SMART CONTRACT AUDIT REPORT

for

Radpie Protocol

Prepared By: Xiaomi Huang

PeckShield

August 26, 2023

Document Properties

Client	Magpie
Title	Smart Contract Audit Report
Target	Radpie Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 26, 2023	Xuxian Jiang	Final Release
1.0-rc1	August 24, 2023	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Radpie` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Radpie

`Radpie` is a `SubDAO` designed to maximize yield and streamline governance for `Radiant` users. It leverages the strong `Radiant` infrastructure for superior benefits. The platform's core mechanism involves the locking of `dLP` tokens, which fortifies governance rights and triggers `RDNT` distribution for deposits and borrows in the `Radiant` ecosystem. `Radpie` enables `Radiant` users and `dLP` holders to access `RDNT` rewards and increased revenue without any lock-up period. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Radpie Protocol

Item	Description
Issuer	Magpie
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 26, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in the audit.

- https://github.com/magpiexyz/Radpie.git (d603286)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/magpiexyz/Radpie.git (0a21c16)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Radpie` protocol implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	3	■ ■ ■
Informational	0	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Radpie Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect setFee()/removeFee() Logic in RadiantStaking	Business Logic	Resolved
PVE-002	High	Nonfunctional _onlyWhiteListed Modifier in MasterRadpie	Business Logic	Resolved
PVE-003	Low	Improved Ether Transfer With Necessary Reentrancy Guard	Coding Practices	Resolved
PVE-004	Medium	Incorrect Reward-Sending Logic in RadiantStaking	Business Logic	Resolved
PVE-005	Low	Possible Costly Share From Improper Liquidity Initialization	Time And State	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect setFee()/removeFee() Logic in RadiantStaking

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: RadiantStaking
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The RadiantStaking contract is the main contract that enables users zap into DLP positions to get boosted yield and vote. The contract has two arrays radiantFeeInfos and rTokenFeeInfos to manage the reward fee and recipients. While examining the current logic to manage these two arrays, we notice the implementation does not follow the intended logic.

To elaborate, we show below the implementation of two related routines setFee() and removeFee(). As the names indicate, the first routine is used to update the fee configuration for a given entry while the second one removes a specific fee entry. It comes to our attention that the first routine needs to be revised to update the totalRDNTFee value if the input _isRDNTFee is true. Otherwise, it should be the totalRTokenFee value. Also the given fee entry should be validated against feeInfo.length if _isRDNTFee is false.

```
    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);
    }

    /// @dev remove some fee
    /// @param _index the index of the fee in the fee list
    /// @param _isRDNTFee true if the fee is for RDNT, false if it is for rToken
    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 3.1: `RadiantStaking::setFee()and removeFee()`

Similarly, in the second `removeFee()` routine, the given fee entry should be validated against `feeInfo.length` if `_isRDNTFee` is false. And the fee removal should be applied to the `feeInfos` array, instead of the `radiantFeeInfos`.

Recommendation Revisit the above two routines to properly update the fee entry.

Status The issue has been fixed by the following PR: `16`.

3.2 Nonfunctional _onlyWhiteListed Modifier in MasterRadpie

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `MasterRadpie`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The `Radpie` protocol has a key`MasterRadpie` contract to manage all reward pools. In particular, each reward pool may be updated with an admin routine, i.e., `updatePoolsAlloc()`. Our analysis shows that this admin routine has a flawed modifier that needs to be fixed.

To elaborate, we show below the related code snippet from the `updatePoolsAlloc()` routine as well as this specific `_onlyWhiteListed` modifier. Our analysis shows that the modifier does not work as expected. Specifically, if the caller is indeed authorized to update the allocation of reward pools, the function body will be simply skipped. In other words, there is no admin routine to update the allocation of current reward pools. Fortunately, it does not result in any fund loss.

```
    function updatePoolsAlloc(
        address[] calldata _stakingTokens,
        uint256[] calldata _allocPoints
    ) external _onlyWhiteListed {
        massUpdatePools();

        if (_stakingTokens.length != _allocPoints.length) revert LengthMismatch();

        for (uint256 i = 0; i < _stakingTokens.length; i++) {
            uint256 oldAllocPoint = tokenToPoolInfo[_stakingTokens[i]].allocPoint;

            totalAllocPoint = totalAllocPoint - oldAllocPoint + _allocPoints[i];

            tokenToPoolInfo[_stakingTokens[i]].allocPoint = _allocPoints[i];

            emit UpdatePoolAlloc(_stakingTokens[i], oldAllocPoint, _allocPoints[i]);
        }
    }
```

Listing 3.2: MasterRadpie::updatePoolsAlloc()

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

```
    }
```

Listing 3.3: MasterRadpie::_onlyWhiteListed()

Recommendation Revise the above logic to properly update the allocation of current reward pools.

Status The issue has been fixed by the following PR: 16.

3.3 Improved Ether Transfer With Necessary Reentrancy Guard

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DustRefunder
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

Description

The Radpie protocol has a DustRefunder contract that provides a convenient approach to refund dust tokens. Specifically, the native ETH token is transferred to the recipient by calling the internal _refundETH() routine. While reviewing the implementation of this routine, we notice that the ETH transfer may possibly fail because of the Out-Of-Gas (OOG) issue.

To elaborate, we show below the code snippet of the _refundETH() routine, which is called from the refundDust() routine to transfer ETH to its claimer. As we can see the _refundETH() routine directly calls the native send() routine (line 28) to transfer ETH. However, it comes to our attention that the send() is not recommended to use any more since the EIP-1884 may increase the gas cost and the 2300 gas limit may be exceeded. Check the following blog stop-using-soliditys-transfer-now for the detail why the send() is not recommended any more.

As a result, the send() may return false and the ETH might be locked in the contract. Based on this, we suggest to use call() directly with value attached to transfer ETH.

```
26    function _refundETH(address payable _dustTo, uint256 _refundAmt) internal {
27        if (_refundAmt > 0) {
28            bool success = _dustTo.send(_refundAmt);
29            require(success, "ETH transfer failed");
30        }
31    }
```

Listing 3.4: DustRefunder::_refundETH()

Recommendation Revisit the _refundETH() routine to transfer ETH using call().

Status The issue has been fixed by the following PR: 16.

3.4 Incorrect Reward-Sending Logic in RadiantStaking

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RadiantStaking`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Radpie` protocol has a key `RadiantStaking` contract that enables users zap into `DLP` positions to get boosted yield and vote. In the process of examining the current logic of sending out rewards, we notice the implementation has an issue that needs to be fixed.

To elaborate, we show below the implementation of the related `_sendRewards()` routine. This routine has a straightforward logic in sending out rewards to intended recipients. Note that the leftover funds, if any, is sent to the protocol owner. However, it comes to our attention that the leftover funds are sent with the first argument `_asset`, instead of `_rewardToken`. Also, the leftover amount is currently computed as `rewardLeft - _amount` (line 706), instead of `rewardLeft`.

```
678     function _sendRewards(address _asset, address _rewardToken, uint256 _amount)
            internal {
679         if (_amount == 0) return;
680         Fees[] storage feeInfos;
681 
682         if (_rewardToken == address(rdnt)) feeInfos = radiantFeeInfos;
683         else feeInfos = rTokenFeeInfos;
684 
685         for (uint256 i = 0; i < feeInfos.length; i++) {
686             Fees storage feeInfo = feeInfos[i];
687             if (!feeInfo.isActive) continue;
688 
689             address rewardToken = _rewardToken;
690             uint256 feeAmount = (_amount * feeInfo.value) / DENOMINATOR;
691             uint256 feeTosend = feeAmount;
692 
693             if (!feeInfo.isAddress) {
694                 IERC20(rewardToken).safeApprove(feeInfo.to, feeTosend);
695                 IBaseRewardPool(feeInfo.to).queueNewRewards(feeTosend, rewardToken);
696             } else {
697                 IERC20(rewardToken).safeTransfer(feeInfo.to, feeTosend);
698             }
699 
700             emit RewardPaidTo(_asset, feeInfo.to, rewardToken, feeTosend);
701         }
702 
703         // if there is somehow reward left, sent it to owner
704         uint256 rewardLeft = IERC20(_rewardToken).balanceOf(address(this));
```

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

Listing 3.5: `RadiantStaking::_sendRewards()`

Recommendation Revise the above routine to properly send out rewards.

Status The issue has been fixed by the following PR: `16`.

3.5 Possible Costly Share From Improper Liquidity Initialization

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RadiantStaking`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `RadiantStaking` contract acts as a vault that accepts user deposit and mints pool share in return. While examining the share calculation with the given deposit, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `depositAssetFor()` routine, which is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    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)
            ;
```

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

327         if (rTokenPrevBal != 0) {
328             // 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
329             uint256 targetVD = ((vdTokenBal * _assetAmount) / (rTokenPrevBal -
                    vdTokenBal));
330             targetVD += vdTokenBal;
331             (address[] memory _assetToLoop, uint256[] memory _targetVDs) = _loopData(
                    _asset, targetVD);

333             _loop(_assetToLoop, _targetVDs);
334         }

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

338         emit NewAssetDeposit(_for, _asset, _assetAmount, poolInfo.receiptToken, shares);
339     }
```

Listing 3.6: `RadiantStaking::depositAssetFor()`

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

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

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

279         return (rTokenBal - vdTokenBal) * WAD / reciptTokenTotal;
280     }
```

Listing 3.7: `RadiantStaking::assetPerShare()`

Specifically, when the pool is being initialized (line 275), the share value directly takes the value of `_assetAmount` (line 319), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = _assetAmount = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been resolved as the team plans to follow a guarded launch so that a trusted user will be the first to deposit.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Radpie` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, add reward pools, and execute privileged ops). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `RDNTRewardManager` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    function startVestingAll() external onlyOwner {
        IRadiantStaking(radiantStaking).vestAllClaimableRDNT();
        nextVestingTime = block.timestamp + RDNTVestingCoolDown; // nextVestingTime has
            to be updated as block.timestamp + RDNTVestingDays
    }

    /// @dev Radpie to claim all vested RDNT and transfer RDNT to RDNTVest Manager so
        user can claim
    function collectVestedRDNTAll() external onlyOwner {
        if (block.timestamp < nextVestingTime) revert VestingTimeNotReached();
        IRadiantStaking(radiantStaking).claimVestedRDNT();
    }

    function setRDNTVestManager(address _rdntVestManager) external onlyOwner {
        if (_rdntVestManager == address(0)) revert NotAllowZeroAddress();
        rdntVestManager = _rdntVestManager;
    }

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

```
    }
```

Listing 3.8: Example Privileged Operations in the RDNTRewardManager Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged owner account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms they plan to use multi-sig for all admin roles.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterRadpie, RadiantStaking
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the approve() routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of approve(), there is a requirement, i.e., require(!((_value != 0) && (allowed[msg.sender][_spender] != 0))). This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling approve(_spender, 0)) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known approve()/transferFrom() race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
```

```
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.9: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.10: `SafeERC20::safeApprove()`

In current implementation, if we examine the `RadiantStaking::_deleverage()` routine that is designed to deleverage into an intended borrow position. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (lines 800 and 808).

```
    function _deleverage(address[] memory _assets, uint256[] memory _targetVdBal) internal nonReentrant {
        uint256 length = _assets.length;

        for (uint256 i = 0; i < length; i++) {
            Pool storage poolInfo = pools[_assets[i]];
            uint256 vdBal = IERC20(poolInfo.vdToken).balanceOf(address(this));
            uint256 vdDiff = vdBal - _targetVdBal[i];

            while (vdDiff > 0) {
                RadiantUtilLib.PositionStats memory stats = RadiantUtilLib.quoteLeverage(
                    lendingPool,
                    address(this),
                    poolInfo.rToken
                );
                uint256 amountToWithdraw = vdDiff > stats.maxWithdrawAmount
                    ? stats.maxWithdrawAmount
                    : vdDiff;
                uint256 assetRecAmount = _safeWithdrawAsset(
                    _assets[i],
                    poolInfo.rToken,
                    amountToWithdraw,
                    poolInfo.isNative
                );

                if (poolInfo.isNative) {
                    IERC20(poolInfo.rToken).approve(address(wethGateway), assetRecAmount);
                    IWETHGateway(wethGateway).repayETH{ value: assetRecAmount }(
                        address(lendingPool),
                        assetRecAmount,
                        2,
                        address(this)
                    );
                } else {
                    IERC20(poolInfo.asset).approve(address(lendingPool), assetRecAmount);
                    ILendingPool(lendingPool).repay(
                        poolInfo.asset,
                        assetRecAmount,
                        2,
                        address(this)
                    );
                }
                vdDiff -= amountToWithdraw;

                emit Deleverage(poolInfo.asset, assetRecAmount);
            }

            poolInfo.lastActionHandled = block.timestamp; // RDNT claimmable updated on Radiant ChefIncetiveContoller;
```

```
        }
    }
```

Listing 3.11: `RadiantStaking::_deleverage()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`. Note another related routine `MasterRadpie::_depositAsset()` shares the same issue.

Status The issue has been fixed by the following PR: `16`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Radpie` protocol, which is a `SubDAO` designed to maximize yield and streamline governance for `Radiant` users. It leverages the strong `Radiant` infrastructure for superior benefits. The platform's core mechanism involves the locking of `dLP` tokens, which fortifies governance rights and triggers `RDNT` distribution for deposits and borrows in the `Radiant` ecosystem. `Radpie` enables `Radiant` users and `dLP` holders to access `RDNT` rewards and increased revenue without any lock-up period. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.