SMART CONTRACT AUDIT REPORT

for

Radiant Protocol

Prepared By: Xiaomi Huang

PeckShield

July 28, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	July 28, 2022	Xuxian Jiang	Final Release
1.0-rc	July 27, 2022	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 `Radiant` 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 `Layer Zero`, 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 protocol 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

Item	Description
Name	Radiant
Website	https://radiant.capital/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 28, 2022

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/radiant-protocol-deployment.git (eb7f62a)

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/radiant-protocol-deployment.git (f74526a)

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 the 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 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) [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. 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` 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	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key Radiant Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in Leverager::loop()	Coding Practice	Resolved
PVE-002	Low	Improved Staking Logic in MultiFeeDistribution::stake()	Coding Practice	Confirmed
PVE-003	Low	Revisited Stable Borrow Logic in LendingPool	Business Logic	Resolved
PVE-004	Medium	Incentive Inconsistency Between AToken And StableDebtToken	Business Logic	Resolved
PVE-005	Low	Fork-Resistant Domain Separator in AToken	Business Logic	Confirmed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved Logic in Leverager::loop()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Leverager`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1099 [1]

Description

The `Radiant` protocol has provided a convenience contract that forms "looping" on stablecoin borrows to maximize leverage and reward APY. Specifically, given certain target borrowing parameters, the contract is designed to borrow and deposit continuously until a target `health factor` is reached. While reviewing its logic, we notice the current implementation can be improved.

Specifically, while there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. And related idiosyncrasies may pose challenges for seamless integration.

In the following, we use the popular token, i.e., `ZRX`, as our example, and show the related `transfer()` routine. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
```

```
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.1: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `loop()` routine in the `Leverager` contract. If the `USDT` token is supported as `asset`, the unsafe version of `IERC20(asset).transferFrom(msg.sender, address(this), amount)` (line 68) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)! Note the same issue is also applicable to the `WETHGateway::emergencyTokenTransfer()` function.

```
60    function loop(
61        address asset,
62        uint256 amount,
63        uint256 interestRateMode,
64        uint256 borrowRatio,
65        uint256 loopCount
66    ) external {
67        uint16 referralCode = 0;
68        IERC20(asset).transferFrom(msg.sender, address(this), amount);
69        IERC20(asset).approve(address(lendingPool), type(uint256).max);
70        lendingPool.deposit(asset, amount, msg.sender, referralCode);
71        for (uint256 i = 0; i < loopCount; i += 1) {
72            amount = amount.mul(borrowRatio).div(10 ** BORROW_RATIO_DECIMALS);
73            lendingPool.borrow(asset, amount, interestRateMode, referralCode, msg.sender
                  );
74            lendingPool.deposit(asset, amount, msg.sender, referralCode);
75        }
76    }
```

Listing 3.2: `Leverager::loop()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. For the safe-version of `approve()`, there is a need to `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new allowance.

Status TBD This issue has been resolved in the following commit: `f74526a`.

3.2 Improved Staking Logic in MultiFeeDistribution::stake()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MultiFeeDistribution`
- Category: Coding Practices [7]
- CWE subcategory: CWE-663 [3]

Description

As mentioned earlier, the `Radiant` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. While reviewing the current staking logic, we notice its implementation can be improved.

To elaborate, we show below the related staking function `stake()` from the `MultiFeeDistribution` contract. This function implements a rather straightforward logic in staking the user funds and updating the accounting information on the staked funds. It comes to our attention that this function has a parameter `lock` to indicate whether the staked funds will be locked or not. However, it also has the following requirement statement `require(lock == true, "Staking disabled")` (line 267), which makes the above parameter redundant. In other words, we can either remove this requirement, or adjust the implementation to remove the `else`-branch (lines 282-284).

```
265     function stake(uint256 amount, bool lock, address onBehalfOf) external override {
266         require(amount > 0, "Cannot stake 0");
267         require(lock == true, "Staking disabled");
268         _updateReward(onBehalfOf);
269         totalSupply = totalSupply.add(amount);
270         Balances storage bal = balances[onBehalfOf];
271         bal.total = bal.total.add(amount);
272         if (lock) {
273             lockedSupply = lockedSupply.add(amount);
274             bal.locked = bal.locked.add(amount);
275             uint256 unlockTime = block.timestamp.div(rewardsDuration).mul(
                    rewardsDuration).add(lockDuration);
276             uint256 idx = userLocks[onBehalfOf].length;
277             if (idx == 0  userLocks[onBehalfOf][idx-1].unlockTime < unlockTime) {
278                 userLocks[onBehalfOf].push(LockedBalance({amount: amount, unlockTime:
                        unlockTime}));
```

```
            } else {
                userLocks[onBehalfOf][idx-1].amount = userLocks[onBehalfOf][idx-1].
                    amount.add(amount);
            }
        } else {
            bal.unlocked = bal.unlocked.add(amount);
        }
        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
        emit Staked(onBehalfOf, amount, lock);
    }
```

Listing 3.3: `MultiFeeDistribution::stake()`

Recommendation Revise the above `stake()` function to avoid unnecessary redundancy.

Status The issue has been confirmed.

3.3 Revisited Stable Borrow Logic in LendingPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `LendingPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Radiant` protocol has the core `LendingPool` contract that provides a number of core routines for borrowing/lending users to interact with, including `deposit()`, `withdraw()`, `borrow()`, `repay()`, `flashloan()`, and etc. To facilitate the execution of each core routine, `Radiant` validates the given arguments to these core routines with corresponding validation routines in `ValidationLogic`, such as `validateDeposit()`, `validateWithdraw()`, `validateBorrow()`, `validateRepay()`, `validateFlashloan()`, and etc.

More importantly, all the actions performed in each core routine follow a specific sequence:

- Step I: It firstly validates the given arguments as well as current state. If current state cannot meet the pre-conditions required for the intended action, the transaction will be reverted.

- Step II: It then updates reserve state to reflect the latest borrow/liquidity indexes (up to the current block height) and further calculates the new amount that will be minted to the treasury. The updated indexes are necessary to get the reserve ready for the execution of the intended action.

- Step III: It next "executes" the intended action that may need to update the user accounting and reserve balance as the action could involve transferring assets into or out of the reserve. The updates could lead to minting or burning of tokens that are related to lending/borrowing positions of current user. The tokens are represented as `ATokens`, `StableDebtTokens`, or `VariableDebtTokens`.

- Step IV: Due to possible changes to the reserve from the action, such as resulting in a different utilization rate from either borrowing or lending, it also needs to accordingly adjust the interest rates to accurately accrue interests.

- Step V: By following the known best practice of the `checks-effects-interactions` pattern, it finally performs the external interactions, if any.

One of the advanced features implemented in `Radiant` is the tokenization of both lending and borrowing positions. When a user deposits assets into a specific reserve, the user receives the corresponding amount of `ATokens` to represent the liquidity deposited and accrue the interests. When a user opens or increases a borrow position, the user receives the corresponding amount of `DebtTokens` (either `StableDebtTokens` or `VariableDebtTokens` depending on the borrow mode) to represent the debt position and further accrue the debt interests.

The above order sequence needs to be properly maintained. Moreover, if a borrow is being requested, Step III and IV need to ensure that the proper borrow rate is used. Our analysis shows that the current implementation can be improved when a stable borrow mode is chosen. To elaborate, we show below the related function `_executeBorrow()`.

This function abstracts the core logic in performing a borrow operation. When a stable borrow is requested, we notice the Step III makes use of the `reserve.currentStableBorrowRate` state to mint the associated `StableDebtTokens` (lines 897-902). However, it comes to our attention that this `reserve.currentStableBorrowRate` was computed using the last utilization rate, not the latest one with the current borrow. In other words, the stable borrow rate needs to be re-computed by taking into account the borrow amount just requested! Otherwise, the current implementation introduces stark inconsistency in the handling of stable and variable borrows, and the inconsistency is currently in favor of borrowing users at the cost of existing liquidity providers.

```
863   function _executeBorrow(ExecuteBorrowParams memory vars) internal {
864     DataTypes.ReserveData storage reserve = _reserves[vars.asset];
865     DataTypes.UserConfigurationMap storage userConfig = _usersConfig[vars.onBehalfOf];
866
867     address oracle = _addressesProvider.getPriceOracle();
868
869     uint256 amountInETH =
870       IPriceOracleGetter(oracle).getAssetPrice(vars.asset).mul(vars.amount).div(
871         10**reserve.configuration.getDecimals()
872       );
```

```
    ValidationLogic.validateBorrow(
      vars.asset,
      reserve,
      vars.onBehalfOf,
      vars.amount,
      amountInETH,
      vars.interestRateMode,
      _maxStableRateBorrowSizePercent,
      _reserves,
      userConfig,
      _reservesList,
      _reservesCount,
      oracle
    );

    reserve.updateState();

    uint256 currentStableRate = 0;

    bool isFirstBorrowing = false;
    if (DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.
        STABLE) {
      currentStableRate = reserve.currentStableBorrowRate;

      isFirstBorrowing = IStableDebtToken(reserve.stableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        currentStableRate
      );
    } else {
      isFirstBorrowing = IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        reserve.variableBorrowIndex
      );
    }

    if (isFirstBorrowing) {
      userConfig.setBorrowing(reserve.id, true);
    }

    reserve.updateInterestRates(
      vars.asset,
      vars.aTokenAddress,
      0,
      vars.releaseUnderlying ? vars.amount : 0
    );

    if (vars.releaseUnderlying) {
```

```
      IAToken(vars.aTokenAddress).transferUnderlyingTo(vars.user, vars.amount);
    }

    emit Borrow(
      vars.asset,
      vars.user,
      vars.onBehalfOf,
      vars.amount,
      vars.interestRateMode,
      DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.STABLE
        ? currentStableRate
        : reserve.currentVariableBorrowRate,
      vars.referralCode
    );
  }
```

Listing 3.4: `LendingPool::_executeBorrow()`

Recommendation Revise the above borrow routine to ensure the latest `stable borrow rate` is used.

Status This issue has been resolved as the team clarifies no plan to enable the stable borrow feature.

3.4 Incentive Inconsistency Between AToken And StableDebtToken

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AToken`, `StableDebtToken`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Radiant` protocol extends the built-in `IncentivesController` framework to engage protocol users. While reviewing the logic to integrate the incentive mechanism, we observe unnecessary inconsistency that may introduce unwanted confusion and errors.

To elaborate, we show below the `_mint()` function from both `IncentivizedERC20` and `StableDebtToken` contracts. It comes to our attention that the first contract uses the post-update balance in the invocation of `IncentivesController::handleAction()` (line 212) while the second contract uses the pre-update balance in the invocation of `IncentivesController::handleAction()` (line 413)!

```
function _mint(address account, uint256 amount) internal virtual {
  require(account != address(0), 'ERC20: mint to the zero address');

  _beforeTokenTransfer(address(0), account, amount);

  uint256 currentTotalSupply = _totalSupply.add(amount);
  _totalSupply = currentTotalSupply;

  uint256 accountBalance = _balances[account].add(amount);
  _balances[account] = accountBalance;

  if (address(_getIncentivesController()) != address(0)) {
    _getIncentivesController().handleAction(account, accountBalance,
        currentTotalSupply);
  }
}
```

Listing 3.5: `IncentivizedERC20::_mint()`

```
function _mint(
  address account,
  uint256 amount,
  uint256 oldTotalSupply
) internal {
  uint256 oldAccountBalance = _balances[account];
  _balances[account] = oldAccountBalance.add(amount);

  if (address(_incentivesController) != address(0)) {
    _incentivesController.handleAction(account, oldAccountBalance, oldTotalSupply);
  }
}
```

Listing 3.6: `StableDebtToken::_mint()`

Recommendation Be consistent in using the account balance for incentivization measurement.

Status This issue has been resolved as the team clarifies no plan to enable the stable borrow feature.

3.5 Fork-Resistant Domain Separator in AToken

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: AToken
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The various tokens in Radiant are designed to strictly follows the widely-accepted ERC20 specification (Section 3.4). In the meantime, we notice the support of EIP-2612 with the permit() function that allows for approvals to be made via secp256k1 signatures. Interestingly, we notice the state variable DOMAIN_SEPARATOR in AToken is initialized once inside the initialize() function (lines 81-89).

```
64  function initialize(
65    ILendingPool pool,
66    address treasury,
67    address underlyingAsset,
68    IAaveIncentivesController incentivesController,
69    uint8 aTokenDecimals,
70    string calldata aTokenName,
71    string calldata aTokenSymbol,
72    bytes calldata params
73  ) external override initializer {
74    uint256 chainId;
75
76    //solium-disable-next-line
77    assembly {
78      chainId := chainid()
79    }
80
81    DOMAIN_SEPARATOR = keccak256(
82      abi.encode(
83        EIP712_DOMAIN,
84        keccak256(bytes(aTokenName)),
85        keccak256(EIP712_REVISION),
86        chainId,
87        address(this)
88      )
89    );
90
91    _setName(aTokenName);
92    _setSymbol(aTokenSymbol);
93    _setDecimals(aTokenDecimals);
94
95    _pool = pool;
96    _treasury = treasury;
97    _underlyingAsset = underlyingAsset;
```

```
    _incentivesController = incentivesController;

    emit Initialized(
      underlyingAsset,
      address(pool),
      treasury,
      address(incentivesController),
      aTokenDecimals,
      aTokenName,
      aTokenSymbol,
      params
    );
  }
```

Listing 3.7: `AToken::initialize()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
  function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
  ) external {
    require(owner != address(0), 'INVALID_OWNER');
    //solium-disable-next-line
    require(block.timestamp <= deadline, 'INVALID_EXPIRATION');
    uint256 currentValidNonce = _nonces[owner];
    bytes32 digest =
      keccak256(
        abi.encodePacked(
          '\x19\x01',
          DOMAIN_SEPARATOR,
          keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, currentValidNonce, deadline))
        )
      );
    require(owner == ecrecover(digest, v, r, s), 'INVALID_SIGNATURE');
    _nonces[owner] = currentValidNonce.add(1);
    _approve(owner, spender, value);
  }
```

Listing 3.8: `AToken::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status This issue has been confirmed.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Radiant` 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.

```
    function setOnwardIncentives(
        address _token,
        IOnwardIncentivesController _incentives
    )
        external
        onlyOwner
    {
        require(poolInfo[_token].lastRewardTime != 0);
        poolInfo[_token].onwardIncentives = _incentives;
    }

    function setClaimReceiver(address _user, address _receiver) external {
        require(msg.sender == _user  msg.sender == owner());
        claimReceiver[_user] = _receiver;
    }
```

Listing 3.9: Example Setters in 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.

```
contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
  string private _marketId;
  mapping(bytes32 => address) private _addresses;
```

```

bytes32 private constant LENDING_POOL = 'LENDING_POOL';
bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR';
bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';
...
}
```

Listing 3.10: The LendingPoolAddressesProvider Contract

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 This issue has been confirmed with the team. For the time being, it is planned to mitigate with a 2-day timelock (owned by a multi-sig account) to balance efficiency and timely adjustment.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Radiant` protocol, which is the first omnichain money market built atop `Layer Zero`, 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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