SMART CONTRACT AUDIT REPORT

for

Radiant V2

Prepared By: Xiaomi Huang

PeckShield

March 5, 2023

Document Properties

Client	Radiant
Title	Smart Contract Audit Report
Target	Radiant V2
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 5, 2023	Stephen Bie	Final Release
1.0-rc	February 8, 2023	Stephen Bie	Release Candidate

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 Radiant V2 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 Radiant

Radiant is the first omnichain money market built atop LayerZero, where users can deposit any major asset on any major chain and borrow/withdraw a variety of supported assets across multiple chains. The audited protocol is in essence a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., AAVE. The protocol allows users to participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. The audited V2 extends the original version with new features for staking-based incentivization and fee distribution. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Radiant V2

Item	Description
Name	Radiant
Website	https://radiant.capital/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 5, 2023

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

- https://github.com/radiant-capital/audit.git (749a460)

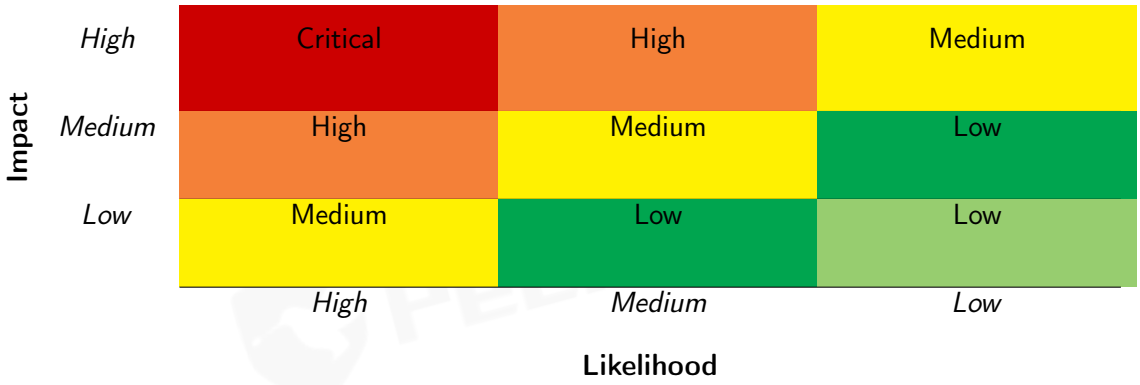
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/radiant-capital/audit.git (acd3e52)

1.2 About PeckShield

PeckShield Inc. [13] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

- 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 checklist of 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 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) [11], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Radiant V2 protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings
Critical	0
High	2
Medium	4
Low	1
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 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 2 high-severity vulnerabilities, 4 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Radiant V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Logic of StargateBorrow::borrowETH()	Business Logic	Fixed
PVE-002	Medium	Revisited Bridge Fee Calculation in RadiantOFT::_getBridgeFee()	Business Logic	Fixed
PVE-003	High	Revisited Withdrawal Amount Calculation in MultiFeeDistribution::exit()	Business Logic	Fixed
PVE-004	High	Possible Price Manipulation for UniswapPoolHelper::getLpPrice()/getPrice()	Time and State	Fixed
PVE-005	Medium	Possible Sandwich/MEV Attacks for Reduced Returns	Time and State	Fixed
PVE-006	Low	Possible Overflow/Underflow Prevention with SafeMath	Coding Practices	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improper Logic of StargateBorrow::borrowETH()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: StargateBorrow
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

In the Radiant V2 protocol, the StargateBorrow contract is one of the main entries for interaction with users, which allows the user to borrow assets from the lending pool on the source chain and bridge the borrowed assets to the destination chain via Stargate. In particular, the internal borrowETH() routine (called inside borrow()) is designed to borrow ETH and bridge the borrowed ETH to the user specified destination chain. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the StargateBorrow contract. Inside the borrowETH() routine, the lendingPool.borrow() is called (line 189) to borrow WETH from the lending pool. And then the borrowed WETH is unwrapped back to ETH (line 196). The unwrapped ETH minus the bridge fee (line 199) will be bridged to the destination chain via the call to router.swap() (line 200). However, it ignores the fact that the router.swap() cannot support native token (i.e., ETH) cross-chain transfer, which will result in getting the transaction always reverted.

```
156     function borrow(
157         address asset,
158         uint256 amount,
159         uint256 interestRateMode,
160         uint16 dstChainId
161     ) external payable {
162         if (address(asset) == ETH_ADDRESS) {
163             borrowETH(amount, interestRateMode, dstChainId);
164         } else {
```

```
            ...
        }
    }
```

Listing 3.1: StargateBorrow::borrow()

```
    function borrowETH(
        uint256 amount,
        uint256 interestRateMode,
        uint16 dstChainId
    ) internal {
        lendingPool.borrow(
            address(WETH),
            amount,
            interestRateMode,
            0,
            msg.sender
        );
        WETH.withdraw(amount);
        uint256 feeAmount = getXChainBorrowFeeAmount(amount);
        _safeTransferETH(daoTreasury, feeAmount);
        amount = amount.sub(feeAmount);
        router.swap{value: msg.value}(
            dstChainId, // dest chain id
            PoolIdETH, // src chain pool id
            PoolIdETH, // dst chain pool id
            msg.sender, // receive address
            amount, // transfer amount
            amount.mul(99).div(100), // max slippage: 1%
            IStargateRouter.lzTxObj(0, 0, "0x"),
            abi.encodePacked(msg.sender),
            bytes("")
        );
    }
```

Listing 3.2: StargateBorrow::borrowETH()

Recommendation Wrap ETH to SGETH before the call to router.swap().

Status The issue has been addressed by the following commit: 5350354.

3.2 Revisited Bridge Fee Calculation in RadiantOFT::_getBridgeFee()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: RadiantOFT
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The RadiantOFT contract implements an OFT-20 token (i.e., NOVA) based on the LayerZero, which is the Radiant's native utility token. The LayerZero Labs's Omnichain Fungible Token (i.e., OFT) interoperability solution enables native, cross-chain token transfers. With LayerZero's guarantee of valid delivery, the token is burned on the source chain and minted on the destination chain directly through the token contract. In particular, the Radiant V2 protocol will charge a certain amount of native token (e.g., ETH) as bridge fee during cross-chain token transfers. The _getBridgeFee() routine is designed to calculate the bridge fee. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the RadiantOFT contract. By design, the amount of the bridge fee is related to the value of the cross-chain token. Inside the _getBridgeFee() routine, the getTokenPrice() routine is called (line 128) to retrieve the price of the token against ETH. The statement of rdntAmount.mul(10**priceDecimals).div(priceInEth).mul(10**18).div(10**_decimals) (line 130) is designed to measure the value of the cross-chain token against ETH. Apparently, it does not meet the requirement. We suggest to improve the implementation as below: rdntAmount.mul(priceInEth).div(10**priceDecimals).mul(10**18).div(10**_decimals) (line 130).

```
124     function _getBridgeFee(uint256 rdntAmount) internal view returns (uint256) {
125         if (address(priceProvider) == address(0)) {
126             return 0;
127         }
128         uint256 priceInEth = priceProvider.getTokenPrice(true);
129         uint256 priceDecimals = priceProvider.decimals();
130         uint256 rdntInEth = rdntAmount.mul(10**priceDecimals).div(priceInEth).mul
                (10**18).div(10**_decimals);
131         return rdntInEth.mul(FEE_BRIDGING).div(FEE_DIVISOR);
132     }
```

Listing 3.3: RadiantOFT::_getBridgeFee()

Recommendation Properly calculate the value of the cross-chain token inside the _getBridgeFee() routine.

Status The issue has been addressed by the following commit: `fd198ea`.

3.3 Revisited Withdrawal Amount Calculation in MultiFeeDistribution::exit()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `MultiFeeDistribution`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

In the `Radiant V2` protocol, the `MultiFeeDistribution` contract provides an incentive mechanism that rewards the staking of the supported `stakingToken` with certain reward tokens. In particular, one entry routine, i.e., `exit()`, is designed to withdraw the unlocked and locked earnings. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `MultiFeeDistribution` contract. Inside the exit() routine, the `withdrawableBalance()` is called (line 1013) to calculate the withdrawable earnings amount. Especially, its returned `amount` includes the unlocked earnings amount (i.e., `bal.unlocked`) (line 644). However, we observe the `bal.unlocked` is added to `amount` again inside the `exit()` routine (line 1018), which will result in unexpected earnings for the user. Given this, we suggest to remove the statement of `amount = amount + bal.unlocked` (line 1018).

```
1007    function exit(bool claimRewards) external override {
1008        address onBehalfOf = msg.sender;
1009        (
1010            uint256 amount,
1011            uint256 penaltyAmount,
1012            uint256 burnAmount
1013        ) = withdrawableBalance(onBehalfOf);
1014
1015        delete userEarnings[onBehalfOf];
1016
1017        Balances storage bal = balances[onBehalfOf];
1018        amount = amount + bal.unlocked;
1019        bal.total = bal.total.sub(bal.unlocked).sub(bal.earned);
1020        bal.unlocked = 0;
1021        bal.earned = 0;
1022
1023        _withdrawTokens(
1024            onBehalfOf,
1025            amount,
1026            penaltyAmount,
```

```
            burnAmount,
            claimRewards
        );
    }
```

Listing 3.4: `MultiFeeDistribution::exit()`

```
    function withdrawableBalance(address user)
        public
        view
        returns (
            uint256 amount,
            uint256 penaltyAmount,
            uint256 burnAmount
        )
    {
        Balances storage bal = balances[user];
        uint256 earned = bal.earned;
        ...
        amount = bal.unlocked.add(earned).sub(penaltyAmount);
        return (amount, penaltyAmount, burnAmount);
    }
```

Listing 3.5: `MultiFeeDistribution::withdrawableBalance()`

Recommendation Correct the `exit()` implementation by properly calculating the withdraw amount.

Status The issue has been addressed by the following commit: `5350354`.

3.4 Possible Price Manipulation for UniswapPoolHelper::getPrice()/getLpPrice()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `UniswapPoolHelper/BalancerPoolHelper`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The `UniswapPoolHelper` contract defines a routine (i.e., `getPrice()`) to obtain the price of the `RDNT` token against `ETH` in the `UniswapV2 RDNT-ETH` pair. While examining its logic, we notice the price of the `RDNT` token is possible to be manipulated.

To elaborate, we show below the related code snippet of the UniswapPoolHelper contract. Inside the getPrice() routine, we observe the final price of the RDNT token against ETH is derived from wethReserve.mul(decis).div(rdntReserve) (line 105), where the value of wethReserve or rdntReserve is the token amount in the UniswapV2 RDNT-ETH pair. Its manipulation may cause the price of the RDNT token not trustworthy.

```
94      function getPrice() public view override returns (uint256 priceInEth) {
95          IUniswapV2Pair lpToken = IUniswapV2Pair(lpTokenAddr);
97          (uint256 reserve0, uint256 reserve1, ) = lpToken.getReserves();
98          uint256 wethReserve = lpToken.token0() != address(rdntAddr)
99              ? reserve0
100             : reserve1;
101         uint256 rdntReserve = lpToken.token0() == address(rdntAddr)
102             ? reserve0
103             : reserve1;
104         uint256 decis = 1e8;
105         priceInEth = wethReserve.mul(decis).div(rdntReserve);
106     }
```

Listing 3.6: UniswapPoolHelper::getPrice()

Note other routines, i.e., UniswapPoolHelper::getLpPrice() and BalancerPoolHelper::getPrice()/getLpPrice(), share the same issue.

Recommendation Revise current execution logic of above-mentioned routines to defensively detect any manipulation attempts.

Status The issue has been addressed by the following commit: fd198ea.

3.5 Possible Sandwich/MEV Attacks for Reduced Returns

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: AutoCompounder/LiquidityZap/BalancerPoolHelper
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

While examining the AutoCompounder contract, we notice there is a routine (i.e., claimAndSwapToBase()) that can be improved with slippage control. To elaborate, we show below the related code snippet of the AutoCompounder contract. By design, the claimAndSwapToBase() routine is used to claim all the rewards and swap various reward tokens to the baseToken. Inside the routine, the

swapExactTokensForTokens() routine of UniswapV2 is called (line 86) to swap the exact token to the baseToken. However, we observe the second input amountOutMin parameter is assigned to 0, which means this transaction does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks.

```
71      function claimAndSwapToBase(address _user) internal {
72          IMultiFeeDistribution mfd = IMultiFeeDistribution(multiFeeDistribution);
73          mfd.claimFromConverter(_user);
74          ILendingPool lendingPool = ILendingPool(ILendingPoolAddressesProvider(
                addressProvider).getLendingPool());
75
76          for(uint256 i = 0; i < rewardBaseTokens.length; i++){
77              uint256 balance = IERC20(rewardBaseTokens[i]).balanceOf(address(this));
78              if(balance == 0){
79                  continue;
80              }
81              address underlying = IAToken(rewardBaseTokens[i]).UNDERLYING_ASSET_ADDRESS()
                    ;
82              uint256 amount = lendingPool.withdraw(underlying, type(uint256).max, address
                    (this));
83
84              if(underlying != baseToken){
85                  IERC20(underlying).safeApprove(uniRouter, amount);
86                  IUniswapV2Router02(uniRouter).swapExactTokensForTokens(
87                      amount, 0, rewardToBaseRoute[underlying], address(this), block.
                            timestamp + 600
88                  );
89              }
90          }
91      }
```

Listing 3.7: AutoCompounder::claimAndSwapToBase()

Note other routines, i.e., LiquidityZap::zapETH()/addLiquidityWETHOnly()/addLiquidityETHOnly() /standardAdd() and BalancerPoolHelper::zapWETH()/zapTokens()/swap(), can also benefit from necessary slippage control.

Recommendation Improve the above-mentioned routines by adding necessary slippage control.

Status The issue has been addressed by the following commit: fd198ea.

3.6 Possible Overflow/Underflow Prevention with SafeMath

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

SafeMath is a Solidity math library especially designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. Our analysis shows that the current implementation can be improved with the use of SafeMath.

In the following, we use the ChefIncentivesController::availableRewards() as an example. This routine is used to calculate the remaining reward that can be distributed. It comes to our attention that an underflow vulnerability will occur if the *depositedRewards* is less than *accountedRewards*. We may intend to use SafeMath library to avoid underflows and/or overflows.

```
675     function availableRewards() internal returns (uint256 amount) {
676         return depositedRewards - accountedRewards;
677     }
```

Listing 3.8: ChefIncentivesController::availableRewards()

Moreover, there are more routines in the ChefIncentivesController/MultiFeeDistribution/RadiantOFT /LzApp/Disqualifier contracts that share the similar issue.

Recommendation Apply the SafeMath to block unintended overflows and/or underflows.

Status The issue has been addressed by the following commit: 5350354.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Radiant V2 protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle adjustment). It

also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
137     function setDisqualifier(IDisqualifier _disqualifier) external onlyOwner {
138         disqualifier = _disqualifier;
139     }
140
141     function setOnwardIncentives(
142         address _token,
143         IOnwardIncentivesController _incentives
144     ) external onlyOwner {
145         require(poolInfo[_token].lastRewardTime != 0);
146         poolInfo[_token].onwardIncentives = _incentives;
147     }
148
149     function setRewardsPerSecond(uint256 _rewardsPerSecond, bool _persist)
150         external
151         onlyOwner
152     {
153         _massUpdatePools();
154         rewardsPerSecond = _rewardsPerSecond;
155         persistRewardsPerSecond = _persist;
156         emit RewardsPerSecondUpdated(_rewardsPerSecond, _persist);
157     }
```

Listing 3.9: ChefIncentivesController

Moreover, the LendingPoolAddressesProvider contract allows the privileged owner to configure protocol-wide contracts, including LENDING_POOL, LENDING_POOL_CONFIGURATOR, POOL_ADMIN, EMERGENCY_ADMIN, LENDING_POOL_COLLATERAL_MANAGER, PRICE_ORACLE, and LENDING_RATE_ORACLE. These contracts play a variety of duties and are also considered privileged.

```
19     contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
20         string private _marketId;
21         mapping(bytes32 => address) private _addresses;
22
23         bytes32 private constant LENDING_POOL = 'LENDING_POOL';
24         bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR'
               ;
25         bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
26         bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
27         bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
28         bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
29         bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';
30         ...
31     }
```

Listing 3.10: LendingPoolAddressesProvider

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure.

Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team. The team intends to introduce multi-sig mechanism to migrate this issue.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Radiant V2 protocol. It is designed to be the first omnichain money market built atop LayerZero, where users can deposit any major asset on any major chain and borrow/withdraw a variety of supported assets across multiple chains. The current implementation extends the original AaveV2 with new features for staking-based incentivization and fee distribution. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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