BlockSec

Security Audit Report for Radpie

Date: Sep 14, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.09.14 16:40:04 +08'00'

Contents

Report Manifest

Item	Description
Client	Magpie XYZ
Target	Radpie

Version History

Version	Date	Description
1.0	Sep 14, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repo of the Radpie smart contracts [1] of Magpie XYZ. Radpie is a yield optimization protocol developed built upon Radiant [2]. During this audit, our presumption is that the dependencies from Radiant, which are being adopted by Radpie, are both reliable and secure. Specifically, this audit only covers the following smart contracts:

- radiant/RadiantStaking.sol
- radiant/RadpiePoolHelper.sol
- rewards/BaseRewardPoolV2.sol
- rewards/MasterRadpie.sol
- rewards/RadpieReceiptToken.sol
- rewards/RDNTRewardManager.sol
- rewards/RDNTVestManager.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Radpie	Version 1	d603286c5ee0115914dea2f7fb8fa4381534f8ee
	Version 2	155174988137bd6078d7d35c200f6a028a254383

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always

[1] https://github.com/magpiexyz/Radpie

[2] https://github.com/radiant-capital/v2

recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **thirteen** potential issues. Besides, we also have **five** recommendations and **seven** notes.

- High Risk: 5
- Medium Risk: 6
- Low Risk: 2
- Recommendation: 5
- Note: 7

ID	Severity	Description	Category	Status
1	High	Inconsistent address parameter	Software Security	Fixed
2	High	Potential reverts in the `_refundETH` function	Software Security	Fixed
3	High	Incorrect parameter in the `_harvestDlpRewards` function	Software Security	Fixed
4	Medium	Incorrect return value of the `assetPerShare` function	Software Security	Fixed
5	Low	Potential DoS risk in the `claim` function	Software Security	Fixed
6	Low	Potential overwriting on existing `poolInfo`	Software Security	Fixed
7	High	Double-counting rewards	DeFi Security	Fixed
8	High	Incorrect `_onlyWhiteListed` modifier	DeFi Security	Fixed
9	Medium	Lack of duplicate checks for function arguments	DeFi Security	Fixed
10	Medium	Incorrect fee removal logic	DeFi Security	Fixed
11	Medium	Lack of sanity check on total fee	DeFi Security	Fixed
12	Medium	Unclaimable rewards due to rewarder modification	DeFi Security	Fixed
13	Medium	Lack of health check	DeFi Security	Fixed
14	-	Remove unused variable	Recommendation	Fixed
15	-	Remove redundant check in the `_sendRewards` function	Recommendation	Fixed
16	-	Prevent multiple native tokens	Recommendation	Fixed
17	-	Prevent accidental native token transfers	Recommendation	Fixed
18	-	Avoid incorrect assignment	Recommendation	Fixed
19	-	The protocol will not support deflation/inflation tokens	Note	-
20	-	Potential centralization risk	Note	-
21	-	Periodic invocation of `batchHarvestDlpRewards`	Note	-
22	-	Periodic invocation of `batchHarvestEntitledRDNT`	Note	-
23	-	Ensure initial TVL in `RadiantStaking` pools	Note	-
24	-	The initialization of vdToken balance	Note	-
25	-	Periodic invocation of `accrueStreamingFee`	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Inconsistent address parameter

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In Radpie, a underlying staking token is associated with a receipt token representing the share of rewards for this kind of token. The `updateFor` function in the `RDNTRewardManager` contract expects the receipt token address as its second parameter. However, in the `_harvestRewards` function of the `MasterRadpie` contract, `updateFor` is called with the `_stakingToken` address as the second parameter, which is the underlying staking token of the receipt token. This results in an inconsistency between the expected and provided arguments.

```
function updateFor(address _account, address _receipt) external {
    _updateForByReceipt(_account, _receipt);
}
```

Listing 2.1: RDNTRewardManager.sol

```
function _harvestRewards(address _stakingToken, address _account) internal {
    if (userInfo[_stakingToken][_account].amount > 0) {
        _harvestRadpie(_stakingToken, _account);
    }

    if (rdntRewardManager != address(0))
    IRDNTRewardManager(rdntRewardManager).updateFor(_account, _stakingToken);

    IBaseRewardPool rewarder = IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder);
    if (address(rewarder) != address(0)) rewarder.updateFor(_account);
}
```

Listing 2.2: MasterRadpie.sol

Impact The `updateFor` function will not update rewards properly.

Suggestion Revise the passed address parameter.

2.1.2 Potential reverts in the `_refundETH` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `_refundETH` function utilizes `send` to transfer native tokens. However, this can fail if the recipient is a proxy contract whose fallback function consumes significant gas. For example, logging the ETH receipt in a `minimalProxy` fallback may expend more than the 2300 gas stipend provided by `send`, causing an out-of-gas failure.

```
function _refundETH(address payable _dustTo, uint256 _refundAmt) internal {
  if (_refundAmt > 0) {
      bool success = _dustTo.send(_refundAmt);
      require(success, "ETH transfer failed");
  }
}
```

Listing 2.3: DustRefunder.sol

Impact Contract users using a proxy will loss their funds due to the revert in this function.

Suggestion Refund users with `WETH` when `send` returns `false`.

2.1.3 Incorrect parameter in the `_harvestDlpRewards` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `_harvestDlpRewards` function in the `RadiantStaking` contract contains an improper invocation of `_sendRewards(address(mDLP), asset, amounts[i])` (line 671). The passed `mDLP` is an uninitialized empty address that is unrelated to the reward sending process.

Additionally, the `_sendReward` function calculates `rewardLeft` as the difference between `_amount` and `_rewardToken` balances, but uses the `_asset` (which is uninitialized `mDLP` in this case) to send the `rewardLeft` amount to the owner.

```
656   function _harvestDlpRewards(bool _force) internal nonReentrant {
657       if (!_force && lastHarvestTime + harvestTimeGap > block.timestamp) return;
658       (address[] memory rewardTokens, uint256[] memory amounts) = this.claimableDlpRewards();
659       if (rewardTokens.length == 0 || amounts.length == 0) return;
660
661       lastHarvestTime = block.timestamp;
662
663       multiFeeDistributor.getReward(rewardTokens);
664
665       for (uint256 i = 0; i < rewardTokens.length; i++) {
666           if (amounts[i] == 0 || rewardTokens[i] == rdnt) continue; // skipping RDNT for now
                  since it's not rToken
667
668           address asset = IAToken(rewardTokens[i]).UNDERLYING_ASSET_ADDRESS();
669           ILendingPool(lendingPool).withdraw(asset, amounts[i], address(this));
670
671           _sendRewards(address(mDLP), asset, amounts[i]);
672       }
673   }
```

Listing 2.4: RadiantStaking.sol

```
function _sendRewards(address _asset, address _rewardToken, uint256 _amount) internal {
    ...
    // if there is somehow reward left, sent it to owner
    uint256 rewardLeft = IERC20(_rewardToken).balanceOf(address(this));
```

```
        if (rewardLeft > _amount) {
            IERC20(_asset).safeTransfer(owner(), rewardLeft - _amount);
            emit RewardFeeDustTo(_rewardToken, owner(), rewardLeft - _amount);
        }
    }
```

Listing 2.5: RadiantStaking.sol

Impact The function will not work as expected.

Suggestion Revise the parameter accordingly.

2.1.4 Incorrect return value of the `assetPerShare` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `assetPerShare` function in the `RadpieReceiptToken` contract returns the exchange rate from the receiptToken to the underlying asset. When the receiptToken is created in the `MasterRadpie` contract, the `radiantStaking` address (there is also a typo here) is set to 0. Thus, the `assetPerShare` function should return `WAD` rather than `10 ** setDecimal` in line 50, since `WAD` represents an exchange rate of 1:1 for the `MasterRadpie` contract.

```
48    function assetPerShare() external view returns(uint256) {
49        if (radiatnStaking == address(0))
50            return 10 ** setDecimal;
51
52        return IRadiantStaking(radiatnStaking).assetPerShare(underlying);
53    }
```

Listing 2.6: RadpieReceiptToken.sol

Impact Contracts that depend on the return value of `assetPerShare` may get incorrect results.

Suggestion Revise the function return value accordingly.

2.1.5 Potential DoS risk in the `claim` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `claim` function in the `RDNTVestManager` contract contains a vulnerability that may introduce DoS risk. The `vestingSchedules` mapping is used to store vesting schedule information, but elements are never removed from it after their associated vesting period expires. This can cause the mapping size to increase over time as more vesting schedules are created. The lack of vesting schedule cleanup poses a DoS risk due to the `claim` function's gas cost scaling with the number of vesting schedules.

```
    function claim() external nonReentrant {
        VestingSchedule[] storage schedules = vestingSchedules[msg.sender];
        uint256 totalClaimable;
```

```

        for (uint256 i = 0; i < schedules.length; i++) {
            VestingSchedule storage schedule = schedules[i];
            if (block.timestamp >= schedule.endTime && schedule.amount > 0) {
                totalClaimable += schedule.amount;
                schedule.amount = 0;
            }
        }

        if (totalClaimable > 0) {
            IERC20(rdntToken).safeTransfer(msg.sender, totalClaimable);
            emit RDNTClaimed(msg.sender, totalClaimable);
        }
    }
```

Listing 2.7: RDNTVestManager.sol

Impact Users may not be able to claim vested RDNT from the contract.

Suggestion Remove expired schedules from `vestingSchedules`.

2.1.6 Potential overwriting on existing `poolInfo`

Severity Low

Status Fixed In `Version 2`

Introduced by `Version 1`

Description The `setPoolInfo` function in the `RadpiePoolHelper` contract lacks a check for existing pools. Thus, the current `poolInfo` for the corresponding asset may be inadvertently overwritten.

```
    function setPoolInfo(
        address Asset,
        address rewarder,
        bool isNative,
        bool isActive
    ) external _onlyOperator {
        if (rewarder == address(0)) revert NullAddress();
        poolInfo[Asset] = PoolInfo(rewarder, isNative, isActive);
    }
```

Listing 2.8: RadpiePoolHelper.sol

Impact N/A

Suggestion Add checks for `poolInfo`.

Feedback from the Project This is intended in case we need to do necessary update/migration/fix.

2.2 DeFi Security

2.2.1 Double-counting rewards

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In Radpie, the receipt tokens represent staking shares, with rewards determined based on users' token holdings. To avoid any discrepancies in reward distribution, Radpie updates the corresponding rewards for both the sender and receiver during a receipt token transfer.

Specifically, the beforeReceiptTokenTransfer function in the MasterRadpie contract is invoked prior to a receipt token transfer. It triggers the _harvestRewards function separately for both the _from and _to addresses during the transfer of receipt tokens. However, this results in a double-counting issue with Radpie rewards when the _from and _to addresses are identical, as seen during a self-transfer of receipt tokens. In such a case, the user's unClaimedRadpie will be added with the same pending reward amount twice.

```
function beforeReceiptTokenTransfer(
    address _from,
    address _to,
    uint256 _amount
) external _onlyReceiptToken {
    address _stakingToken = receiptToStakeToken[msg.sender];
    updatePool(_stakingToken);

    if (_from != address(0)) _harvestRewards(_stakingToken, _from);

    if (_to != address(0)) _harvestRewards(_stakingToken, _to);
}
```

Listing 2.9: MasterRadpie.sol

```
function _harvestRewards(address _stakingToken, address _account) internal {
    if (userInfo[_stakingToken][_account].amount > 0) {
        _harvestRadpie(_stakingToken, _account);
    }

    if (rdntRewardManager != address(0))
    IRDNTRewardManager(rdntRewardManager).updateFor(_account, _stakingToken);

    IBaseRewardPool rewarder = IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder);
    if (address(rewarder) != address(0)) rewarder.updateFor(_account);
}
```

Listing 2.10: MasterRadpie.sol

```
function _harvestRadpie(address _stakingToken, address _account) internal {
    // Harvest Radpie
    uint256 pending = _calNewRadpie(_stakingToken, _account);
    userInfo[_stakingToken][_account].unClaimedRadpie += pending;
}
```

Listing 2.11: MasterRadpie.sol

```
function _calNewRadpie(
    address _stakingToken,
```

```
    address _account
) internal view returns (uint256) {
    UserInfo storage user = userInfo[_stakingToken][_account];
    uint256 pending = (user.amount * tokenToPoolInfo[_stakingToken].accRadpiePerShare) /
        1e12 -
        user.rewardDebt;
    return pending;
}
```

Listing 2.12: MasterRadpie.sol

Impact Users could receive extra rewards during self-transfer.

Suggestion Revise the reward updating logic accordingly.

2.2.2 Incorrect _onlyWhiteListed modifier

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description The incorrect implementation of the _onlyWhiteListed modifier in the MasterRadpie contract renders any function declared with it unexecutable, regardless of the msg.sender.

```
modifier _onlyWhiteListed() {
    if (AllocationManagers[msg.sender]) return;
    if (PoolManagers[msg.sender]) return;
    if (msg.sender == owner()) return;
    revert OnlyWhiteListedAllocaUpdator();
    _;
}
```

Listing 2.13: MasterRadpie.sol

Impact Functions decorated with this modifier becomes unusable.

Suggestion Revise the modifier.

2.2.3 Lack of duplicate checks for function arguments

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The addRegisteredReceipt function in the RDNTRewardManager contract allows the RewardQueuer role to add _receiptToken addresses to the registeredReceipts list. However, this function lacks a check to verify if _receiptToken already exists in registeredReceipts, which could lead to duplicates.

Additionally, the poolTokenList in the RadiantStaking contract is append-only. If a pool becomes inactive and the registerPool function is called again, it may also cause duplicates in poolTokenList.

```
function addRegisteredReceipt(address _receiptToken) external onlyRewardQueuer {
    registeredReceipts.push(_receiptToken);
}
```

Listing 2.14: RDNTRewardManager.sol

```
function registerPool(
    address _asset,
    address _rToken,
    address _vdToken,
    uint256 _allocPoints,
    uint256 _maxCap,
    bool _isNative,
    string memory name,
    string memory symbol
) external onlyOwner {
    if (pools[_asset].isActive != false) {
        revert PoolOccupied();
    }
    ...
    poolTokenList.push(_asset);
    ...
}
```

Listing 2.15: RadiantStaking.sol

Impact Duplicate items may affect reward calculations.

Suggestion Add checks to avoid duplicated items.

2.2.4 Incorrect fee removal logic

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `removeFee` function in the `RadiantStaking` contract is designed to remove the fee from either `radiantFeeInfos` or `rTokenFeeInfos` based on the `_isRDNTFee` parameter. However, it mistakenly removes the fee from `radiantFeeInfos` regardless of `_isRDNTFee`, leaving elements in `rTokenFeeInfos` unremovable. Furthermore, the value of the removed fee is not deducted from either `totalRTokenFee` or `totalRDNTFee`.

```
function removeFee(uint256 _index, bool _isRDNTFee) external onlyOwner {
    if (_index >= radiantFeeInfos.length) revert InvalidIndex();
    Fees[] storage feeInfos;

    if (_isRDNTFee) feeInfos = radiantFeeInfos;
    else feeInfos = rTokenFeeInfos;

    Fees memory feeToRemove = feeInfos[_index];
    if (feeToRemove.isActive) revert StillActiveFee();

    for (uint256 i = _index; i < radiantFeeInfos.length - 1; i++) {
        radiantFeeInfos[i] = radiantFeeInfos[i + 1];
    }
```

```

    radiantFeeInfos.pop();
    emit RemoveFee(feeToRemove.value, feeToRemove.to, feeToRemove.isAddress);
}
```

Listing 2.16: RadiantStaking.sol

Impact Elements in `radiantFeeInfos` might be mistakenly removed, whereas elements in `rTokenFeeInfos` cannot be removed at all.

Suggestion Revise the `removeFee` function.

2.2.5 Lack of sanity check on total fee

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `addFee` function of the `RadiantStaking` contract, a sanity check exists to ensure that the value of the added fee must not exceed `DENOMINATOR`. However, it should also validate that the total fee, represented by `totalRDNTFee` or `totalRTokenFee`, does not surpass `DENOMINATOR` either. This total fee sanity check is currently absent in both the `addFee` and `setFee` functions. Moreover, due to the limitation that fees can only be appended and not removed within the `removeFee` function (refer to Issue 2.2.4), the total fee exceeding `DENOMINATOR` becomes a more probable scenario.

```
function addFee(
      uint256 _value,
      address _to,
      bool _isForRDNT,
      bool _isAddress
  ) external onlyOwner {
      if (_value > DENOMINATOR) revert InvalidFee();

      if (_isForRDNT) {
          radiantFeeInfos.push(
              Fees({ value: _value, to: _to, isAddress: _isAddress, isActive: true })
          );
          totalRDNTFee += _value;
      } else {
          rTokenFeeInfos.push(
              Fees({ value: _value, to: _to, isAddress: _isAddress, isActive: true })
          );
          totalRTokenFee += _value;
      }

      emit AddFee(_to, _value, _isForRDNT, _isAddress);
  }
```

Listing 2.17: RadiantStaking.sol

```
  function setFee(
      uint256 _index,
```

```
        uint256 _value,
        address _to,
        bool _isRDNTFee,
        bool _isAddress,
        bool _isActive
    ) external onlyOwner {
        if (_value > DENOMINATOR) revert InvalidFee();
        if (_index >= radiantFeeInfos.length) revert InvalidIndex();

        Fees[] storage feeInfo;
        if (_isRDNTFee) feeInfo = radiantFeeInfos;
        else feeInfo = rTokenFeeInfos;

        Fees storage fee = feeInfo[_index];
        fee.to = _to;
        fee.isAddress = _isAddress;
        fee.isActive = _isActive;
        totalRDNTFee = totalRDNTFee - fee.value + _value;
        fee.value = _value;

        emit SetFee(_to, _value);
    }
```

Listing 2.18: RadiantStaking.sol

Impact If the fee exceeds the limit, the reward distribution will revert due to insufficient balances in the contract.

Suggestion Add sanity checks for total fee.

2.2.6 Unclaimable rewards due to rewarder modification

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `set` function in the `MasterRadpie` contract provides the functionality to modify the rewarder for a pool. However, users cannot retrieve unclaimed rewards from the previous rewarder after a modification. Because the previous rewarder information vanishes in the `MasterRadpie` and the `getReward`/`getRewards` functions in the `BaseRewardPoolV2` contract (rewarder contract) can only be called by `MasterRadpie`.

```
function set(
    address _stakingToken,
    uint256 _allocPoint,
    address _rewarder
) external _onlyPoolManager {
    if (!Address.isContract(address(_rewarder)) && address(_rewarder) != address(0))
        revert MustBeContractOrZero();

    if (!tokenToPoolInfo[_stakingToken].isActive) revert OnlyActivePool();

```

```
    massUpdatePools();

    totalAllocPoint = totalAllocPoint - tokenToPoolInfo[_stakingToken].allocPoint + _allocPoint;

    tokenToPoolInfo[_stakingToken].allocPoint = _allocPoint;
    tokenToPoolInfo[_stakingToken].rewarder = _rewarder;

    emit Set(
        _stakingToken,
        _allocPoint,
        IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder)
    );
}
```

Listing 2.19: MasterRadpie.sol

```
function getReward(address _account, address _receiver)
    public
    onlyMasterPenpie
    updateReward(_account)
    returns (bool)
{
    uint256 length = rewardTokens.length;

    for (uint256 index = 0; index < length; ++index) {
        address rewardToken = rewardTokens[index];
        uint256 reward = userInfos[rewardToken][_account].userRewards; // updated during updateReward modifier
        if (reward > 0) {
            _sendReward(rewardToken, _account, _receiver, reward);
        }
    }

    return true;
}
```

Listing 2.20: BaseRewardPoolV2.sol

```
modifier onlyMasterPenpie() {
  if (msg.sender != operator)
      revert OnlyMasterPenpie();
  _;
}
```

Listing 2.21: BaseRewardPoolV2.sol

Impact The accrued rewards from a previous rewarder are inaccessible.

Suggestion Revise the contraints on `getReward` and `getRewards` function caller.

2.2.7 Lack of health check

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description In the RadiantStaking contract, the depositAssetFor function leverages user deposits automatically based on the current contract position. However, the health factor of the position is not checked. A fully leveraged position can become risky when the borrow interest rate exceeds the deposit interest rate.

```
function depositAssetFor(
    address _asset,
    address _for,
    uint256 _assetAmount
) external payable whenNotPaused _onlyActivePoolHelper(_asset) {
    Pool storage poolInfo = pools[_asset];

    // we need to calculate share before changing r, vd Token balance
    uint256 shares = _assetAmount * WAD / this.assetPerShare(_asset);
    // only direct deposit should be considered for max cap
    if (poolInfo.maxCap != 0 && IERC20(poolInfo.receiptToken).totalSupply() + shares > poolInfo
        .maxCap) revert ExceedsMaxCap();

    uint256 rTokenPrevBal = IERC20(poolInfo.rToken).balanceOf(address(this));
    _depositHelper(_asset, poolInfo.vdToken, _assetAmount, poolInfo.isNative, false);
    uint256 vdTokenBal = IERC20(poolInfo.vdToken).balanceOf(address(this));

    if (rTokenPrevBal != 0) {
        // calculate target vd balance to start looping, target vd is calculated based on
            health factor for this asset should be consistent before and after looping
        uint256 targetVD = ((vdTokenBal * _assetAmount) / (rTokenPrevBal - vdTokenBal));
        targetVD += vdTokenBal;
        (address[] memory _assetToLoop, uint256[] memory _targetVDs) = _loopData(_asset,
            targetVD);

        _loop(_assetToLoop, _targetVDs);
    }

    IMintableERC20(poolInfo.receiptToken).mint(_for, shares);

    emit NewAssetDeposit(_for, _asset, _assetAmount, poolInfo.receiptToken, shares);
}
```

Listing 2.22: RadiantStaking.sol

Impact Fully leveraged position can be liquidated and cause financial losses.

Suggestion Add checks for health factor.

2.3 Additional Recommendation

2.3.1 Remove unused variable

Status Fixed in Version 2

Introduced by Version 1

Description Since rewardToken is already used as the key for the rewards mapping, The rewardToken within the Reward structure is redundant and can be removed.

```
struct Reward {
    address rewardToken;
    uint256 rewardPerTokenStored;
    uint256 queuedRewards;
}
...
mapping(address => Reward) public rewards;                    // [rewardToken]
```

Listing 2.23: BaseRewardPoolV2.sol

Suggestion Remove the unused variable.

2.3.2 Remove redundant check in the _sendRewards function

Status Fixed in Version 2

Introduced by Version 1

Description Both _rewardToken and _amount have been validated in the _harvestDlpRewards function, thus eliminating the need for redundant checks in the _sendRewards function.

```
if (amounts[i] == 0 || rewardTokens[i] == rdnt) continue; // skipping RDNT for now since it's
    not rToken

address asset = IAToken(rewardTokens[i]).UNDERLYING_ASSET_ADDRESS();
ILendingPool(lendingPool).withdraw(asset, amounts[i], address(this));

_sendRewards(address(mDLP), asset, amounts[i]);
```

Listing 2.24: RadiantStaking.sol

```
function _sendRewards(address _asset, address _rewardToken, uint256 _amount) internal {
    if (_amount == 0) return;
    Fees[] storage feeInfos;

    if (_rewardToken == address(rdnt)) feeInfos = radiantFeeInfos;
    else feeInfos = rTokenFeeInfos;
    ...
}
```

Listing 2.25: RadiantStaking.sol

Suggestion Remove the redundant check.

2.3.3 Prevent multiple native tokens

Status Fixed in Version 2

Introduced by Version 1

Description It is recommended to include a check in the setPoolInfo function to prevent multiple native tokens from being added to the poolInfo mapping.

```
function setPoolInfo(
    address Asset,
    address rewarder,
    bool isNative,
    bool isActive
) external _onlyOperator {
    if (rewarder == address(0)) revert NullAddress();
    poolInfo[Asset] = PoolInfo(rewarder, isNative, isActive);
}
```

Listing 2.26: RadpiePoolHelper.sol

Suggestion Add checks for multiple native tokens.

2.3.4 Prevent accidental native token transfers

Status Fixed in Version 2

Introduced by Version 1

Description In the depositAsset function, it is recommended to include a an additional check in the else branch (line 101) to prevent accidental transfers of native tokens from depositors.

```
96  function depositAsset(address _asset, uint256 _amount) external payable onlyActivePool(_asset)
     {
97      if (poolInfo[_asset].isNative) {
98          if (msg.value == 0) revert InvalidAmount();
99          uint256 _amt = msg.value;
100         _depositAssetNative(_asset, msg.sender, _amt);
101     } else {
102         if (_amount == 0) revert InvalidAmount();
103         _depositAsset(_asset, msg.sender, _amount);
104     }
105 }  poolInfo[Asset] = PoolInfo(rewarder, isNative, isActive);
```

Listing 2.27: RadpiePoolHelper.sol

Suggestion Add checks to prevent accidental transfers.

2.3.5 Avoid incorrect assignment

Status Fixed in Version 2

Introduced by Version 1

Description In the updateEmissionRate function, oldEmissionRate is supposed to record the previous value of radpiePerSec before updating it. Therefore, oldEmissionRate should be assigned with radpiePerSec rather than _radpiePerSec (line 759).

```
757 function updateEmissionRate(uint256 _radpiePerSec) public onlyOwner {
758     massUpdatePools();
759     uint256 oldEmissionRate = _radpiePerSec;
760     radpiePerSec = _radpiePerSec;
761
```

```
        emit UpdateEmissionRate(msg.sender, oldEmissionRate, radpiePerSec);
    }
```

Listing 2.28: MasterRadpie.sol

Suggestion Revise the assignment accordingly.

2.4 Note

2.4.1 The protocol will not support deflation/inflation tokens

Description The `MasterRadpie` contract mints or burns receipt tokens at a 1:1 ratio based on the specified deposited or withdrawn amounts. However, if `_stakingToken` is a deflationary or inflationary token, the actual transferred amount in the `deposit` function will diverge from the specified amount. To avoid potential side effects, the protocol should not support such tokens.

```
    function deposit(address _stakingToken, uint256 _amount) external whenNotPaused nonReentrant {
        PoolInfo storage pool = tokenToPoolInfo[_stakingToken];
        IMintableERC20(pool.receiptToken).mint(msg.sender, _amount);

        IERC20(pool.stakingToken).safeTransferFrom(address(msg.sender), address(this), _amount);
        emit Deposit(msg.sender, _stakingToken, pool.receiptToken, _amount);
    }
```

Listing 2.29: MasterRadpie.sol

2.4.2 Potential centralization risk

Description The privileged `withdrawRDNT` function in the `RDNTVestManager` contract enables the owner to withdraw all RDNT rewards. This introduces a centralization risk since the owner may potentially cause losses to users.

```
    function withdrawRDNT(uint256 _amount) external onlyOwner {
        require(_amount > 0, "Amount must be greater than zero");
        IERC20(rdntToken).transfer(msg.sender, _amount);
    }
```

Listing 2.30: RDNTVestManager.sol

In `Version 2`, the `IMasterRapie` interface adds a `emergencyWithdraw` function. This privileged function can call the `emergencyWithdraw` function in the `BaseRewardPoolV3` contract to withdraw all reward tokens, introducing a centralization risk. The `MasterRadpie` contract currently does not implement this `emergencyWithdraw` function.

```
    function emergencyWithdraw(address _stakingToken, address sender) external;
```

Listing 2.31: IMasterRadpie.sol

```
    function emergencyWithdraw(address _rewardToken, address _to) external onlyMasterRadpie {
        uint256 amount = IERC20(_rewardToken).balanceOf(address(this));
        IERC20(_rewardToken).safeTransfer(_to, amount);
```

```
        emit EmergencyWithdrawn(_to, amount);
    }
```

Listing 2.32: BaseRewardPoolV3.sol

Feedback from the Project Radpie will use multisig.

For the update in `Version 2`: This is intended for now, since emergent withdraw usually involves migration, which is not needed now.

2.4.3 Periodic invocation of `batchHarvestDlpRewards`

Description The `batchHarvestDlpRewards` function in the `RadiantStaking` contract can be invoked by anyone to collect rewards from Radiant and queue them to rewarders. Before each `receiptToken` transfer, the rewarders are invoked to update the claimable rewards of users based on their current `receiptToken` balances (shares). This leads to a potential flashloan attack where an attacker can temporarily inflate shares to manipulate rewards.

Specifically, the attacker could take the following steps to launch the attack:

- Borrow a significant amount of funds through a flashloan.
- Deposit into `RadiantStaking` and acquire a substantial quantity of receipt tokens.
- Invoke the `batchHarvestDlpRewards` function.
- Withdraw from `RadiantStaking`. This invokes the `_harvestRewards` function which unfairly updates the attacker's claimable rewards due to their temporarily inflated shares.
- Repay the flashloan.

To mitigate potential loss, the protocol promises the periodic invocation of `batchHarvestDlpRewards` to prevent the accumulation of excessive rewards. This limits the amount of manipulable rewards to attackers, making an attack unprofitable.

```
    function batchHarvestDlpRewards() external whenNotPaused {
        _harvestDlpRewards(true);
    }
```

Listing 2.33: RadiantStaking.sol

```
    function _harvestRewards(address _stakingToken, address _account) internal {
        if (userInfo[_stakingToken][_account].amount > 0) {
            _harvestRadpie(_stakingToken, _account);
        }

        if (rdntRewardManager != address(0))
        IRDNTRewardManager(rdntRewardManager).updateFor(_account, _stakingToken);

        IBaseRewardPool rewarder = IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder);
        if (address(rewarder) != address(0)) rewarder.updateFor(_account);
    }
```

Listing 2.34: MasterRadpie.sol

Feedback from the Project HarvestDlpRewards are reward only for mDLP pools, I'm not too concerned about sandwich attack:

1. _harvestDlpRewards is a bit gas intense operation.

2. Hacked will have to convert dlp -> mDlp and stake to get reward, but converting mDlp -> dlp will for sure get a significant discount (like around 20%), so it's not economically beneficial to do sandwich attack.

3. Currently, we have cronjob to harvest 1 time every other 3 days, which won't cause much reward accumulated but not harvested.

2.4.4 Periodic invocation of `batchHarvestEntitledRDNT`

Description The `batchHarvestEntitledRDNT` function in the `RadiantStaking` contract can be invoked by anyone to distribute claiamble RDNT across different pools. Before updating, weights are calculated in the `entitledRdntGauge` function to determine the reward proportions each pool is entitled to. Specifically, the weight of each pool is determined by the proportion of `rToken` and `vdToken` held by the staking contract in that pool relative to the total supply. This leads to a potential unfair reward distribution where an attacker inflate the rToken total supply, thus manipulating a specified pool's weight.

An attacker could take the following steps to launch the attack:

- Mint substantial rTokens in Radiant to artificially inflate the total `rToken` supply.
- Invoke the `batchHarvestEntitledRDNT` function to update rewards based on the manipulated weight.
- Profit from the unfair reward distribution.

Additionally, the `batchHarvestEntitledRDNT` function updates `entitledPerTokenStored` of each pool, but do not update `userInfos`. The `userInfos` stores each user's entitled RDNT (presented by `userEntitled`) and `entitledPerTokenStored` in last `userEntitled` updates (presented by `userEntitledPerTokenPaid`). Only when the `_updateForByReceipt` function is invoked and `entitledPerTokenStored` is updated, will users' entitled RDNT be updated. Therefore, an attacker can take the following steps to temporarily inflate shares and manipulate entitled RDNT:

- Mint substantial receipt tokens in `RadiantStaking`.
- Invoke the `batchHarvestEntitledRDNT` function to update `entitledPerTokenStored`.
- Invoke the `vestRDNT` function in `RDNTRewardManager` to update entitled RDNT and start vesting. Since `entitledPerTokenStored` is changed, the `_updateForByReceipt` function automatically update `userEntitled` based on the inflated shares, presenting inflated entitled RDNT rewards. The vesting is then scheduled via the `scheduleVesting` function and rewards can be claimed after `vestedTime`.
- Burn the receipt tokens and withdraw the assets.

To mitigate potential loss, the protocol promises the periodic invocation of `batchHarvestEntitledRDNT` to prevent the accumulation of excessive rewards. This limits the amount of manipulable rewards to attackers, making an attack unprofitable.

```
function batchHarvestEntitledRDNT(bool _force) external whenNotPaused {
    (uint256 totalWeight, ,uint256[] memory weights) = this.entitledRdntGauge();
    ...
    uint256 updatedClamable = chefIncentivesController.userBaseClaimable(address(this));

    for (uint256 i = 0; i < poolTokenList.length; i++) {
        Pool storage poolInfo = pools[poolTokenList[i]];
        /// diff of current updated userBaseClaimable and previosly seen userBaseClaimable is
            the new RDNT emitted for Radpie.
        uint256 toEntitled = (updatedClamable - lastSeenClaimableRDNT) * weights[i] /
            totalWeight;
```

```
        if (toEntitled > 0) _enqueueEntitledRDNT(poolInfo.receiptToken, toEntitled);
    }

    lastSeenClaimableTime = block.timestamp;
    lastSeenClaimableRDNT = updatedClamable;
}
```

Listing 2.35: RadiantStaking.sol

```
function entitledRdntGauge() external view returns(uint256 totalWeight, address[] memory assets, uint256[] memory weights) {
    uint256 length = poolTokenList.length;
    assets = new address[](length);
    weights = new uint256[](length);

    for (uint256 i = 0; i < poolTokenList.length; i++) {
        Pool storage poolInfo = pools[poolTokenList[i]];
        assets[i] = poolTokenList[i];

        if (!poolInfo.isActive) continue;

        uint256 rTokenBal = IERC20(poolInfo.rToken).balanceOf(address(this));
        uint256 vdTokenBal = IERC20(poolInfo.vdToken).balanceOf(address(this));

        (uint256 rTokenTotalSup, uint256 rAlloc,,,) = chefIncentivesController.poolInfo(poolInfo.rToken);
        (uint256 vdTokenTotalSup, uint256 vdAlloc,,,) = chefIncentivesController.poolInfo(poolInfo.vdToken);

        uint256 weight = 10 ** 12 * rTokenBal * rAlloc / rTokenTotalSup + 10 ** 12 * vdTokenBal * vdAlloc / vdTokenTotalSup;
        weights[i] = weight;
        totalWeight += weight;
    }
}
```

Listing 2.36: RadiantStaking.sol

```
function _updateForByReceipt(address _account, address _receipt) internal {
    UserInfo storage userInfo = userInfos[_receipt][_account];
    RDNTRewardStats storage rewardStat = rdntRewardStats[_receipt];

    if (userInfo.userEntitledPerTokenPaid == rewardStat.entitledPerTokenStored) return;

    userInfo.userEntitled = entitledRDNTByReceipt(_account, _receipt);
    userInfo.userEntitledPerTokenPaid = rewardStat.entitledPerTokenStored;

    emit EntitledRDNTUpdated(
        _account,
        _receipt,
        userInfo.userEntitled,
        userInfo.userEntitledPerTokenPaid
```

```
        );
    }
```

Listing 2.37: RDNTRewardManager.sol

Feedback from the Project Since batchHarvestEntitledRDNT might be intense gas consumption, that's why we are currently not adding this upon deposit/Withdraw Asset, but we're leaving a force argument to see if on mainnet, the gas is ok, we might add batchHarvestEntitledRDNT in deposit/Withdraw Asset with false for force argument.

But currently, there will be a cap for each pool, so technically an attacker won't be able to deposit a large amount into Radpie and withdraw. Also, I think the pool might be cap full most of the time due to the constraints of RDNT eligibility.

The batchHarvestEntitledRDNT now is designed to be called at most 1 time every other 3 days, if gas fee allowed, we might do at a higher frequency. (Other provide incentives so community can help to harvest ensure seamless reward distribution)

2.4.5 Ensure initial TVL in `RadiantStaking` pools

Description The `depositAssetFor` function in the `RadiantStaking` contract is vulnerable to an inflation attack. It calculates the minted shares using:

$$shares = \frac{\_assetAmount * WAD}{asserPerShare(\_asset)}$$

The `asserPerShare` function returns `WAD` when `receiptTokenTotal` or `rTokenBal` equals 0. Otherwise, the `pricePerShare` of an asset is calculated as:

$$pricePerShare = \frac{(rTokenBal - vdTokenBal) * WAD}{receiptTokenTotal}$$

Additionally, the `pricePerShare` carries `WAD`, leading to potential precision loss when `_assetAmount` is converted to `shares`. The `shares` could be rounded down to zero. The `depositAssetFor` function does not validate that minted shares is greater than zero, resulting in a potential inflation attack:

- Initially, an attacker mints 1 receipt token with 1 rToken.
- The attacker transfers a substantial amount of rTokens to `RadiantStaking`, inflating `assetPerShare`.
- Another user deposits rTokens into `RadiantStaking`. Due to the manipulated `assetPerShare`, the user deposits rTokens but receives 0 receipt token. This further inflates `assetPerShare`.
- With 1 receipt token, the attacker can withdraw all rTokens from the pool to gain profits.

To imitate such inflation attacks, the protocol promises there must be a certain amount of initial total value locked (TVL) before any deposits can occur. This ensures that `assetPerShare` cannot be arbitrarily inflated before any legitimate users interact with the contract.

```
    function depositAssetFor(
        address _asset,
        address _for,
        uint256 _assetAmount
    ) external payable whenNotPaused _onlyActivePoolHelper(_asset) {
        Pool storage poolInfo = pools[_asset];

        // we need to calculate share before changing r, vd Token balance
        uint256 shares = _assetAmount * WAD / this.assetPerShare(_asset);
        ...
```

```
}
```

Listing 2.38: RadiantStaking.sol

```
function assetPerShare(address _asset) external view returns (uint256) {
    Pool storage poolInfo = pools[_asset];

    uint256 reciptTokenTotal = IERC20(poolInfo.receiptToken).totalSupply();
    uint256 rTokenBal = IERC20(poolInfo.rToken).balanceOf(address(this));
    if (reciptTokenTotal == 0 || rTokenBal == 0) return WAD;

    uint256 vdTokenBal = IERC20(poolInfo.vdToken).balanceOf(address(this));

    return (rTokenBal - vdTokenBal) * WAD / reciptTokenTotal;
}
```

Listing 2.39: RadiantStaking.sol

Feedback from the Project We will make sure the core team supplies the initial TVL with a certain amount of TVL.

2.4.6 The initialization of vdToken balance

Description For a new pool in the `RadiantStaking` contract, the owner must manually call the `loop` function before any deposits to properly initialize the vdToken balance.

This is because the `depositAssetFor` function calculates `targetVD` and leverage borrows on Radiant until vdTokens reaches this `targetVD` amount or cannot borrow further. The `targetVD` is calculated as:

$$targetVD = \frac{vdTokenBal * \_assetAmount}{rTokenPrevBal - vdTokenBal}$$

On the first deposit, `rTokenPrevBal` is 0, so `targetVD` will not be calculated. On subsequent deposits, `targetVD` stays 0 as long as `vdTokenBal` is 0.

Without an initial vdToken balance, the `depositAssetFor` function cannot properly leverage borrow for each deposit. Accordingly, the owner must invoke `loop` to set the initial vdToken balance and ensure `depositAssetFor` functions properly.

Feedback from the Project Yes, this is intended behavior. Leverage position must be started by admin since looping upon depositAssetFor assumes health factor of that asset should not changed, that's why we need admin do initialize vdToken balance to secure health factor

2.4.7 Periodic invocation of `accrueStreamingFee`

Description In `Version 2`, Radpie introduces a new streaming fee mechanism. The `accrueStreamingFee` function in the `RadiantStaking` contract mints receipt tokens as a management fee to the owner. As a result, users may receive fewer rewards because the total supply of receipt tokens increases from the minted streaming fees. However, this impact can be minimized if the accumulated fees do not become excessive. Therefore, Radpie needs to ensure the periodic invocation of the `accrueStreamingFee` function to prevent excessive fee accumulation.

```
function accrueStreamingFee(address _receiptToken) external nonReentrant onlyOwner {
    uint256 feeQuantity;

    if (IRewardDistributor(rewardDistributor).streamingFeePercentage(_receiptToken) > 0) {
        uint256 inflationFeePercentage = IRewardDistributor(rewardDistributor).
            getCalculatedStreamingFeePercentage(_receiptToken);
        feeQuantity = IRewardDistributor(rewardDistributor).calculateStreamingFeeInflation(
            _receiptToken, inflationFeePercentage);
        IMintableERC20(_receiptToken).mint(owner(), feeQuantity);
    }

    IRewardDistributor(rewardDistributor).updatelastStreamingLastFeeTimestamp(_receiptToken,
        block.timestamp);

    emit StreamingFeeActualized(_receiptToken, feeQuantity);
}
```

Listing 2.40: RadiantStaking.sol

Feedback from the Project Yes, we're looking at 0.5% 1% streaming fee, depending on the pool's APR performance (USDC might be higher while wETH, wBTC might be lower), so the fee should not go too large.

The fee receiver can withdraw so that it no longer take too much reward away from user, but I don't think the minted receipt token amount won't go too much due to streaming fee.

Yes, we need periodic invocation, and we're looking like once every other 2 weeks or even longer (at most a month I think).