are paid in the different `rTokens` supported by the platform. Hunters can claim these rewards every 24 hours for opted-in users to get them all swapped into LP tokens, which are automatically locked. For this last bounty type, the total amount paid is not fixed but instead, it represents a percentage of the total amount of rewards claimed on behalf of the user. A percentage of this bounty is paid to the hunter and the remaining is kept by the bounty reserve in order to have enough tokens to pay the other two types of bounties.

All bounty types are paid in RDNT tokens, which are subject to the same 90-day vesting discussed before. Additionally, hunters need to be eligible as well in order to be able to claim bounties.

Radiant V2 OFT

OFT stands for Omnichain Fungible Token. This new version of the Radiant governance token (RDNT) implements the OFT interface provided by LayerZero in order to make it natively transferrable cross-chain, and it is not upgradeable. Old v1 token holders can migrate their holdings into the new version by leveraging the `Migration` contract.

The token can be transferred to and from any of the chains where the protocol is deployed (BSC and Arbitrum at the time of writing). In order to do so, users need to pay two fees: the LayerZero fee and the Radiant bridging fee. Both are paid in the chain's native currency when executing the transfer transaction.

Cross-Chain Borrowing

Users have the option of borrowing a certain subset of assets and getting them sent directly into another chain. For instance, users can deposit USDC on Arbitrum and borrow USDT on Binance Smart Chain. This can be done via the `StargateBorrow` contract, which communicates with Stargate's bridge routers in order to perform the swaps.

There are also two fees involved, one to cover LayerZero's fee and another one charged by the Radiant protocol and sent to their treasury.

Looping

It features a variety of functions to achieve that goal, depending on the asset being used and whether funds should be pulled from the user's wallet or borrowed against their already existing collateral. A manually leveraged position requires multiple transactions to deposit collateral, borrow against it and then redepositing the borrowed amount. This can all be performed in a single looping transaction as many times as necessary to reach the desired leverage.

Zapping

Providing liquidity in a pool is a complex process which involves swapping source assets into the two necessary tokens in the right proportion. The `ZapLock` contract, along with all the pool helpers, provides the necessary functions to lock LP tokens directly from any source asset users may hold, in a single transaction. It even allows for tokens to be borrowed instead of supplied when necessary.

Price Oracles

Price oracles are used for two different purposes within the protocol:

1) To track RDNT prices in order to manage eligibility and properly provide liquidity in the form of wrapped native token and RDNT in the right proportion.

2) To manage the different money markets supported by the platform.

The audited codebase contains a variety of price oracles focused on the first task, so they are all specifically designed to retrieve RDNT prices. This gives the team enough flexibility to choose the best candidate depending on the blockchain the protocol is deployed, Chainlink price feed availability, market conditions and the automated market maker of choice where users will provide liquidity. The different price oracles available are:

- **Uniswap V2 TWAP Oracle:** This is the oracle currently configured on the BSC deployment. It uses a 60-second period between updates and returns the average price registered by the associated PancakeSwap V2 pool on the last period. It relies on the protocol being poked frequently enough by user actions in order to perform these updates in a timely manner. Note that this period is particularly low, which carries some risk of making the price manipulatable. However, the Radiant team made the conscious decision to set it this way in order to more

stale prices to be reported. There is a feature implemented to avoid using the TWAP price and instead, simply calculate the price based on the current pool reserves. Since this alternative is highly manipulatable, it is encouraged that it is only used during protocol deployment when the pool is being created. The team stated that they will switch to using Chainlink as soon as there is an available price feed for RDNT on BSC.

- **Uniswap V3 TWAP Oracle:** This oracle is not actively being used, but it contains the necessary features to correctly track the TWAP price registered by the associated Uniswap V3 pool.

- **Chainlink V3 Adapter:** This is the oracle of choice on Arbitrum, where there is an available price feed for the RDNT token.

- **Base oracle:** This is not a full oracle but instead it is intended to be inherited by the TWAP oracles in order to enforce a consistent API with Chainlink, which conforms to its associated interface. It provides the basic functionality and configuration needed by their children.

Additionally, there is an independent oracle that tracks wrapped staked ETH (wstETH) prices by querying Chainlink's price feeds. This specific oracle is not used within the audited codebase.

Security Model & Trust Assumptions

The `owner` of the Radiant contracts is the multisig wallet managed by the Radiant team. The protocol `owner` has the power to:

- Pause/unpause certain contracts to restrict new fund inflows
- Upgrade implementation contracts
- Change external contract addresses
- Change key protocol configuration settings such as fees, incentives, and slippage ratios
- Change oracle settings such as allowing/disallowing stale prices and allowing the use of fallback oracles

Users of the protocol place their trust in the Radiant team to act in their best interests when making administrative changes to the system.

maintain an up-to-date protocol state.

The community can participate in governance by submitting proposals to the Radiant DAO. All users that hold locked RDNT tokens in the `MultiFeeDistribution` contract are eligible to vote. Voting weight is proportional to the number of tokens held. The foundation that administers the DAO is trusted to abide by governance decisions.

Update: *After our assessment of the implemented fixes, it is worth noting that the Radiant team now has the ability to remove existing rewards from* `MultiFeeDistribution` *and then recover any existing balance. As previously stated, the team is expected to always act in the best interest of its user base.*

Eligibility Manipulation Leads to Stolen Rewards

Users can leverage their position by calling the `loop` function within the `Leverager` contract. This function will loop through multiple cycles of depositing and borrowing of the same asset in order to achieve the desired leveraged position.

During this process, `ChefIncentivesController`'s `setEligibilityExempt` function is called to temporarily activate an exemption for the user. This skips the check on their eligibility after every iteration in order to save gas. Once the position is fully leveraged, this exemption is set back to `false` on the last loop iteration.

However, a `loopCount` of 0 iterations is allowed, so an attacker may pass `0` as a value for `loopCount` which will skip the loop execution and thus, also skip setting the eligibility exemption back to `false`. Similar logic has been implemented in `loopETH` and `loopETHFromBorrow`, but in those functions, the eligibility exemption is toggled back to `false` outside of the `for` loop, so they are not vulnerable.

This exemption is used in the `handleActionAfter` hook to prevent eligibility updates for `ChefIncentivesController` pools. Since this functionality is crucial for balance change updates used by the `IncentivizedERC20` (transfers, mint, burn) and `StableDebtToken` (mint, burn) contracts it opens up different attack vectors such as:

- Reusing the same deposit across multiple accounts in order to emit additional rewards by transferring the rTokens between multiple accounts and requalifying every account for rewards via `requalifyFor`.
- Inflating the user's pool balance by depositing a flash-loaned asset, requalifying the position for rewards eligibility via `requalify` and finally withdrawing the deposit to pay back the flash loan amount. At this point, the total deposited collateral is zero but the user is still emitting rewards for the entire amount previously deposited.

The only requirement in both scenarios is to have the minimum required LP locked from the attack setup until the closest maturity date. After that, since deposited collateral will be zero, the required amount of LP tokens locked to maintain eligibility will also be zero.

***Update:** Resolved in pull request #170 at commit 5f64342. A new `InvalidLoopCount` custom error has been introduced. It is used to revert the `loop`, `loopETH`, and `loopETHFromBorrow` functions when they are called with a `loopCount` value equal to zero.*

High Severity

Vested RDNT Tokens Can Be Withdrawn Early With No Penalties

The `individualEarlyExit` function within the `MultiFeeDistribution` contract allows withdrawing a specific user earnings entry, which is identified by passing its `unlockTime` to `ieeWithdrawableBalances`. Once a record is matched, its index is assigned and returned along with the penalty and burn amounts. The `individualEarlyExit` function contains a check that presumes that an out-of-bounds `index` returned by `ieeWithdrawableBalances` indicates the target `unlockTime` was not found.

However, if the specified `unlockTime` is not found, `ieeWithdrawableBalances` will return all variables set to zero, including the `index`. Instead of halting execution as intended, it will continue and `individualEarlyExit` will remove the first `userEarnings` item from the list, while not correctly decreasing the user's `earned` balance accordingly. This transaction will not revert and will end up transferring 0 RDNT tokens to the user and the DAO treasury.

A malicious user can call `individualEarlyExit` as many times as necessary using an invalid `unlockTime` until their `userEarnings` array is empty, and then call the `exit` function to determine the withdrawable balance by looping through the `userEarnings` array. Since there are no entries (they were all removed), the penalty and burn amounts returned by `withdrawableBalance` will be zero, resulting in a withdrawable balance equal to the full `balances[user].earned` value.

The end result is that any user can withdraw their full amount of RDNT rewards at any time, bypassing any penalties for exiting early.

To ensure users cannot withdraw earnings that are not fully vested without penalties, consider reverting when the specified `unlockTime` is not found.

RDNT Token Bridge Fee Can Be Bypassed

The RDNT token is implemented as a LayerZero Omnichain Fungible Token (OFT), which allows it to be sent to any blockchain that has a LayerZero endpoint. The tokens can be transferred cross-chain by using the OFT `sendFrom` function, which internally calls the overridden `_send` function in the `RadiantOFT` contract. The purpose of this override is to allow the Radiant team to collect an extra fee called the bridge fee, which is expressed in the chain's native currency and dependent on the amount of RDNT being transferred.

This fee can potentially be bypassed by instead using the OFT `sendAndCall` function, which is similar to `sendFrom` but allows executing a specific payload after executing the token transfer. If crafted correctly, this will execute the inherited `_sendAndCall` function, which has not been overridden to also collect the bridge fee.

Consider overriding the internal `_sendAndCall` function in order to also collect the extra fee and avoid the potential loss of protocol revenue.

Update: *Resolved in pull request #176 at commits 01b2bde and eddbf1c.*

Initial Liquidity Can Be Lost by Front-Running Pool Initialization

Both the `UniswapPoolHelper` and `BalancerPoolHelper` contracts implement an `initializePool` function that is responsible for creating a new liquidity pool for the RDNT token paired with a base token. In the currently deployed contracts, RDNT is paired with WETH on Arbitrum and WBNB on Binance Smart Chain. In the case of Uniswap, this initialization will revert if a pair has already been created.

The logic in both pool initialization functions expects the input RDNT/WETH or RDNT/WBNB tokens to be sent to the respective helper contract in advance, before calling either `initializePool` function. This is outlined in the deployment script for UniswapPoolHelper and BalancerPoolHelper as a manual set of transactions, instead of being executed atomically. After depositing liquidity into a pool, the respective `initializePool` function will send the received liquidity (LP) tokens to the caller (see here and here).

contract, and then front-runs the `initializePool` transaction, or alternatively (on Layer 2 chains that do not feature public mempools), the attacker can back-run the second token transfer to accomplish the same goal. The existing RDNT and WETH or WBNB contract balances will then be used to provide liquidity and send the minted LP tokens to the attacker.

Consider adding the `onlyOwner` modifier to the `initializePool` functions of `UniswapPoolHelper` and `BalancerPoolHelper` to ensure they can only be executed by the protocol owners, and/or implementing an atomic flow of contract deployment, transfer of funds, and pool initialization.

***Update:** Resolved in pull request #177 at commit 7ce4838. The `onlyOwner` modifier was added to both `initializePool` functions.*

Bounty Hunters Can Incur a Denial-of-Service

Successful bounty hunters are expected to claim a large number of bounties per day. The larger the protocol grows, the more situations will arise for hunters to get a reward for ensuring the protocol works as expected.

These bounties are paid out in RDNT tokens, but not directly to the hunter's wallet. Instead, these tokens go into a 90-day vesting period. In order to track penalties for each individual amount in vesting, every bounty needs to be separately added to the hunter's earnings array.

A successful bounty hunter can potentially claim a very large amount of bounties within the same 90-day period (before vesting ends for his first bounty), causing the `userEarnings` array to be so large that any interaction with it on a transaction will fail due to running out of gas because it has reached the block gas limit. When this happens, hunters will not be able to withdraw their earnings or perform static calls to the `earnedBalances` function, essentially bricking protocol interactions for that specific user.

The same effect can be achieved by a malicious actor performing a grieving attack on any account by claiming numerous times on behalf of a victim's address since there is no minimum amount claimable and there is no access control enforced when claiming rewards.

functionality if necessary.

Regarding bounty hunters, in order to prevent them from incurring a denial-of-service, consider implementing an alternate solution to how bounties are added to the `userEarnings` array. For example, the same array item could be used to combine multiple bounty amounts claimed within the same day.

Update: *Resolved in pull request #201 at commits 42c6772, bec761e, 5b43456, 82d6503, d4c1489 and 814adc6. Claims of 0 RDNT are disallowed and a 1-day epoch system has been implemented so that both RDNT claims and LP locks of the same type happening on the same day are merged within 1 entry. LP locks aggregation only takes place if the current lock will unlock on the exact same day as the last lock from the user list. This strategy makes the protocol more gas efficient and protects both users and bounty hunters from denial-of-service attacks by limiting the number of entries in both arrays.*

Incorrect Accounting of `earned` and `unlocked` Tokens on `exit` May Brick the Protocol

The `exit` function within the `MultiFeeDistribution` contract is responsible for withdrawing RDNT tokens (earnings) that are already unlocked and still in vesting. It is using `withdrawableBalance` to calculate the balance that can be withdrawn, and then it releases the tokens to the user via `_withdrawTokens`.

While the function is correctly deleting `userEarnings`, it is not updating the `unlocked` and `earned` balances to `0`, which is where the `withdrawableBalance` function gets the data from.

A regular call to `exit` will break that user's internal accounting for earnings and lockings, leading to reverts for crucial functionalities of the system and thus, causing unexpected results and potentially bricking most of the `MultiFeeDistribution` contract alongside other parts of the protocol.

The following features will be impacted by the incorrect value of earned or unlocked tokens:

- The `earnedBalances` function will be returning anincorrect value.

In order to maintain a correct internal accounting of user balances, it is recommended to reset both the `unlocked` and `earned` balances to `0` in the `exit` function.

Update: *Resolved in pull request #66 at commit 1830760. Both `unlocked` and `earned` values are now reset to `0` within the `exit` function.*

Medium Severity

Incorrect Calculation of Tokens to Burn on `withdraw`

The `withdraw` function of the `MultiFeeDistribution` contract implements the logic for withdrawing earned and unlocked tokens from the protocol. It loops through all earnings to sum the amount of earned tokens, penalties for the early withdrawal, and the burn amounts.

The problem is that `burnAmount` is incorrectly calculated by using the total `penaltyAmount` instead of the individual penalty for the current loop iteration, which leads to an inflated value of `burnAmount`.

It is recommended to correct the calculation of `burnAmount` by replacing `penaltyAmount` with the individual penalty associated with the current loop iteration.

Update: *Resolved in pull request #208 at commit 65220b8.*

Incorrect Earnings Accounting When `ChefIncentivesController` Reserve Is Empty

The `ChefIncentivesController` contract allows claiming rewards through its `claim` function. The rewards are then sent to the `MultiFeeDistribution` contract by calling `_mint`, which uses the `_sendRadiant` function to perform the actual transfer. Following the transfer, the claimed rewards are vested by calling `MultiFeeDistribution`'s `mint` function.

will record RDNT tokens that it never actually received as vested, leading to the insolvency of `MultiFeeDistribution` because the accounting will record user reward amounts in excess of the contract's balance.

To ensure adequate funds are present in the reserve, consider reverting when there are insufficient funds for any rewards claim. In addition, consider using OpenZeppelin Defender to detect this event and subsequently pause the contract or fill the reserve.

Update: *Resolved in pull request #174 at commits 4c7d287 and 482c14f. The `ChefIncentivesController` contract will revert with an `OutOfRewards` custom error if there are not enough reserve funds to pay out a claim. Additionally, the contract is no longer paused when this situation is encountered.*

Incorrect Usage of Uniswap V2 Oracle

Uniswap provides example code for a Uniswap V2 oracle and `UniV2TwapOracle` follows the example implementation to retrieve oracle price data. However, Uniswap's implementation actually relies on the unchecked arithmetic in Solidity `0.6.6` in order to work correctly. In contrast, the Solidity version `0.8.12` used by `UniV2TwapOracle` contains default checks to prevent overflow from occurring. Since the Uniswap implementation expects overflows as part of normal operation, the use of modern Solidity `>=0.8.0` may eventually cause the code to revert unexpectedly.

Consider adding `unchecked` to the parts of the `UniV2TwapOracle` code that are expected to overflow.

Update: *Resolved in pull request #204 at commit 6e67d1c. All parts that are expected to overflow have been wrapped in `unchecked` blocks.*

Incorrect Leveraged Borrow Calculation

The `Leverager` contract provides the ability to leverage a user's position by "looping" the deposit and borrow actions, i.e. funds that are borrowed are then deposited, increasing the

Users are charged a percentage fee on every deposit during the leveraging process but the fee is not correctly deducted from the `amount` used for the next deposit. This can be observed in the loop where the amount to deposit is reduced by the `fee` but `amount` is not updated for the next iteration of the loop, leading to an increased amount of borrowed tokens beyond what the user expected. The issue is present in `loop`, `loopETH`, `loopETHFromBorrow`, and the view function `wethToZapEstimation`.

Consider updating the value of the `amount` by deducting `fee` from it before using it for the deposit.

Update: *Resolved in pull request #206 at commits 1483fb5, c95a251 and bb54b79.*

Leveraged Positions May Receive Reduced Reward Emissions

The `loopETH` and `loopETHFromBorrow` functions set a user as exempt from eligibility for the duration of the loop, which disables any updates to `ChefIncentivesController` regarding changes in `rToken` balances. After the loop is finished, the exemption is revoked and `zapWETHWithBorrow` is called to lock more LP tokens when necessary, in order to ensure that the user maintains reward emission eligibility. When the `_zap` function locks more liquidity by calling `MultiFeeDistribution`'s `stake` function, updated `rToken` balances will correctly be reflected within `ChefIncentivesController`.

However, when users are still eligible even after reaching the desired leverage, the function `wethToZap` will return `0` and thus, `zapWETHWithBorrow` will not lock any more liquidity. This will cause users to miss rewards generated by the newly deposited amount since `ChefIncentivesController` has not been notified about the new balance. Reward emission will continue the next time liquidity is locked on behalf of this specific user.

Consider redesigning both `loopETH` and `loopETHFromBorrow` functions in order to ensure that `ChefIncentivesController` is always notified about final `rToken` balances after the position is leveraged.

Update: *Resolved in pull request #219 at commit 5a7a905. Both `loopETH` and `loopETHFromBorrow` have been refactored to behave like the `loop` function, where the*

the position is fully leveraged.

Slippage Tolerances May Be Too Large

When performing a `selfCompound` operation or when a hunter claims an auto-compound bounty, a series of swap operations take place so that an account's rewards can be converted into a combination of the base token (WETH or WBNB) and RDNT.

The first set of swap operations takes place when converting all `aToken` rewards into the base token. Each of those swaps has a maximum slippage ratio between 0% and 20%.

Once all rewards have been converted into the base token, the necessary amount of RDNT tokens charged as a fee when an auto-compound bounty is being claimed is swapped out from the base token balance. This swap is also subject to a 0%-20% maximum slippage.

Finally, the necessary percentage of the base token balance is swapped again into RDNT in order to be able to provide liquidity in the right ratio. This last swap operation is subject to the `ACCEPTABLE_RATIO` parameter, which can range from 0% to 100%.

All these slippage amounts can be compounded and an attacker may be able to extract up to the maximum slippage tolerance at all times.

Since Radiant users may have different amounts of capital invested in the protocol, different slippage settings should be enforced for each situation. Apart from enforcing sensible general parameters for these tolerances, consider giving users the ability to choose their own slippage settings so that smaller positions can get away with much tighter ratios.

Update: *Resolved in pull request #225 at commits 59d7f1d, 6cb87f9, 937a6b9, 843555b, af0b342 and f466ecc. Users are now able to specify their own custom slippage value for swaps. When auto-compounding is enabled, the* `MultiFeeDistribution` *contract stores the user's desired slippage tolerance for compounding. Users are prevented from using a slippage tolerance greater than 5%. If a user has not set a slippage limit for compounding, the maximum slippage tolerance configured for the system will be used, which is currently set at 5%.*

Missing `whenNotPaused` Modifier

have the [illegible] modifier already applied, the function `zapAlternateAsset` is missing that modifier. As a result, even when the contract is paused, it will still be possible to use the `zapAlternateAsset` function to stake additional liquidity.

Consider adding the `whenNotPaused` modifier to the `zapAlternateAsset` function so that it cannot be executed while the contract is paused.

Update: *Resolved in pull request #234 at commit 6e5faf7.*

Loss of RDNT Rewards When Compounding

Function `claimCompound` allows compounding accrued rewards into locked liquidity. The full list of available rewards is configured within `MultiFeeDistribution` and it contains all supported `rTokens` and the RDNT token itself. When rewards are claimed from the `MultiFeeDistribution` contract via `claimFromConverter`, this list is looped through and any available reward is transferred into the `Compounder` contract. At the time of writing, the RDNT token is added to that list but has no active emissions, so it will never get transferred.

However, the Radiant team confirmed that the RDNT token is added to the list of reward tokens within `MultiFeeDistribution` because in the future they might want to distribute them as rewards. If that happens, these tokens will get stuck on the `Compounder` contract because only `rTokens` are being handled, skipping the iteration when the token is RDNT.

Consider adding specific logic to handle RDNT rewards within the `claimCompound` function in order to avoid the potential loss of user rewards in the future.

Update: *Resolved in pull request #213 at commits 8bad016, e43df4e, 2e579cc, 77ca628, 02db1ea, 776bda9 and 26bea56. RDNT rewards are no longer forwarded to the `Compounder` contract. Additionally, a safety check has been added so that no reward token can be added if it implements the `UNDERLYING_ASSET_ADDRESS` function but is not part of the `rToken` suite.*

Missing `PriceProvider` Oracle Updates

The `PriceProvider` contract is responsible for providing pricing data to multiple contracts of the protocol. It uses either pool data or one of the following oracles: `ChainlinkV3Adapter`, `UniV3TwapOracle`, `UniV2TwapOracle`.

The following contracts are using `PriceProvider` but are missing oracle updates:

- `RadiantOFT`
- `Compounder`
- `EligibilityDataProvider`

It is recommended to always trigger the `update` function before using `PriceProvider` to query price data.

Update: *Resolved in pull request #235 at commits 6d1516f, c654e43 and 4467c8d.*

Use of Deprecated Chainlink Function

The contracts `BaseOracle` and `PriceProvider` are using Chainlink's deprecated `latestAnswer` function to retrieve the price of ETH. Although `latestAnswer` returns the price of the asset, it is not possible to check if the data is fresh.

The following instances of using `latestAnswer` were identified:

- `latestAnswer` in `BaseOracle.sol`
- `getTokenPriceUsd` and `getLpTokenPriceUsd` in `PriceProvider.sol`

Consider replacing calls to `latestAnswer` with Chainlink's `latestRoundData` function, and adding checks on the returned data to ensure the price value is positive, is not stale, and that the round is complete.

Update: *Resolved in pull request #220 at commit 1612d75. Calls to the deprecated `latestAnswer` function have been replaced with calls to `latestRoundData`. Additionally, transactions will be reverted if reported prices are older than a day, non-positive, or if the round is not complete.*

Incomplete Validation of Chainlink Price Data

The `ChainlinkV3Adapter` and `WSTETHOracle` contracts are using Chainlink's `latestRoundData` function, but are not fully validating the returned data. The returned price

incorrect prices being used by the protocol.

The following instances of missing security checks were identified:

- `latestAnswer` of `ChainlinkV3Adapter` contract.
- `latestAnswerInEth` of `ChainlinkV3Adapter` contract.
- `latestAnswer` of `WSTETHOracle` contract.

In each case, consider adding security checks on the returned data with proper revert messages if the price is stale or if the round is incomplete.

Update: *Resolved in pull request #221 at commits 0f5c958, 7ebf11a, a6c51b7, 0dc3ddf and cff685f. Prices retrieved from `ChainlinkV3Adapter` will revert if they are non-positive, older than a day, or from an incomplete round.*

Incorrect Reward Updates for Scheduled Emissions

The `ChefIncentivesController` contract supports two methods of distributing rewards. The protocol can either set a fixed value for `rewardsPerSecond` or change the `rewardsPerSecond` value according to an emission schedule. Usage of emission schedules requires a frequent check on the current rewards per second value against the schedule to determine when emissions should be adjusted. This logic has been implemented in the `setScheduledRewardsPerSecond` function that is triggered by `_updateEmissions`.

However, the `_updateEmissions` function is only used by `addPool` and `claim`, but not in `_handleActionAfterForToken` or `afterLockUpdate`. The latter two functions both call the `_updatePool` function in order to update pool rewards using the `rewardsPerSecond` value, but do not support emission schedule updates, which may lead to incorrect reward emissions until there is a new pool added or a claim is triggered.

Consider calling `_updateEmissions` whenever rewards need to be recalculated, to ensure that the `rewardsPerSecond` value tracks the current scheduled emission rate as closely as possible.

Update: *Acknowledged, not resolved. The Radiant team stated:*

reward inaccuracy is worth the gas savings on the nearly used function.

Possible Denial-of-Service for Compounding

The `claimCompound` function allows compounding accrued rewards from the `MultiFeeDistribution` contract by claiming them, converting them to the base token, and finally zapping them into locked LP tokens via `LockZap`. The logic for swapping rewards to the base token has been implemented in `_claimAndSwapToBase` by approving `uniRouter` to spend an `amount` of tokens and then executing `swapExactTokensForTokens` on Uniswap's router.

Since the swap is inside a `try-catch` block, it is possible that the swap reverts but the transaction succeeds, so long as the total received base token meets the slippage requirement. This leads to a scenario where the token has been correctly approved to be spent via `safeApprove` but then the actual spending did not happen. The implementation of `safeApprove` requires either the new allowance to be reset to `0` or the current allowance to be equal to `0`. This may not be true for `uniRouter` if any of the swaps revert since there will be an unspent allowance. This will lead to a denial-of-service situation for the `Compounder`.

Consider using the `SafeERC20` `forceApprove` function instead of `safeApprove`.

Update: *Resolved in pull request #184 at commit d72012f.*

Lack of Event Emission

The following functions do not emit relevant events after executing sensitive actions:

- `setMinStakeAmount`, `setBounties` and `addAddressToWL` in `BountyManager.sol`
- `setBountyManager`, `addRewardConverter`, `setLockTypeInfo`, `setAddresses`, `setLPToken` and `addReward` in `MultiFeeDistribution.sol`
- `setOracle`, `setPoolHelper`, `setAggregator` and `setUsePool` in `PriceProvider.sol`
- `setFallback` and `enableFallback` in `BaseOracle.sol`

- `toggleTokenForPricing` in `UniV3TwapOracle.sol`
- `setAdmin` and `addReward` in `MiddleFeeDistribution.sol`
- `setBountyManager`, `setCompoundFee` and `setSlippageLimit` in `Compounder.sol`
- `setOnwardIncentives`, `setBountyManager`, `setEligibilityEnabled`, `batchUpdateAllocPoint`, `setLeverager`, `setEndingTimeUpdateCadance` and `registerRewardDeposit` in `ChefIncentivesController.sol`
- `setPriceProvider`, `setMfd` and `setPoolHelper` in `LockZap.sol`
- `setLiquidityZap` and `setLockZap` in `UniswapPoolHelper.sol`

Consider emitting events after sensitive changes take place to facilitate tracking and to notify off-chain clients following the contracts' activity.

Update: *Resolved in pull request #227 at commits 53348f6 and ed9cabf.*

Uniswap Pool Initialization Fails if Pool Exists

The `UniswapPoolHelper` and `BalancerPoolHelper` contracts both implement an `initializePool` function that is responsible for creating a trading pair via `IUniswapV2PairFactory` or `IWeightedPoolFactory`, respectively. However, in the former case, Uniswap's `createPair` function will revert if a pair has already been created for the given tokens.

During the deployment process on a new chain, an attacker may create a Uniswap trading pair that matches the same pair Radiant intends to create before this `initializePool` function is called. This will cause the function to always revert from that point onward. This would require Radiant to upgrade the `UniswapPoolHelper` contract to resolve this issue.

Consider always checking if the desired Uniswap pool already exists before attempting to create it.

Update: *Acknowledged, not resolved. The Radiant team stated:*

> *As this issue only affects deployments on new chains, we will tackle this minor upgrade once (and if) it does actually end up occurring.*

The `BaseOracle` contract is inherited by several other contracts, but does not declare its own `__gap` variable. Gap variables provide a standard approach to reserve storage slots in base contracts, so that new variables can be safely added while avoiding collisions with the storage used by child contracts.

Consider adding storage gaps to inherited contracts by using gap variables to avoid future storage clashes.

Update: *Resolved in pull request #246 at commit 45833b7.*

Duplicated Code

There are instances of duplicated code within the codebase, which can lead to issues later on in the development lifecycle. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. Some examples are:

- Functions allowing the owner to recover stray ERC-20 tokens (`recoverERC20`) can be found with different implementations within the `MiddleFeeDistribution`, `MultiFeeDistribution` and `ChefIncentivesController`, and `BountyManager` contracts. All implementations are similar but only the first two emit a `Recovered` event.
- Internal functions to safely transfer ETH into a certain address (`_safeTransferETH`) are replicated on both the `Leverager` and `StargateBorrow` contracts.
- The `loop` and `loopETH` functions in the `Leverager` contract implement identical logic for updating a token's approved allowance for the `lendingPool` and `treasury`.
- When withdrawing unvested RDNT tokens, return values from `_penaltyInfo` are ignored and then amounts after the penalty are recalculated. Consider using the `amount`, `penaltyAmount` and `burnAmount` return values directly to determine if the earned amount from this lock is enough to cover the withdrawal instead of recalculating the values. Duplicating code like this may become error-prone as seen on issue M-01
- The `wethToZap` and `wethToZapEstimation` functions in the `Leverager` contract use an identical code block for computing `wethAmount`.

replicating the same logic.

- In the `Compounder` contract, the `selfEligibleCompound` function implements a special case of the `userEligibleForCompound` function where `_user = msg.sender`. Consider unifying the logic in an internal function that can be called by both public functions instead.
- The `slippageLimit` validation within the `Compounder` contract is duplicated in the `constructor` and in `setSlippageLimit`.
- When adding liquidity within `LiquidityZap`, it performs the optimal token amount calculation directly instead of leveraging the existing public `quote` function.
- Consider leveraging the `DustRefunder` contract within `LiquidityZap` to avoid duplicating logic to refund token leftovers after providing liquidity.
- When calculating the LP token price via `getLpPrice` in `UniswapPoolHelper`, consider leveraging the existing `getReserves` function in order to retrieve the pool reserves and total supply instead of duplicating its logic.
- When calling `ltv`, the asset configuration is queried directly from the lending pool. Consider making the existing `getConfiguration` function public so that it can be reused for this query.

Instead of duplicating code, consider extracting it into a shared function or a helper library when applicable.

Update: *Resolved in pull request #230 at commits 6c94f65, f946b0c, 57c2ee1 and 22cd008.*

Missing Error Message in `require` Statement

The `require` statement on line 1172 of `MultiFeeDistribution.sol` lacks an error message.

Consider including specific, informative error messages in `require` statements to improve overall code clarity and facilitate troubleshooting whenever a requirement is not satisfied.

Update: *Resolved in pull request #224 at commits d14d322 and e869cd9.*

Missing or Incorrect Docstrings

In `EligibilityDataProvider.sol`:

- The internal `_lockedUsdValue` function retrieves the USD value for the LP token but the docstring states that the reported value is in ETH.

In `wstethOracle.sol`:

- The function `latestAnswer` returns the price for `wstETH/USD` instead of `wstETH/ETH` as the comments suggest.

In `MultiFeeDistribution.sol`:

- Line 143 is a stray comment that is not associated with any code.
- The `initialize` function only has `@param` tags for 4 out of 10 input parameters and has a `@dev` comment that refers to it as being a constructor instead of an initializer.
- The docstring for `setLockTypeInfo` says the function adds a new reward token, but the function actually sets the lock periods and corresponding multipliers. The `@notice` documentation for `setLockTypeInfo` is a copy from the `addReward` function's docstring.
- There is a section divider named "View functions", but non-view setter functions are mixed with getter functions in this section.
- Documentation associated with the `totalBalance` function states that it returns the total balance of an account, including earned tokens. This is not the case when the staking token is the LP token instead of the RDNT token.
- The `earnedBalances` `@dev` comment states that earned balances can be immediately withdrawn for a 50% penalty. This is not the case since the penalty factor is implemented as a function of time, starting at 90% and finishing at 25% right before vesting is complete.
- The `_notifyUnseenReward` function's documentation states that it is meant for rewards other than the staking token, which in this case is the LP token. However, the function only checks if the `token` argument is the RDNT token.
- All references to `burn` and `burnAmount` are not valid. Rather than being burnt, the "burnt" share of early-exit penalty tokens is sent to the `starfleetTreasury` contract.

Consider renaming the function to `ieeWithdrawableBalance` and updating the corresponding documentation.

In `MiddleFeeDistribution.sol`:

- Missing `@param _operationExpenses` and `@param _operationExpenseRatio` for the `setOperationExpenses` function
- Missing `@param _configurator` for the `setAdmin` function
- Missing `@param _rewardsToken` for the `addReward` function
- Missing `@param _rewardTokens` for the `forwardReward` function
- Missing `@return` for the `getRdntTokenAddress` function
- Missing `@return` for the `getMultiFeeDistributionAddress` function
- Missing `@param asset` and `@param lpReward` for the `emitNewTransferAdded` function
- Missing `@param tokenAddress` and `@param tokenAmount` for the `recoverERC20` function
- Incorrect `@notice` comment for the `recoverERC20` function
- The `forwardReward` function docstring is a copy of the `recoverERC20` function docstring, which is misleading because it does forward tokens to the `MultiFeeDistribution` contract, but not LP rewards such as BAL as indicated.

In `ChefIncentivesController.sol`:

- The `@title` docstring is a copy of the `@title` from the `UniV3TwapOracle` contract.
- The docstring for the internal `_mint` function states that it "can be called by owner or leverager contract". This appears to be copied from the `setEligibilityExempt` function docstring.
- The `emissionSchedule` array documentation states that block number will be used to determine which schedule is the current one, but in reality it uses `block.timestamp` instead, so offsets refer to seconds not blocks.

In `RadiantOFT.sol`:

In `Leverager.sol`:

- Missing `@param _cic` for the `constructor` function
- Missing `@return` for the `wethToZapEstimation` function
- Missing `@return` for the `wethToZap` function
- Missing `@return` for the `requiredLocked` function

In `StargateBorrow.sol`:

- Incorrect docstring for the `borrow` function.
- Missing `@return` for the `getXChainBorrowFeeAmount` function.
- Missing all `@param` documentation and `@return` for the `quoteLayerZeroSwapFee` function

In `LockerList.sol`:

- Missing `@return` for the `lockersCount` function
- Missing all `@param` documentation `@return` for the `getUsers` function
- Missing `@param user` for the `addToList` function
- Missing `@param user` for the `removeFromList` function

In `BountyManager.sol`:

- Missing `@param _eligibilityDataProvider` and `@param _compounder` for the `initialize` function.
- Incomplete `@return` documentation for the `getMfdBounty` function. It describes `totalBounty` but not `issueBaseBounty`.
- Incomplete `@return` documentation for the `getChefBounty` function. It describes `totalBounty` but not `issueBaseBounty`.
- The docstring for `getAutoCompoundBounty` states that `MFDPlus.claimCompound` is called and that `MFDPlus` is the incentivizer; in both cases `MFDPlus` should be `Compounder`.
- Missing `@return issueBaseBounty` documentation for the `getAutoCompoundBounty` function.

In `Compounder.sol`:

- Missing `@return fee` documentation for the `claimCompound` function.

In `LockZap.sol`:

- The docstring for the `LockZap` contract is a copy of the `Stargateborrow` contract's docstring
- Missing `@return` for the `zap` function
- Missing `@return` for the `zapOnBehalf` function
- Missing `@return` for the `zapFromVesting` function
- Missing `@return` for the `_zap` function

In `BaseOracle.sol`:

- Incorrect `@notice` comment for `latestAnswerInEth`. It returns the token price in ETH, not USD.

The following instances lack docstrings entirely:

- The `ChainlinkV3Adapter` contract
- The `Compounder` contract, with the exception of the `claimCompound` function
- Most of the functions within the `wstethOracle` contract
- The `onUpgrade` function in `MultiFeeDistribution`
- The `getPrice` and `getReserves` functions within `BalancerPoolHelper`
- The entire `UniswapPoolHelper` contract, with the exception of the `swapToWeth` function

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #190 at commits 49d2c01, 8f4573e, and 4afdeb2. All indicated instances of incorrect documentation have been corrected and any missing docstrings have been*

Some instances were found across the codebase where an `assert` statement is used:

- Line 63 in `LockerList.sol`
- Line 256 in `EligibilityDataProvider.sol`
- Line 218 in `LiquidityZap.sol`

A failed assertion will revert with a Panic exception in the EVM. According to the Solidity documentation, the `assert` statement should only be used to test for internal errors where an invariant is violated (i.e., it should be reserved to test for "this should never happen" states). Well-designed Solidity contracts should never generate a Panic, even when an input parameter is invalid.

Consider replacing all `assert` statements with an equivalent `require` or `revert` statement.

Update: *Resolved in pull request #223 at commits 41ae45a and 7154ae9.*

Missing Slippage Protection for Direct Zapping

The `LiquidityZap` contract implements the `zapETH` and `addLiquidityETHOnly` functions which allow adding liquidity by directly providing ETH. The received ETH is wrapped into WETH tokens and then half of it is swapped for RDNT.

However, there is no slippage protection when swapping WETH for RDNT, making users of these functions vulnerable to sandwich attacks. An attacker may monitor the mempool and front-run `zapETH` and `addLiquidityETHOnly` transactions by buying a significant amount of RDNT tokens prior to a user's transaction and then selling them immediately afterward, profiting at the cost of the user.

Note that this contract is only used in the BSC deployment, so all references to ETH and WETH are technically BNB and WBNB. ETH terms have been used for convenience since both functions contain references to it in their names.

Update: *Resolved in pull request #264 at commit 523d4ab.*

Invalid Reward Tokens Can Brick Reward Claiming

In order to get a new token added to the list of reward emitting tokens, addresses with the minter role can do so via the `MultiFeeDistribution` contract's `addReward` function, or alternatively, the `owner` or the `admin` of `MiddleFeeDistribution` can do so as well by calling its `addReward` function.

These tokens are always assumed to be `AToken` instances or the RDNT token itself. However, if by mistake a different token was added, there is no validation to prevent it as long as it is not the zero address.

Adding an erroneous token will cause multiple parts of the protocol to revert, since this token will not conform to the `IAToken` interface and there will not be a proper response from the `AaveOracle` when trying to figure out its price. Furthermore, there is no way to remove a reward token from the list, forcing the team to issue an emergency upgrade in order to address the problem if it occurs.

Consider implementing logic to remove reward tokens from the list when they are no longer supported or when they are added by mistake.

Update: *Resolved in pull request #248 at commits 6573a30, 16503ae, fe562e4 and 855cfa6. The Radiant team added a function that allows the removal of existing reward tokens if necessary.*

Unchecked ERC-20 Return Value

According to the ERC-20 token standard, many function calls return a `bool` to signal the success or failure of the call. Such functions include `transfer`, `transferFrom` and `approve`. While many tokens throw an exception when these functions are unsuccessful, some tokens instead return `false`.

In the `LockZap` contract, the function `zapAlternateAsset` does not check the return value of the `transferFrom` call.

which is already being imported by the [illegible] contract. This library safely handles ERC-20 tokens that behave in different ways, and ensures that all calls revert on failure, regardless of whether the underlying token does.

Update: *Resolved in pull request #244 at commit d30d1ab. The unchecked* `transferFrom` *function call was replaced with a call to* `safeTransferFrom`*.*

Assets With More Than 18 Decimals Not Supported for Zapping

The `LockZap` contract allows zapping any ERC-20 asset into staking via the `zapAlternateAsset` function by converting it to the base token (WETH or WBNB depending on the chain), adding liquidity, and then staking those LP tokens within the `MultiFeeDistribution` contract. In order to handle slippage, the expected amount of base token is calculated by assuming that the asset's decimals will be equal to or smaller than 18, which may not always be correct. Attempting to zap assets with more than 18 decimals will result in an underflow that will make the transaction revert.

In order to avoid unexpected reverts and be compatible with more asset types, consider refactoring the `expectedEthAmount` calculation in `zapAlternateAsset` to also support assets with more than 18 decimals.

Update: *Resolved in pull request #191 at commit 95400d3.*

Users Can Burn Their Tokens Without Incentives

The RDNT token features an unrestricted `burn` function which allows anyone to voluntarily burn any token holdings without incentive.

While burning tokens is a necessary action for specific protocol flows, consider restricting who can call this function, in order to avoid accidental loss of user funds in the event that the function is called by mistake.

Update: *Acknowledged, not resolved. The Radiant team stated:*

> *We prefer to maintain a degree of flexibility here that will allow for potential future utility that is yet unforeseen.*

`initialize` function to set up storage variables. Additional initialization steps are done in the `initLiquidityZap` function which is responsible for setting up pool configuration. The contract uses a boolean variable named `initialized` in order to ensure that `initLiquidityZap` can be executed only once.

However, `LiquidityZap`'s variable declaration of `initialized` shadows the `initialized` variable from the parent contract `Initializable`.

Although the code works as expected, this practice is error-prone and should be avoided. Consider renaming the `LiquidityZap` storage variable `initialized` to something different such as `initializedLiquidity`, to ensure it does not shadow the other `initialized` variable from the `Initializable` contract.

Update: *Resolved in pull request #172 at commits b0cd872 and b416eff.*

Duplicate Emission Schedules Are Allowed

When appending a new set of emission schedules, their starting offsets should be in ascending order to make sure they are activated correctly as time goes by.

However, when enforcing this ascending order, it is possible to schedule different emission rates with the same starting time offset. This will cause only the latest one in the list to be enforced when its starting timestamp is reached, ignoring all the other schedule items.

In order to avoid potential misconfiguration and unexpected results, consider reverting when attempting to set an emission schedule that contains duplicated start time offsets.

Update: *Resolved in pull request #171 at commit 95fe10c. Before a new emission schedule can be added, it is required that its starting offset is unique and does not match any starting offset in the existing list of emission schedules.*

Locked ETH in Contracts

There are several instances in the codebase where ETH can be locked:

- The `receive` function of `LockZap.sol`

In order to avoid accidentally locking ETH on this contract, consider removing the first two instances, and restricting the third one in `Stargateborrow` so that only the WETH contract can send ETH.

Update: *Resolved in pull request #214 at commit 750dcbf. ETH withdrawal functions were added to `LockZap` and `StargateBorrow`, protected by the `onlyOwner` modifier. In `UniswapPoolHelper` the `receive` function was removed since the Radiant team deemed it unnecessary.*

Unnecessary Truncation of Balancer LP Total Supply

The `getReserves` function in the `BalancerPoolHelper` contract returns the amounts of RDNT and WETH in the Balancer pool and the corresponding amount of LP tokens.

However, the `lpTokenSupply` value is divided by `1e18` before being returned, effectively removing all decimals from the value. This calculation is not done in the `getReserves` function of the `UniswapPoolHelper` contract.

Even though this `lpTokenSupply` value is never used within the codebase, consider removing the division by `1e18` in order to return the total supply of Balancer LP tokens with no precision loss.

Update: *Resolved in pull request #215 at commit 23fbf42. The unused `lpTokenSupply` return value was removed from the `getReserves` function.*

Unnecessary access control and incorrect mutability

The `BalancerPoolHelper` contract limits access to its `getSwapFeePercentage` function with the `onlyOwner` modifier. However, there is no reason to limit access to value, since it can be read directly from the pool contract. Additionally, this function does not change the contract's state, but is missing `view` function state mutability.

Consider removing the `onlyOwner` modifier from the `getSwapFeePercentage` function, and also adding `view` mutability to correctly reflect its behavior.

Several contracts within the codebase include commented-out lines of code without giving developers enough context on why those lines have been discarded, thus providing them with little to no value at all. For instance:

- Line 579 in `ChefIncentivesController.sol`
- Lines 48, 147 and 796 in `MultiFeeDistribution.sol`

As the purpose of these lines is unclear and may confuse future developers and external contributors, consider removing them from the codebase. If they are to provide alternate implementation options, consider extracting them to a separate document where a deeper and more thorough explanation could be included.

Update: *Resolved in pull request #245 at commit 100d94e. All instances of commented-out code were removed except one, where the Radiant team added an inline comment that explains the rationale for leaving that instance in place.*

Unsafe ABI Encoding

It is common practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first option is not typo-safe, and the second option is not type-safe. The result is that both of these methods are error-prone and should be considered unsafe.

Line 140 of `BountyManager` uses the unsafe `encodeWithSignature` ABI encoding.

Consider replacing all the occurrences of unsafe ABI encodings with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function, and also avoids errors caused by typos.

Update: *Resolved in pull request #216 at commit ce21edd.*

Tokens May Get Stuck in `Compounder` Contract

The `claimCompound` function allows compounding accrued rewards from the `MultiFeeDistribution` contract by claiming them, converting them to the base token, and

router.

Because the swap is inside a `try-catch` block, it is possible that the swap fails but the transaction succeeds (as long as the slippage requirement is satisfied), which leads to a scenario where tokens will get stuck in the `Compounder` contract without any withdrawal mechanism.

In order to avoid this scenario, consider allowing the swap transaction to revert by removing the `try-catch` block.

Update: *Resolved in pull request #183 at commit 150782f. Rather than removing the `try-catch` block, the team decided to revert on the `catch` block with a custom `SwapFailed` error that reports the underlying asset address and the intended swap amount.*

Inconsistent Use of Named Return Values

Throughout the codebase, most functions that return values use named return variables. However, there are several instances where functions use unnamed return values, such as:

- The `getBridgeFee function` within the `RadiantOFT` contract
- The first return value from the `lockedBalances` function within the `MultiFeeDistribution` contract
- The `lockersCount` and `getUsers` functions within the `LockerList` library

For consistency, consider always using named return values.

Update: *Resolved in pull request #196 at commit e6a9cde.*

Incorrect Input Validation

The `EligibilityDataProvider` and `Compounder` contracts incorrectly validate input parameters used to set the values of storage variables `priceToleranceRatio` and `slippageLimit`:

- `EligibilityDataProvider`'s `setPriceToleranceRatio` function incorrectly implements validation of the `_priceToleranceRatio` parameter, which makes it

any value as a slippage limit without triggering a revert.

In both cases, the intended behavior is to accept values between `8000` (80%) and `10000` (100%).

Consider updating the input validation logic to ensure that only values within the intended range are permitted.

Update: *Resolved in pull request #195 at commit 82e419d and pull request #230 at commit 22cd008.*

Inconsistent Amount Calculation When Providing Liquidity

When users zap their RDNT tokens from vesting, the optimal amount of base token necessary to provide liquidity is calculated via the pool helper `quoteFromToken` function, and then an extra 3% safety margin is added on top.

This is inconsistent with the `_zap` function's call to the same `quoteFromToken` function, where no extra safety margin is added.

Consider applying this extra safety margin consistently when calculating the amount of base tokens required to supply liquidity.

Update: *Resolved in pull request #247 at commit 503383f. The Radiant team decided not to use an extra safety margin by using the already existing pool helper's `quoteFromToken` function directly.*

RDNT Bridge Fee Is Being Overcharged

The new Radiant OFT v2 token allows cross-chain transfers via LayerZero. Such a transfer involves two types of fees: the native LayerZero fee and a bridging fee charged by the Radiant protocol.

In order to enforce cross-chain compatibility (even with non-EVM chains), prior to a transfer LayerZero needs amounts to be truncated to a certain number of decimals configured as token

multiple. This means that for example when users want to transfer [illegible] tokens, only [illegible] tokens will be transferred.

However, the `getBridgeFee` calculation does not truncate input amounts so in the extreme scenario where the minimum value of `1e10` is transferred, the final fee charged will be almost double what it would have been if the amount was truncated first. This effect becomes much less noticeable for larger transfer amounts.

Consider truncating the transfer `_amount` before calculating the bridge fee in order to not overcharge users.

Update: *Resolved in pull request #217 at commit d59c6a1.*

Use of Low Liquidity Source When Compounding on Arbitrum

User rewards can be compounded into locked LP tokens via the `Compounder` contract. They can do so themselves, or a hunter can claim a bounty for doing so on their behalf. When claiming an auto-compounding bounty, several swaps are made:

- Each reward token is withdrawn for the underlying token and then swapped for the base token (WETH on Arbitrum and WBNB on BSC).
- A percentage of the total base token is swapped for RDNT as a fee that will be split between the hunter and the bounty reserve.
- Half of the remaining base token will be swapped for RDNT so that liquidity can be provided in the right proportion via `zapOnBehalf`.

In the case of Arbitrum, liquidity is provided on a Balancer pool. This means that the highest liquidity available for WETH and RDNT will be found on that Balancer pool. However, the second swap is performed in a Sushiswap pair, which has far lower liquidity since there is no incentive for users to provide liquidity there.

Consider enforcing consistency to always get the best liquidity possible by performing all swaps between the base token and RDNT on the pool where users lock their LP tokens.

Update: *Acknowledged, not resolved. The Radiant team stated:*

Use of Hard-Coded Values

- The `swapToWeth` function in the `BalancerPoolHelper` contract contains multiple hard-coded addresses and pool ids that are too tightly coupled with the specific Arbitrum deployment.
- Functions `borrow` and `borrowETH` are using fixed slippage set to `1%`. Consider adding functionality to adjust slippage based on liquidity and market conditions.
- The `wethToZap` function is using a fixed value of `6%` as a margin for wrapping WETH.

Consider creating constants to store these values or assigning them dynamically upon deployment in order to avoid deployment issues on new chains.

Update: *Resolved in pull request #231 at commits 74f00dc and c91894b.*

Uniswap V2 Oracle Does Not Scale Price

The `UniV2TwapOracle` contract implements the `consult` function, which determines the current RDNT price by calculating how many base tokens would be received when swapping one unit of RDNT. When `consult` calls the internal `_consult` function to get the price, it sets `_amountIn` equal to `10 ** decimals`, which represents 1 token.

However, the function `latestAnswerInEth` of the parent contract `BaseOracle` expects the price returned by `consult` to always be scaled to 18 decimals. Based on this expectation, `latestAnswerInEth` divides the price received from `consult` by `10 ** 10`, which should result in the output having 8 decimals. If the quote token does not use 18 decimals, this will result in an incorrect price value, because the return value of `consult` will violate `latestAnswerInEth`'s precondition. Correct scaling was implemented in `UniV3TwapOracle` where the amount of received tokens is scaled to 18 decimals.

Consider scaling the price returned by the `consult` function to 18 decimals.

Update: *Acknowledged, not resolved. The Radiant team stated:*

> *We don't ever intend to utilize any token outside of RDNT in this context (and we don't intend to utilize any version of the RDNT token that doesn't have 18 decimal places).*

identified. For example:

- In the `MultiFeeDistribution` contract, the `mint` function does not mint RDNT tokens. The `mint` function assumes that the rewards being added have already been minted and are available to be distributed by the MFD, which may not be true.
- The internal `_withdrawExpiredLocksFor` function features a boolean `isRelockAction` parameter which is described as an indicator to determine whether the current staking operation is a relock. However, it will not relock those tokens if it is set to `true`. Additionally, the alternative `withdrawExpiredLocksWithOptions` function describes the same parameter as `_ignoreRelock`, which is the opposite meaning for the same value.
- The `setFee` function within the `RadiantOFT` contract accepts an input parameter called `_fee` which will be used to update the state variable `feeRatio`. If this `_fee` value is larger than a certain threshold, it reverts with an error message specifying "Invalid ratio" and if it does not, it successfully emits the `FeeUpdated` event. Consider renaming `setFee` to `setFeeRatio`, `_fee` to `_feeRatio`, and `FeeUpdated` to `FeeRatioUpdated`.
- The `getBestBounty` function in the `BountyManager` contract is responsible for finding and calculating the bounty that can be received for the specified user. It can be executed either by passing an `_actionTypeIndex` value that corresponds to one of the bounty types (`MFD`, `CIC` or `Compounder`) or by passing the value of `0` which should iterate over all bounty types and find the best available bounty for the user. The logic of the function does not implement that behavior. It iterates over the `MFD`, `CIC` and `Compounder` bounties, until finding one that offers any bounty, not until finding the best one. Consider changing the function name to `getAvailableBounty`.
- In the `MultiFeeDistribution` contract, the `earnedBalances` function returns `total` and `unlocked` earnings balances. The `total` variable name is confusing since it seems to account for both vested and already unlocked amounts. However, `total` in reality means the total amount in vesting (before applying penalties), excluding any already unlocked amounts.
- The `claimFromConverter` function within the `MultiFeeDistribution` contract is called by the `Compounder`, but the docstrings indicate that rewards are claimed by and

Consider providing explicit and consistent naming for both variables and functions all across the project in order to avoid confusion and improve readability.

Update: *Resolved in pull request #232 at commits 666d900, dcaa6cd and dc0934b.*

`LockZap` Does Not Correctly Enforce Maximum Borrowing Power

When zapping assets into locked LP tokens, users need to provide both RDNT and the base token (WBNB or WETH) in equal value in order to provide liquidity. Among other options, users can provide the base token themselves by approving the `LockZap` contract to spend them, or they can borrow it from the lending pool against their deposited collateral.

When users decide to borrow against their collateral, there is a check in place to prevent users from borrowing amounts larger than their maximum borrowing power, in order to ensure they maintain an overcollateralized position. This check asks the lending pool how much ETH this user can borrow and compares it with the requested amount, scaled down to 8 decimals.

The original `getUserAccountData` function from Aave V2 documentation states that the returned value `availableBorrowsETH` specifies an ETH amount with 18 decimals. However, the modified Radiant version of the lending pool returns all values as USD-denominated amounts with 8 decimals. This comparison is flawed since it is comparing USD amounts with ETH amounts. The end result is that at current prices, this condition is always satisfied since ETH amounts are usually lower than the maximum borrowing power expressed in USD amounts, even when trying to borrow beyond your limit.

Consider converting the amount in ETH to USD by querying the `ethOracle` in order to correctly enforce that borrow amounts are lower than the actual borrowing power. Even if the lending pool is expected to revert when a user tries to borrow beyond their limit, external code should not be relied on to prevent actions that may lead to insolvency.

Update: *Resolved in pull request #192 at commit db69559. The `ethOracle` Chainlink ETH/USD price feed is now queried to get the requested borrow `amount` in USD. However, the deprecated `latestAnswer` function is used to perform the query.*

Incorrect Value Used for Slippage Calculations

base token (WETH or WBNB) and then does the same for the received LP tokens in `_calcSlippage`. After that it makes sure the required `ACCEPTABLE_RATIO` is satisfied.

However, it assumes that the entire token amounts passed to `zapTokens` have been used to add liquidity, which is not always true. This may lead to unexpected reverts due to acceptable slippage not being met when the value of the passed asset is higher than the actual value used when providing liquidity.

Consider calculating the exact amount that was used for adding liquidity by retrieving the token balances before and after adding liquidity through the `zapTokens` function.

Update: *Resolved in pull request #178 at commits 39ff88b, 7d447a4 and af19454. In both the* `_zap` *and* `zapAlternateAsset` *functions, the difference in the WETH or WBNB amount held by the* `LockZap` *contract before and after zapping is now used as the input to the* `_calcSlippage` *function.*

Incorrect Fee Returned in Case Compounding Is Not Being Executed

The function `claimCompound` of the `Compounder` contract returns the fee that is charged for compounding. In case it is a self-compounding operation being executed, there is no fee charged and the function returns `0`. It is also possible to pass the boolean parameter `_execute` set to `false` as a simulation mode, skipping actual execution. The `selfCompound` function always sets `_execute` to `true`.

However, when `claimCompound` is called directly by a user on behalf of themselves with `_execute` set to `false`, it always returns the auto-compound fee instead of returning `0` for this self-compound operation.

It is recommended to always return a fee value of `0` for self-compounding in `claimCompound`. In addition, consider removing the `fee` returned by the `selfCompound` function to make it clear that there is no fee charged when performing it.

Update: *Resolved in pull request #218 at commit e409b80.*

Flawed `lastEligibleTime` Function

users are currently eligible.

However, if a given user is not currently eligible, this function will still loop through their locks and incorrectly return the nearest unlock time, even when the amount locked does not cover the minimum requirement for eligibility.

Consider handling the scenario where a given user is not eligible differently or, alternatively, provide some more documentation on why this behavior is expected.

Update: *Resolved in pull request #226 at commit 80324eb.*

Incorrect Bounty Returned When `BountyManager` Reserve Is Low

The `executeBounty` function of the `BountyManager` contract returns the amount of RDNT token that will be paid as a `bounty` to the hunter that claims it. At the end of its execution, `executeBounty` uses the `_sendBounty` function to transfer the `bounty` amount to the caller.

However, `executeBounty` assumes that the total `bounty` was successfully paid, which is not correct when the reserve does not hold enough funds. In that case, the logic of `_sendBounty` does a partial payout by sending the remaining reserve balance of RDNT and returns that value from the function. Regardless of the actual amount paid to the hunter, `executeBounty` will always return the full bounty amount.

Consider modifying `executeBounty` to return the exact amount paid to the hunter, even when the reserve is not large enough. To ensure an adequate balance at all times in the `BountyManager` reserve, consider implementing logic that reverts the `executeBounty` transaction when there are insufficient funds to pay a bounty. Additionally, consider using OpenZeppelin Defender to detect this event and subsequently pause the contract or fill the reserve.

Update: *Resolved in pull request #222 at commit 1a05a30. The `executeBounty` function no longer assumes that the reserve balance was sufficient to pay the entire bounty, and now returns the actual bounty amount transferred.*

Logic Contracts Initialization Is Allowed

already calling the [illegible] function on any of the implementation contracts. In this case, the state of the given implementation would be initialized to some meaningful value.

Leaving an implementation contract uninitialized is generally an insecure pattern to follow.

For each initializable contract, consider adding a constructor that calls the `_disableInitializers` function to ensure that the implementation contracts cannot be initialized by a third party.

Update: *Resolved in pull request #233 at commit b00066b.*

Lack of Input Validation

Throughout the codebase, there are several functions that lack input validation. In particular:

- The `owner` of the `MultiFeeDistribution` contract can arbitrarily update the value for the `rewardsLookback` variable, while it is enforced to be non-zero during initialization.
- The `owner` of the `Leverager` contract can call `setFeePercent` to update the value of `feePercent` to 10000 (100%). Consider adding a check that will enforce setting the `feePercent` to a reasonable value.
- Consider enforcing that `borrowRatio` is larger than 0 apart from being lower than 100% in order to avoid dividing by zero. Note that the functions `loopETH` and `loopETHFromBorrow` should also follow the same recommendation.
- The `setXChainBorrowFeePercent` function in the `StargateBorrow` contract allows `owner` to update the value of `xChainBorrowFeePercent` to 10000 (100%). Consider adding a check that will enforce setting the `xChainBorrowFeePercent` to a reasonable value.
- The value of `ACCEPTABLE_RATIO` in the `LockZap` contract that is responsible for handling slippage can be set to any value between 0 and 10000 via `setAcceptableRatio`. Consider adding a check that will enforce setting the `ACCEPTABLE_RATIO` to a reasonable minimum value as it happens in other parts of the codebase. Additionally, the `initialize` function does not enforce any checks on this value.

enforce that LP tokens cannot be accidentally burnt by checking that the recipient `to` is not the zero address, except the `standardAdd` function. Additionally, this function is the only one that does not check if either of the input amounts is zero. Consider enforcing consistency on input validations including the `standardAdd` function to prevent accidental user loss of funds.

- The `setFee` function in the `RadiantOFT` contract allows setting the bridge fee equal to 10000 (100%). Consider adding a reasonable limit on the maximum bridge fee.
- The `setTWAPLookbackSec` and `initialize` functions in the `UniV3TwapOracle` both set the value of `lookbackSecs` that is used as a TWAP lookback period. Consider adding a check to these functions that will enforce a minimal value of lookback seconds.
- Consider adding a check that `_actionTypeIndex` is less than or equal to `bountyCount` in order to avoid an out of bounds error when calling the `getBestBounty` function.

Consider implementing the suggested validations in order to prevent unexpected behaviour that may lead to potential attacks on the protocol.

Update: *Resolved in pull request #207 at commits a2de2ee, 400b1b1, bb4b163, ca7ce42, 3eac7d6, 20c279c, 4e23cff and 2ce50e6.*

Incorrect Event Emission

Within the `LockerList` contract, if an attempt is made to add a `locker` address that already exists to the `userlist`, it will not be re-added since there is a check that the address is not already inserted. However, the `LockerAdded` event is being emitted regardless of whether the user is already on the list.

To avoid hindering the task of off-chain services by emitting misleading information, consider only emitting the `LockerAdded` event when adding an address that is not already inserted in the user list.

Update: *Resolved in pull request #202 at commit 2d7307a.*

Incorrect or Inconsistent Error Types

- The `addReward` function in the `MultiFeeDistribution` contract is used to add reward tokens other than RDNT to the system. It contains a check that ensures the `_rewardToken` input address is not zero, but instead of reverting with the `AddressZero` error it uses the `InvalidBurn` error.
- The `_stake` function in the `MultiFeeDistribution` contract reverts with `InvalidAmount` if the provided `typeIndex` is beyond the size of the lock types array. Consider using the more appropriate `InvalidType` error.
- The `setLPToken` function in the `MultiFeeDistribution` contract correctly reverts with `AddressZero` when trying to set the staking token value to the zero address. However, if this value was already set, it is also reverting with `AddressZero`. Consider leveraging the `AlreadyAdded` error or creating a more specific one such as `AlreadySet` in order to be more informative.
- The `withdraw` function in the `MultiFeeDistribution` contract reverts with `InvalidAmount` when trying to withdraw a zero amount. Consider using the more explicit `AmountZero` error type instead.
- When initializing the `BountyManager` contract, if `_hunterShare` is larger than 10000 it will revert with an `InvalidNumber` custom error. However, when using the `setHunterShare` function to update the `hunterShare` variable, it reverts with `Override` under the same conditions.

Consider updating the aforementioned instances in order to make reported errors more accurate and explicit.

Update: *Resolved in pull request #209 at commit b61bfe1.*

Configuration Changes May Negatively Impact Users

Within the codebase there are several instances of configuration update actions that may have a negative impact on a user's transaction if the configuration change happens just before the user's transaction:

- The `setLockTypeInfo` function in the `MultiFeeDistribution` contract can be used by the owner to set or update the lock periods and their multipliers. Any user who wants

`setDefaultRelockTypeIndex` is confirmed, this user will end up potentially committing to an unknown locking period and reward multiplier, different from their expected values.

- The `Leverager` contract charges a fee to users that want to leverage their positions. The `feePercent` can be set at any time by the `owner` using the `setFeePercent` function. That leads to a scenario where the owner's transaction to update this fee may be confirmed before the user's looping transaction, resulting in the user being charged a different fee than expected.
- The `Compounder` contract charges a fee to users that want to compound their rewards into locked liquidity. The `compoundFee` can be set at any time by the `owner` using the `setCompoundFee` function. If the owner's update transaction gets confirmed before the user's compound transaction, they will be charged a different fee than expected.
- The `setXChainBorrowFeePercent` function of the `StargateBorrow` contract allows updating the cross-chain fee that will be charged when users want to receive funds on another chain. If the owner's transaction to update this fee is included before the user's borrow transaction, they will end up being charged a different fee than expected.

In these cases, consider allowing users to specify the expected values of the relevant configuration state as additional function parameters so that their transaction will revert when the configuration does not match the expected state.

Update: *Acknowledged, not resolved. The Radiant team stated:*

> *During the period of a protocol variable change, the front end can be frozen until the new values are active and displayed. In our opinion, the proposed changes are not worth the likely UX decline.*

Rewards Can Be Emitted to the `MiddleFeeDistribution` And `MultiFeeDistribution` Contracts

The `ChefIncentivesController` contract implements the `handleActionAfter` function that is used by `IncentivizedERC20` (`_transfer`, `mint` and `burn` functions) and `StableDebtToken` (`mint` and `burn` functions) in order to track any changes in user balances after performing those actions. It does not track the balances of `rewardMinter` (the

However, this can be bypassed if the `requalifyFor`, `stake`, `withdrawExpiredLocksFor`, or `withdrawExpiredLocksForWithOptions` functions are called with the address of either the `MiddleFeeDistribution` or `MultiFeeDistribution` contracts as a parameter.

Consider enforcing consistency and preventing any action intended only for protocol users from being performed on behalf of the `MiddleFeeDistribution` and `MultiFeeDistribution` contracts.

Update: *Acknowledged, not resolved. The Radiant team stated:*

> *It is intended that these functions can be executed on behalf of any valid EVM addresses. We have reviewed the exclusion of the MFD and `MiddleFeeDistribution` contracts from the rewards and believe that functionality is sound.*

Missing Zero Address Checks

Multiple contracts are missing zero address checks for setting storage variables. Accidentally setting an address variable to address zero might result in an incorrect configuration of the protocol. For instance, the following parameters are not checked:

- `_multiFeeDistribution` in the `initialize` function of the `MiddleFeeDistribution` contract.
- `_lpToken` in the `setLPToken` function of the `EligibilityDataProvider` contract.
- `_rdntAddr`, `_wethAddr`, `_routerAddr` and `_liquidityZap` in the `initialize` function of the `UniswapPoolHelper` contract.
- `_poolFactory` in the `initialize` function of the `BalancerPoolHelper` contract.
- `_stETHUSDOracle` and `_stEthPerWstETHOracle` in the `initialize` function of the `WSTETHOracle` contract.

Consider adding zero address checks to the listed parameters in order to avoid accidental misconfigurations.

Both pool helper contracts `UniswapPoolHelper` and `BalancerPoolHelper` implement a `swapToWeth` function that is responsible for swapping from a variety of source assets into the base token (WETH or WBNB depending on the specific chain) and then transferring them to the message sender. This functionality is used only by the `LockZap` contract and unlike the related pool helper functions `zapWETH` and `zapToken`, it can be called by anyone.

An attacker may steal any token balance held by the `UniswapPoolHelper` or `BalancerPoolHelper` contracts by calling `swapToWeth`, which will swap the specified `_inToken` to the base token.

Consider adding a check to the `swapToWeth` function of `UniswapPoolHelper` and `BalancerPoolHelper` to make sure they can only be called from the `LockZap` contract.

Update: *Resolved in pull request #173 at commits 4ba1fa9 and 6f5fd15.*

Lack of Access Control for `LiquidityZap` Contract Initialization

The `LiquidityZap` contract allows any user to initialize its configuration through the `initLiquidityZap` function. During protocol deployment, an attacker can execute `initLiquidityZap` before it is called by `UniswapPoolHelper`'s `initializePool` function. As a result, this would require the Radiant team to deploy a new instance of `LiquidityZap`.

Consider limiting access to `initLiquidityZap` by ensuring only the pool helper can call it, or by adding the `onlyOwner` modifier and calling it externally rather than from the pool helper's `initializePool` function.

Update: *Resolved in pull request #200 at commit 6ea6553. Access to the `initLiquidityZap` function is now restricted to the contract's owner, and the function is no longer invoked from `initializePool`.*

Usage of Insecure Approval Functions

Throughout the codebase there are multiple instances of insecure approvals that may lead to security issues:

`SafeERC20`'s `safeIncreaseAllowance` function which cannot handle non-standard ERC-20 token implementations such as USDT on Ethereum Mainnet, because USDT requires the allowance to be reset to zero before setting it to any positive value. If the allowance is not already zero when attempting to change the existing approval to a non-zero value, the USDT `approve` function will revert.

- `Compounder.sol` in lines 154, 172 and 274 uses the deprecated `safeApprove` function.
- `Leverager.sol` in lines 191, 194, 233, 236, 282, 285 and 386 uses the deprecated `safeApprove` function.
- `StargateBorrow.sol` in lines 204-205 uses the deprecated `safeApprove` function.
- `LockZap.sol` in lines 244, 320 and 333 uses the deprecated `safeApprove` function.
- `BalancerPoolHelper.sol` in lines 113-114 uses the deprecated `safeApprove` function.
- `UniswapPoolHelper.sol` in lines 60-61 uses the deprecated `safeApprove` function.

Consider using the `SafeERC20` contract's new `forceApprove` function instead in order to avoid running into issues when dealing with non-standard ERC-20 implementations.

Update: *Resolved in pull request #228 at commits 299d6d5, 25a9377 and 06f49a7.*

Notes & Additional Information

Inconsistent SPDX License Identifiers

Throughout the codebase, the MIT SPDX license identifier is used, except for these instances:

- `wstethOracle.sol` does not contain an SPDX license identifier.
- `PriceProvider.sol` uses the agpl-3.0 license instead of MIT.
- `LiquidityZap.sol` uses the MIT license, but the file contains documentation that indicates the original code was licensed under GPL.

Consider working with a legal team with knowledge about software licensing to resolve these inconsistencies to avoid potential legal issues regarding copyright.

Within `MultiFeeDistribution.sol`, `uint` is used instead of `uint256`.

In favor of explicitness, consider replacing all instances of `uint` with `uint256`.

Update: *Resolved in pull request #239 at commit 9488446.*

TODO Comments in the Code

The following instances of TODO comments were found in the codebase:

- The `TODO` comment on line 847 in `ChefIncentivesController.sol`
- The `TODO` comment on line 1070 in `MultiFeeDistribution.sol`

During development, having well-described TODO comments will make the process of tracking and resolving them easier. Without this information, these comments might age and important information for the security of the system might be forgotten by the time it is released to production.

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO to a corresponding backlog issue.

Update: *Resolved in pull request #263 at commit a5e28ad.*

Variables Could Be Marked `immutable`

Variables that are only assigned a value from within the `constructor` of a contract can be declared as `immutable`.

Within the codebase, there are several variables that could be marked `immutable`. For instance:

- The `lendingPool`, `eligibilityDataProvider`, `lockZap`, `cic`, `weth` and `aaveOracle` variables in the `Leverager` contract
- The `tokenV1` and `tokenV2` variables in the `Migration` contract

To better convey the intended use of variables and to potentially save gas, consider adding the `immutable` keyword to variables that are only set in the constructor.

Throughout the codebase, instances of externally-called functions with `public` visibility were found.

Some examples include:

- `initialize`, `pause`, `unpause`, `autocompoundThreshold` `isEligibleForCompound`, `userEligibleForCompound` and `selfEligibleCompound` functions in `Compounder.sol`
- `getVDebtToken` and `ltv` functions in `Leverager.sol`
- `initialize` function in `StargateBorrow.sol`
- `isMarketDisqualified` function in `EligibilityDataProvider.sol`
- `setFallback` and `enableFallback` functions in `BaseOracle.sol`
- `setLookback` function in `MultiFeeDistribution.sol`
- `getSwapFeePercentage` and `setSwapFeePercentage` functions in `BalancerPoolHelper.sol`
- `batchUpdateAllocPoint` function in `ChefIncentivesController.sol`
- `getPoolHelper` and `getVDebtToken` functions in `LockZap.sol`

To better convey the intended use of functions and to potentially realize some additional gas savings, consider changing a function's visibility from `public` to `external` if it is never used internally.

Update: *Resolved in pull request #237 at commit 026ec8a.*

Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function body for the purpose of being returned as the function's output. They are an alternative to explicit inline `return` statements.

Throughout the codebase, there are multiple instances of unused named return variables. Some examples are:

In `ChainlinkAdapter.sol`:

- The `roundId`, `answer`, `startedAt`, `updatedAt` and `answeredInRound` return variables in the `latestRoundData` function

In `ChefIncentivesController.sol`:

- The `issueBaseBounty` return variable in the `checkAndProcessEligibility` function
- The `amount` return variable in the `availableRewards` function

In `UniswapPoolHelper.sol`:

- The `optimalWETHAmount` return variable in the `quoteFromToken` function

Consider using the existing named return variables or alternatively removing them.

Update: *Resolved in pull request #251 at commit e9984e9.*

Use Custom Errors

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed versus using `require` and `revert` statements with custom error strings.

There are instances of `require` statements found in these files:

- `Leverager.sol`
- `StargateBorrow.sol`
- `BountyManager.sol`
- `BaseOracle.sol`
- `ChainlinkV3Adapter.sol`
- `PriceProvider.sol`
- `UniV2TwapOracle.sol`
- `UniV3TwapOracle.sol`
- `RadiantOFT.sol`

To improve conciseness, consistency, and gas savings, consider replacing hard-coded `require` and `revert` messages with custom errors.

Update: *Resolved in pull request #252 at commits 6e081bc, 6451a5d, and 362c4a4.*

Use of Non-Explicit Imports

The use of non-explicit imports in the codebase can decrease the clarity of the code and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Several instances where global imports are being used in the codebase were identified:

- Lines 4-8 and 10-20 of `Compounder.sol`
- Line 4 and Lines 6-10 of `ChainlinkV3Adapter.sol`
- Lines 2-5 of `wstethOracle.sol`
- Line 10 of `RadiantOFT.sol`
- Line 8 of `DustRefunder.sol`
- Lines 5-13 and Lines 15-21 of `UniswapPoolHelper.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Resolved in pull request #253 at commit c05064a.*

Redundant Use of `SafeMath` Library

The OpenZeppelin SafeMath library provides arithmetic functions with overflow/underflow protection, but Solidity `0.8.0` has added built-in overflow and underflow checking, supplanting the functionality provided by the library.

Throughout the codebase, the `SafeMath` library is being used in contracts with a Solidity version greater than `0.8.0`, resulting in the addition of redundant overflow/underflow checks.

The following contracts import the `SafeMath` library:

- `EligibilityDataProvider.sol`
- `BaseOracle.sol`
- `ChainlinkV3Adapter.sol`
- `PriceProvider.sol`
- `UniV3TwapOracle.sol`
- `ChefIncentivesController.sol`
- `MiddleFeeDistribution.sol`
- `MultiFeeDistribution.sol`
- `Migration.sol`
- `RadiantOFT.sol`
- `LockZap.sol`
- `BalancerPoolHelper.sol`
- `LiquidityZap.sol`
- `UniswapPoolHelper.sol`

Consider removing the `SafeMath` import and its associated function calls from the codebase.

Update: *Resolved in pull request #254 at commits 18a98c1 and 3c3dd90.*

Implicitly Abstract Contract

The `consult` virtual function of `BaseOracle.sol` is declared but not implemented. As a result, `BaseOracle` can never be instantiated directly.

To better signal this intention, consider explicitly marking the `BaseOracle` contract as `abstract`.

Update: *Resolved in pull request #255 at commit 1e74c37.*

Follow Solidity Style Guide

There are several occurrences in the codebase where the Solidity style guide is not followed which makes code more difficult to read and prone to errors.

In `DustRefunder.sol`:

- The function `getPairReserves` is internal and its name should start with an underscore (`_`).
- Public storage variables `_token` and `_tokenWETHPair` should start with a lowercase letter, not with an underscore (`_`).
- The `else` block within `_addLiquidity` does not have curly braces. The subsequent `if block` where RDNT token proceeds are refunded also does not have curly braces.

In `MultiFeeDistribution.sol`:

- The style guide's recommended layout of functions is not followed. Consider grouping view functions together.
- The `getBalances` function uses an underscore on its input parameter `address _user`. This is inconsistent with the other nearby functions that also accept an `address user` input parameter, such as `withdrawableBalance`.
- All initializer arguments have a leading underscore, except for `priceProvider`. At the same time, all state variables do not have leading underscores, even private mappings, except for `_priceProvider`.

In `MiddleFeeDistribution.sol`:

- The `emitNewTransferAdded` function is internal and its name should start with an underscore (`_`).
- The `aaveOracle` parameter of the initializer is the only one without a leading underscore (`_`).

In `LockerList.sol`:

- The `userlist` state variable should be renamed `userList` to follow the same naming convention as the rest of the codebase.

In `Leverager.sol`:

- The internal function name `requiredLocked` should start with an underscore (`_`).

In `BountyManager.sol`:

- Functions `getBestBounty`, `getMfdBounty`, `getChefBounty` and `getAutoCompoundBounty` are internal and their names should start with an underscore (`_`).

In `LockZap.sol`:

- The `ACCEPTABLE_RATIO` storage variable name consists of all capital letters which suggests it is a constant variable, but it can be changed via the `setAcceptableRatio` function.

In `UniswapPoolHelper.sol`:

- Consider making the value `100000000` more readable by changing it to `100_000_000`.

In `BalancerPoolHelper.sol`:

- The `computeFairReserves` function is internal and its name should start with an underscore (`_`).
- The `joinPool` function is internal and its name should start with an underscore (`_`).
- The `sortTokens` function is internal and its name should start with an underscore (`_`).

To improve the readability of the codebase, consider following the Solidity style guide.

Update: *Resolved in pull request #256 at commits 66b6ecd, 8a8abf6 and 8a8abf6.*

Non-library Contract in `libraries` Directory

The `LockerList.sol` contract is located in the `libraries` directory, but it is not defined using the `library` keyword.

To avoid confusion, consider relocating this contract to the `staking` directory.

Update: *Resolved in pull request #257 at commits 859dacd and a5fbb2f.*

For instance:

- Imports `IAToken`, `IMultiFeeDistribution`, `ILendingPoolAddressesProvider`, `ILendingPool`, and `ILockZap` of `BountyManager.sol`
- Imports `SafeMath`, `Initializable`, and `IChainlinkAggregator` of `ChainlinkV3Adapter.sol`
- Imports `IChainlinkAggregator` and `IBaseOracle` of `wstethOracle.sol`
- Import `IERC20Metadata` of `ChefIncentivesController.sol`
- Import `LockedBalance` of `MiddleFeeDistribution.sol`
- Imports `IMultiFeeDistribution`, `ILendingPool`, `IPoolHelper`, and `IERC20DetailedBytes` of `UniswapPoolHelper.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #258 at commit f6dfecb.*

Inconsistent Gas Optimizations in `for` Loops

Throughout the codebase, the `unchecked` keyword is used frequently in `for` loops to increment the loop index variable. However, the use of `unchecked` is inconsistent, with only some loops including it. For instance, in the `ChefIncentivesController` contract, there are 11 for loops, but only 2 of them use `unchecked` to increment the loops' index.

Within `ChefIncentivesController` there are also 3 instances where the counter is pre-incremented:

- Line 393 within a `for` loop
- Line 657 within an `unchecked` block
- Line 730 within a `for` loop

Consider applying the `unchecked` pattern consistently in `for` loops throughout the codebase and enforcing consistency on pre-increments vs post-increments on the counter variable, specifying the reason behind each deviation with a comment.

Throughout the codebase, there are several opportunities for gas optimizations:

- In the `MultiFeeDistribution` contract, consider moving the `penaltyAmount` and `burnAmount` calculation into the previous `if` block in order to skip the calculation when `penaltyFactor` is not computed due to tokens already being fully vested.
- Consider checking if the bridge `fee` is positive before sending it to the treasury.
- Consider checking if the looping `fee` is positive before sending it to the treasury on every looping function within the `Leverager` in order to save gas when this fee is deactivated.
- When calling `loopETH` within the `Leverager` contract, the amount borrowed in WETH is fully unwrapped into native ETH in order to send the `fee` to the treasury, and then gets wrapped back into WETH before being deposited into the lending pool. Consider just unwrapping the `fee` amount in order to skip the unnecessary wrapping operation.
- Consider overriding the `OFTCoreV2` internal `_sendAndCall` function and adding the `whenNotPaused` modifier. Additionally, consider adding the modifier to `_send` directly as well. That way the overridden `_debitFrom` can be removed entirely and gas will be saved when the contract is paused.
- In the `RadiantOFT` overridden `_send` function, consider requiring a positive amount before trying to debit that amount from the caller in order to save gas, and skip the overwriting of the `amount` variable after the fact, since it returns the same value as the input parameter.
- Consider delaying the call to `getBaseBounty` until the value of `issueBaseBounty` is known. There is no need to retrieve it when paying auto-compound bounties.
- In the `ChefIncentivesController` contract, the `_mint` function calls the `_getMfd` function twice. Consider storing the result from the first call in a local variable that can be used in place of the second call.
- Consider removing both the `whenNotPaused` and `isWhiteListed` modifiers from the `claim` function in order to avoid executing them twice, since they will be enforced directly on the `executeBounty` function.
- Consider whether it is necessary to emit the `NewTransferAdded` event within `MiddleFeeDistribution` since this is part of the auto-compounder bounty claiming flow of execution. It is important to keep this bounty gas-efficient in order to make sure hunters are incentivized to claim it.

this optimization was not already being used:

- On line 160 of `StargateBorrow.sol`
- On line 135 of `MiddleFeeDistribution.sol`
- On line 848 of `MultiFeeDistribution.sol`
- On line 1142 of `MultiFeeDistribution.sol`

To improve gas consumption, readability, and code quality, consider refactoring the code in these examples and carefully reviewing the entire codebase.

Update: *Resolved in pull request #260 at commits 9b770e2 and c2e0af6. The `whenNotPaused` modifier was also removed from the `quote` function in the `BountyManager` contract. The `NewTransferAdded` event emission was left unchanged.*

Floating Pragma

Contract `wstethOracle` uses the floating pragma `^0.8.0`. It is recommended to set the pragma to `0.8.12` to align with the protocol's practice of locking the pragma for all contracts. This precautionary measure helps prevent the accidental deployment of contracts with outdated compiler versions that could potentially introduce vulnerabilities.

Update: *Resolved in pull request #249 at commit f4e029e.*

Redundant Code

Several instances of redundant or unnecessary code were found throughout the codebase:

- The `override` keyword is often used in cases where there is no base function body being overridden. Consider eliminating the unnecessary use of `override` to avoid confusion.
- The `pragma abicoder v2` statement is redundant on the current Solidity version used to compile the project, 0.8.12. Consider whether it should stay for explicitness or alternatively, remove it from all occurrences since it is now enabled by default.

In `MiddleFeeDistribution.sol`:

In `MultiFeeDistribution.sol`:

- When all user locks are withdrawable, the function `_cleanWithdrawableLocks` overrides the values of `lockAmount` and `lockAmountWithMultiplier`, and deletes `userLocks[user]`, all of which is unnecessary since it was already computed.
- The `individualEarlyExit` function contains a line of unreachable code. The `ieeWithdrawableBalances` function returns an `index` value that will be between `0` and `userEarnings[onBehalfOf].length - 1`, so the `if` statement on line 903 can never be true. Additionally, even if this could happen, the correct behavior in this case would be to revert rather than return.

In `ChefIncentivesController.sol`:

- The `initialize` function performs an unnecessary `address()` cast on `_poolConfigurator`, which is already an address.
- The `_emitReserveLow` internal function is only called by `claim`. Consider eliminating this function by moving its code into the location where it is called.
- The `endRewardTime` function's calculation for `unclaimedRewards` duplicates the existing `availableRewards` function.
- The `_isEligible` boolean parameter passed to `stopEmissionsFor` is unnecessary since that execution point cannot be reached if `_isEligible` is `true`.

In `LockerList.sol`:

- The empty constructor is unnecessary since it will be called (and the parent `Ownable` constructor) implicitly.

In `Compounder.sol`:

- The contract uses a `<= 0` check for input validation of `_compoundFee`, both in the `initializer` and the `setCompoundFee` function. This is an unsigned value, so the value can never be less than zero.
- The `_claimAndSwapToBase` function triggers a call to `swapExactTokensForTokens` within Uniswap's router, using a deadline value of

Uniswap's router, using the deadline value of `block.timestamp + 600`. This additional 600-second value is not needed; passing `block.timestamp` is sufficient.

In `StargateBorrow.sol`:

- The constant `POOL_ID_ETH` is not used and should be removed.

In `EligibilityDataProvider.sol`:

- The storage variables `eligibleDeposits` and `userDeposits` are declared but never used.
- The `isMarketDisqualified` function should be removed since it is not used anywhere and it is not coherent with the current protocol's behavior.

In `BountyManager.sol`:

- The storage variable `bountyBooster` does not appear to be used anywhere after being set and can be removed along with the associated `setBountyBooster` function and `BountyBoosterUpdated` event.
- The storage variable `slippageLimit` does not appear to be used anywhere after being set and can be removed along with the associated `setSlippageLimit` function and `SlippageLimitUpdated` event.
- The check for positive `_amount` in `_sendBounty` is redundant since it has to be positive at this point due to this previous condition.

In `LockZap.sol`:

- The `setPriceProvider`, `setMfd`, and `setPoolHelper` functions perform unnecessary `address()` casts on their input variables, which are already addresses.

In `UniswapPoolHelper.sol`:

- The `zapTokens` function reassigns the value of `liquidity`.
- The `zapWETH` reassigns the value of `liquidity`.

- The `swap` function call to `IVault(vaultAddr).swap` uses a deadline value of `block.timestamp + 3 minutes`. The addition of 3 minutes is not needed; passing `block.timestamp` is sufficient.

In `ChainlinkV3Adapter.sol`:

- The `update` function logic which updates `ethLatestTimestamp` and `tokenLatestTimestamp` is not used since `canUpdate` returns `false`. Consider removing the `update` function and the associated storage variables `ethLatestTimestamp` and `tokenLatestTimestamp`.

In `PriceProvider.sol`:

- The storage variable `eligibilityProvider` is not used and should be removed.

Update: *Resolved in pull request #259 at commits 7fe0064, 74f889a, 855c142, e86d67c, ea37c2b and 724e290. The `isMarketDisqualified` function was removed in pull request #226 at commit 7f4a0bc.*

Inconsistent Use of `IERC20` Interfaces

Throughout the protocol, the `IERC20` and `IERC20Metadata` interfaces are frequently used to interact with ERC-20 tokens, with the latter interface only being used to access the `decimals` function of tokens.

However, within the `Compounder` contract, the `IERC20DetailedBytes` interface is used instead of `IERC20Metadata`. The end result is the same, but this approach may be error-prone for future development, since the return value types for the `name` and `symbol` functions in `IERC20DetailedBytes` are different than the ones used in `IERC20Metadata`, and there is no apparent reason to favor `IERC20DetailedBytes` in the `Compounder` contract.

Consider enforcing consistency by replacing the `IERC20DetailedBytes` interface with `IERC20Metadata` within the `Compounder` contract.

Update: *Resolved in pull request #262 at commit 659430e.*

In `Leverager.sol`:

- Consider leveraging the existing `RATIO_DIVISOR` constant on both the `constructor` and `setFeePercent` function when validating `_feePercent` instead of the hard-coded `1e4` value.
- Consider assigning the value `2 ** 16` within the `ltv` function to a constant variable or alternatively, leveraging the existing `getLtv` calculation which is more self-explanatory and gas efficient.
- Consider adding a constant value for calculating margin in `wethToZapEstimation`.
- Consider using a constant when adding a safety margin to `wethToZap`.
- Consider using a constant when choosing the desired interest rate mode instead of the hard-coded value `2` in `zapWETHWithBorrow`.

In `Compounder.sol`:

- Consider adding a `MAX_COMPOUND_FEE = 2000` constant to handle validation in the `initialize` and `setCompoundFee` functions.
- Consider adding a `MIN_SLIPPAGE_LIMIT = 8000` constant to handle validation in the `initialize` and `setSlippageLimit`.
- Consider dividing by `PERCENT_DIVISOR` and not by `10000` in `_wethToRdnt`.
- Consider adding a `MIN_DELAY = 1 days` constant to handle the check in `isEligibleForAutoCompound`.

In `StargateBorrow.sol`:

- Consider using the `FEE_PERCENT_DIVISOR` constant to validate `_xChainBorrowFeePercent` parameter in the `initialize` and `setXChainBorrowFeePercent` functions.
- Consider using a constant variable to hold the referral code used (`0`) when interacting with the lending pool in the `borrow` and `borrowETH` functions.

In `EligibilityDataProvider.sol`:

`priceToleranceRatio` during initialization.

- Consider adding a `MIN_PRICE_TOLERANCE_RATIO = 8000` constant to validate the value of `_priceToleranceRatio` in `setPriceToleranceRatio`.

In `BountyManager.sol`:

- Consider adding a `RATIO_DIVISOR = 10000` constant and use it for validating `_hunterShare` in the `initialize`, `setHunterShare`, and `executeBounty` functions.

In `UniV2TwapOracle.sol`:

- Consider adding a `PERIOD_MIN = 10` constant for validating `_period` in the `initializer` and `setPeriod` functions.

In `RadiantOFT.sol`:

- Consider using a `SHARED_DECIMALS = 8` constant for `OFTV2` constructor initialization.
- Consider using `FEE_DIVISOR` for validating the `_fee` parameter in the `setFee` function.

In `BalancerPoolHelper.sol`:

- Consider adding `RDNT_WEIGHT = 800000000000000000` and `WETH_WEIGHT = 200000000000000000` constants for setting pool weights.
- Consider adding `INITIAL_SWAP_FEE_PERCENTAGE = 1000000000000000` constant for setting `swapFeePercentage`.
- Consider dividing by the pool weights ratio instead of using the hard-coded value of `4` within `quoteFromToken`.

In `LockZap.sol`:

- Consider using `RATIO_DIVISOR` to validate `_ethLPRatio`.
- Consider adding constant variables for the values `100` and `97` in the `quoteFromToken` calculation.

To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. For complex values, consider adding an inline comment explaining how they were calculated or why they were chosen.

Update: *Resolved in pull request #240 at commits 46a113f, cf2bc80, and f98d6b2.*

Typographical Errors

Consider addressing the following typographical errors:

In `Leverager.sol`:

- Line 394: "underlyig" should be "underlying".

In `BountyManager.sol`:

- Line 52: "whiteliested" should be "whitelisted".
- Line 310: "dont" should be "don't".

In `EligibilityDataProvider.sol`:

- Line 55: "Elgible" should be "Eligible".

In `PriceProvider.sol`:

- Line 123: "heler" should be "helper".

In `UniV3TwapOracle.sol`:

- Line 33: "loopback" should be "lookback".
- Line 77: "loopback" should be "lookback".
- Line 78: "Loopback" should be "Lookback".

In `ChefIncentivesController.sol`:

- Line 136: "poitns" should be "points".
- Line 518: "eligbility" should be "eligibility".

- Line 259: "convert" should be "converter".
- Line 473: "Earnings which is locked yet" is an incomplete sentence.
- Line 628: "transfered" should be "transferred".
- Line 689: "users" should be "user's".
- Line 1046: "loopback" should be "lookback".

In `BalancerPoolHelper.sol`:

- Line 290: "ad" should be "and".

In `DustRefunder.sol`:

- Line 11: "remained" should be "remaining".

In `LiquidityZap.sol`:

- Line 186: "transfered" should be "transferred".
- Line 200: "transfered" should be "transferred".

In `LockZap.sol`:

- Line 29: "RAITO" should be "RATIO".
- Line 82: "paramter" should be "parameter".

Update: *Resolved in pull request #238 at commit d042343.*

1 critical, 5 high, and 15 medium-severity issues were found, apart from multiple lower-severity findings. Communication with the Radiant team was smooth and the team's thorough answers to our questions were helpful throughout the audit. The Radiant team started working on the higher-severity findings as soon as they were disclosed. The codebase is quite complex and will benefit from incorporating our informational notes and lower-severity recommendations in order to improve its readability and robustness. In terms of documentation, there is high-quality material available to end users, but it is still recommended to generate more technical documentation.

Appendix

Monitoring Recommendations

While audits help in identifying potential security risks, the Radiant team is encouraged to also incorporate automated monitoring of on-chain contract activity into their operations. Ongoing monitoring of deployed contracts helps in identifying potential threats and issues affecting the production environment.

- To ensure the `ChefIncentiveController` contract remains funded, consider monitoring the `ChefReserveLow` and `ChefReserveEmpty` events.
- To ensure the `BountyManager` contract remains funded, consider monitoring the `BountyReserveEmpty` event, and also periodically checking if the `balanceOf` RDNT token held by the `BountyManager` drops below some predefined low level.
- Monitor positions that meet the criteria for a bounty to make sure incentives are large enough to incentivize hunters to claim them. Otherwise, protocol health might be affected.
- Monitor transaction history for the routine occurrence of bounties, which are a necessary service required by the protocol to ensure a fair distribution of rewards. If users are not performing bounty transactions on a regular basis, this may indicate that the `maxBaseBounty` amount needs to be increased, or the bounty system is not working as intended for some other reason.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits