Security Testing Report for Radiant V2

Date: Mar 21, 2023

Version: 2.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.03.22 13:56:06
+08'00'

Contents

Report Manifest

Item	Description
Client	Radiant Capital
Target	Radiant V2

Version History

Version	Date	Description
1.0	March 15, 2023	First Version
2.0	March 21, 2023	Second Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Security Testing

We are invited by Radiant Capital to conduct a **security testing** (as the red team) for the Radiant V2's smart contracts to identify potential risks. As a responsible team, Radiant Capital takes security seriously. Hence the team decided to put more efforts into securing those smart contracts, while they have been audited by multiple security companies [1].

Note that security testing is different from security audit in both goals and requirements. Specifically, security testing aims to discover extra/unusual vulnerable **points** by mimicking attackers to break the program/protocol, while security audit aims to give a relatively comprehensive security check by enumerating the possible attack **surfaces**. As such, security testing might not be able to cover some complicated logic bugs that could be identified by security audit due to the limited time and resource.

1.2 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Static analysis, dynamic analysis, Semi-automatic and manual verification

The target repository is Radiant_v2.1.1 [2]. The commit SHA values during the security testing are shown in the following. Our report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the report.

Project		Commit SHA
Radiant	Version 1	1328c4d9015b035530b516de817252fb0df8e11e
	Version 2	d104447bc5af84e0c62cfa814b3ecdf7c024c0f7
	Version 3	bd641dd13ed52f2c731f47263e2f2bd144683d81
	Version 4	bcc5ff3674c6d865115d69936b5744dd314fec0b
	Version 5	77664d84ddcf77089dbb629700d2276fdedf1bca
	Version 6	1cd1f60d3bc90ade88fcced8ed3406867c6dcc97
	Version 7	acd3e5284e5069ac23ee08edace5520e31957d58
	Version 8	3c877c12af60bb168231c4bc0254da51731bdb7a
	Version 9	757288c422cabd63a97cf9fa6a9f3adee25abd76
	Version 10	40776c3cd3c88e2d9d17dacdb28da63043054e43
	Version 11	156d2b578d5a22ffca318fedf45e26bec872e932

Note that, this report only covers smart contracts under the **radiant_v2.1.1/contracts** folder of this repository, including:

- bounties

[1] https://twitter.com/RDNTCapital/status/1625579906502475776

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

- deployments
- flashloan
- leverage
- lock
- oracles
- staking
- zap
- eligibility
- misc
- oft
- protocol
- stargate

After the update in `Version 8`,the files covered in this security testing include:

- lending/AaveOracle.sol
- lending/AaveProtocolDataProvider.sol
- lending/ATokensAndRatesHelper.sol
- lending/StableAndVariableTokensHelper.sol
- lending/UiPoolDataProviderV2V3.sol
- lending/UiPoolDataProvider.sol
- lending/WETHGateway.sol
- lending/WalletBalanceProvider.sol
- lending/configuration
- lending/flashloan
- lending/lendingpool
- lending/tokenization
- radiant/accessories
- radiant/eligibility
- radiant/oracles
- radiant/staking
- radiant/token
- radiant/zap

1.3 Disclaimer

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

This report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This security testing does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As the security testing cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the

security of smart contracts.

The scope of this security testing is limited to the code mentioned in Section 1.2. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.4 Procedure of Security Testing

We perform the security testing according to the following procedure. We will first go through/review the code to understand the overall design and interactions between different modules, and then conduct the security testing based on our in-house tools:

- the code review is based on our know-how of potential attack surfaces derived from our previous research and experience.
- we will use some in-house tools including fuzzing tools to locate possible vulnerabilities. For each issue being discovered, we will provide the PoC for confirmation.

We show the main concrete checkpoints in the following.

1.4.1 Software Security

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

1.4.2 DeFi Security

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

1.4.3 NFT Security

* Duplicated item

- Verification of the token receiver
- Off-chain metadata security

1.4.4 Additional Recommendation

- Gas optimization
- Code quality and style

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

1.5 Security Model

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

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

Table 1.1: Vulnerability Severity Classification

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

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

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

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

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

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

Chapter 2 Automated Security Testing

2.1 Automated Static Security Testing

We use our in-house static analysis tool based on Slither to check the existence of the vulnerabilities. After checking the results manually, no issues were found. Detailed testing results can be found in Table 4.1 in Appendix.

2.2 Automated Dynamic Security Testing

We utilize fuzzing techniques to test the robustness, reliability, and precision of the target contracts. Specifically, the initial seed in the fuzzing process is determined based on the function semantics and contract test scripts. To simulate the on chain environment, we also maintain a set of addresses that have interacted with the contract `LendingPool` and `MultiFeeDistribution`.

Our fuzzer also considers function semantics during transaction sequence generation. For example, function `stake` in contract `MultiFeeDistribution` and function `deposit` in contract `LendingPool` are likely to be invoked first in the sequence. The mutation to the function parameters and sequence is guided by the contract code coverage. If a certain parameter or sequence reaches higher code coverage, it will has higher priority to be mutated in the next fuzzing round. To explore some path constrained by magic number, we collect the values read from storage (i.e., the `SLOAD` instruction) at runtime and use them to generate function parameters during the mutation process.

In total, we generate 100,000 test cases and utilize 31 oracles, which is used to detect if a failure has occurred. For each test case, it contains 30 transactions with specified orders. Finally, we discovered one critical issue (i.e., Section 3.2.6), which is also discovered in our manual security testing process. Detailed testing results can be found in Tables 4.2, 4.3, and 4.4 in Appendix.

Chapter 3 Manual Security Testing

We involve manual efforts to understand the overall design and interactions between different modules, and then conduct the security testing based on our know-how of potential attack surfaces derived from our previous research and experience.

In total, we find **seventeen** potential issues. Besides, we have **three** recommendations and **one** notes as follows:

- High Risk: 2
- Medium Risk: 8
- Low Risk: 7
- Recommendations: 3
- Notes: 1

ID	Severity	Description	Category	Status
1	Medium	No Reserved Interface for Resetting Function Pointers	Software Security	Fixed
2	Medium	Improper Calculation of the Oracle	DeFi Security	Fixed
3	High	Potential Drain of Funds through BaseBounty	DeFi Security	Fixed
4	Low	Potential Invalid Emission Schedules	DeFi Security	Fixed
5	Low	Skippable Emission schedules	DeFi Security	Confirmed
6	Medium	Changeable Exchange Rate during Migration	DeFi Security	Fixed
7	High	Improper Implementation of _transfer() (I)	DeFi Security	Fixed
8	Low	Lack of Check on Period in UniV2TwapOracle	DeFi Security	Fixed
9	Medium	Non-Refundable Dust Tokens	DeFi Security	Fixed
10	Medium	Improper Implementation of _transfer() (II)	DeFi Security	Fixed
11	Medium	Manipulatable Compound Rewards	DeFi Security	Fixed
12	Medium	Lack of Access Control in setLeverager()	DeFi Security	Fixed
13	Medium	No Slippage Check in addLiquidityWETHOnly()	DeFi Security	Confirmed
14	Low	Lack of Check of borrowRatio in loopETH()	DeFi Security	Fixed
15	Low	Lack of Check of Length between assets and poolIDs in setPoolIDs()	DeFi Security	Fixed
16	Low	Lack of mint Privilege Revoke in addBountyContract()	DeFi Security	Confirmed
17	Low	Minters Can Only be Assigned Once	DeFi Security	Confirmed
18	-	Gas Optimization (zapVestingToLp() in Mfd)	Recommendation	Fixed
19	-	Non-empty Bounty Reserve in BountyManager	Recommendation	Fixed
20	-	Inconsistent Naming in requiredUsdValue()	Recommendation	Confirmed
21	-	Depreciated MFDPlus	Note	Confirmed

The details are provided in the following sections.

3.1 Software Security

3.1.1 No Reserved Interface for Resetting Function Pointers

Severity Medium

Status Fixed in `Version 7`

Introduced by `Version 1`

Description Three functions, `getLpMfdBounty()`, `getChefBounty()`, and `getAutoCompoundBounty()`, are invoked through function pointers in contract `BountyManager`. Meanwhile, the inheritance from `Ownable-Upgradable` shows that this contract would be the implementation of a proxy. This indicates that the implementation contract can be upgraded in the future, which brings an issue related to the function pointers.

```
77    function initialize(
78        address _rdnt,
79        address _weth,
80        address _lpMfd,
81        address _mfd,
82        address _chef,
83        address _priceProvider,
84        address _eligibilityDataProvider,
85        uint256 _hunterShare,
86        uint256 _baseBountyUsdTarget,
87        uint256 _maxBaseBounty,
88        uint256 _bountyBooster
89    ) external initializer {
90        require(_rdnt != address(0));
91        require(_weth != address(0));
92        require(_lpMfd != address(0));
93        require(_mfd != address(0));
94        require(_chef != address(0));
95        require(_priceProvider != address(0));
96        require(_eligibilityDataProvider != address(0));
97        require(_hunterShare <= 10000);
98        require(_baseBountyUsdTarget != 0);
99        require(_maxBaseBounty != 0);
100
101       rdnt = _rdnt;
102       weth = _weth;
103       lpMfd = _lpMfd;
104       mfd = _mfd;
105       chef = _chef;
106       priceProvider = _priceProvider;
107       eligibilityDataProvider = _eligibilityDataProvider;
108
109       HUNTER_SHARE = _hunterShare;
110       baseBountyUsdTarget = _baseBountyUsdTarget;
111       bountyBooster = _bountyBooster;
112       maxBaseBounty = _maxBaseBounty;
113
114       bounties[1] = getLpMfdBounty;
115       bounties[2] = getChefBounty;
116       bounties[3] = getAutoCompoundBounty;
117       bountyCount = 3;
118
119       slippageLimit = 10;
120       minDLPBalance = uint256(5).mul(10 ** 18);
121
```

```
122
123         __Ownable_init();
124         __Pausable_init();
125     }
```

Listing 3.1: BountyManager.sol

Impact When the offsets of the above mentioned three functions are changed, the function pointers cannot work as expected and the whole logic of the contract can be changed.

Suggestion The contract should provide interfaces for resetting the function pointers.

3.2 DeFi Security

3.2.1 Improper Calculation of the Oracle

Severity Medium

Status Fixed in `Version 11`

Introduced by `Version 1` and `Version 4`

Description The function `consult()` in contract `ComboOracle` is used to compute the average price from several sources. In the implementation of version 1, it uses arithmetic mean to calculate the final price, which can be manipulated by influencing one of the source oracles.

```
36     function consult() public view override returns (uint256 price) {
37         require(sources.length != 0);
38
39         uint256 sum;
40         for (uint256 i = 0; i < sources.length; i++) {
41             uint256 price = sources[i].consult();
42             require(price != 0, "source consult failure");
43             sum = sum.add(price);
44         }
45         price = sum.div(sources.length);
46     }
```

Listing 3.2: ComboOracle.sol

In the implementation of version 4, when the average price is greater than the lowest price $\times$ 1.025, lowest price will be returned. However, the return value can still be manipulated if the result returned from one of the source oracles is abnormally low.

```
36     /**
37     * @notice Calculated price
38     * @return price Average price of several sources.
39     */
40     function consult() public view override returns (uint256 price) {
41         require(sources.length != 0);
42
43         uint256 sum;
44         uint256 lowestPrice;
45         for (uint256 i = 0; i < sources.length; i++) {
```

```
46          uint256 price = sources[i].consult();
47          require(price != 0, "source consult failure");
48          if (lowestPrice == 0) {
49              lowestPrice = price;
50          } else {
51              lowestPrice = lowestPrice > price ? price : lowestPrice;
52          }
53          sum = sum.add(price);
54      }
55      price = sum.div(sources.length);
56      price = price > ((lowestPrice * 1025) / 1000) ? lowestPrice : price;
57  }
```

Listing 3.3: ComboOracle.sol

Impact The price returned from `ComboOracle` can be manipulated, which allows the attacker to gain profit from it.

Suggestion We suggest using medium value instead of the average value. If there are only two source oracles and a rather big difference occurs, it is more reasonable to revert the transaction when the avarage price is rather larger than the lowest price.

Feedback There will be only two source oracles. If there is a rather big difference occurs, we'll use an `OZ Defender Sentinel` to pause associated contracts.

Note The contract `ComboOracle` is removed and no longer used.

3.2.2 Potential Drain of Funds through BaseBounty

Severity High

Status Fixed in `Version 4`

Introduced by `Version 1`

Description A user can lock tokens (i.e., `RDNT`) for a fixed duration to earn rewards. When the lock expires, other users can invoke the function `executeBounty()` to relock the tokens for this user to earn the `BaseBounty` if this user has the `AutoRelock` enabled. During the relocking process, the expired locks will be cleared and restaked into the pool in the internal function `_cleanWithdrawableLocks()`. However, there is a variable `maxLockWithdrawPerTxn` limiting the maximum number of locks that can be cleared. In this case, uncleared expired locks may still exist even after the function `executeBounty()` being executed. This can further bypass the check in line 106 of function `claimBounty()` in the contract `MFDPlus`. The `issueBaseBounty` will be set as true and returned back.

```
1074    **
1075    * @notice Withdraw all lockings tokens where the unlock time has passed
1076    */
1077   function _cleanWithdrawableLocks(
1078       address user,
1079       uint256 totalLock,
1080       uint256 totalLockWithMultiplier
1081   ) internal returns (uint256 lockAmount, uint256 lockAmountWithMultiplier) {
1082       LockedBalance[] storage locks = userLocks[user];
1083
```

```
1084        if (locks.length != 0) {
1085            uint256 length = locks.length <= maxLockWithdrawPerTxn ? locks.length :
                    maxLockWithdrawPerTxn;
1086            for (uint256 i = 0; i < length; ) {
1087                if (locks[i].unlockTime <= block.timestamp) {
1088                    lockAmount = lockAmount.add(locks[i].amount);
1089                    lockAmountWithMultiplier = lockAmountWithMultiplier.add(
1090                        locks[i].amount.mul(locks[i].multiplier)
1091                    );
1092                    locks[i] = locks[locks.length - 1];
1093                    locks.pop();
1094                    length = length - 1;
1095                } else {
1096                    i = i + 1;
1097                }
1098            }
1099            if (locks.length == 0) {
1100                lockAmount = totalLock;
1101                lockAmountWithMultiplier = totalLockWithMultiplier;
1102                delete userLocks[user];
1103
1104                userlist.removeFromList(user);
1105            }
1106        }
1107    }
```

Listing 3.4: MultiFeeDistribution.sol

Specifically, the attacker can stake 1 wei token with the same expiration time for multiple times, which is rather larger than `maxLockWithdrawPerTxn`. After that, the attacker can set the action as getLpMfdBounty and invoke `executeBounty()` repeatedly. As the amount of cleared locks is limited by the `maxLockWithdrawPerTxn`, the `BaseBounty` in the contract `BountyManager` can be drained by the attacker.

Impact The attacker can drain all funds in the contract `BountyManager` in one transaction, leading to the disruption of designed bounty mechanisms.

Suggestion Ensure the function `_cleanWithdrawableLocks()` can clear all expired locks and set a minimum staking amount in function `_stake()`.

3.2.3 Potential Invalid Emission Schedules

Severity Low

Status Fixed in `Version 10`

Introduced by `Version 1`

Description In the contract `ChefIncentivesController` , function `setEmissionSchedule()` is invoked by the `owner` to set schedules for different rewards rates. In this case, the start time for each schedule (`_startTimeOffsets[i] + startTime`) should be validated to be larger than the current timestamp. However, it only checks the first element in `_startTimeOffsets`, which is not enough. Furthermore, the `_startTimeOffsets[i]` is converted from `uint256` to `uint128` when it's being added to `emissionSchedule`, which can be truncated if the original input is too large.

```
262    function setEmissionSchedule(
263        uint256[] calldata _startTimeOffsets,
264        uint256[] calldata _rewardsPerSecond
265    ) external onlyOwner {
266        uint256 length = _startTimeOffsets.length;
267        require(length > 0 && length == _rewardsPerSecond.length, "empty or mismatch params");
268        if (startTime > 0) {
269            require(_startTimeOffsets[0] > block.timestamp.sub(startTime), "invalid start time");
270        }
271
272        for (uint256 i = 0; i < length; i++) {
273            emissionSchedule.push(
274                EmissionPoint({
275                    startTimeOffset: uint128(_startTimeOffsets[i]),
276                    rewardsPerSecond: uint128(_rewardsPerSecond[i])
277                })
278            );
279        }
280        emit EmissionScheduleAppended(_startTimeOffsets, _rewardsPerSecond);
281    }
```

Listing 3.5: ChefIncentivesController.sol

Impact If `_startTimeOffsets` is not in ascending order, some promised rewards will not be distributed to the users. If `_startTimeOffsets[i]` is beyond the range of `uint128` , an invalid emission schedule will be added.

Suggestion Ensure the `_startTimeOffsets` is in ascending order and all elements are within the `uint128` range.

3.2.4 Skippable Emission schedules

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In contract `ChefIncentivesController`, the function `setScheduleRewardsPerSecond()` will iterate `emissionSchedule` to locate the target schedule with the largest index that has already started, and update the reward rate accordingly. However, in this case, some emission schedules may be skipped.

```
217    function setScheduledRewardsPerSecond() internal {
218    if (!persistRewardsPerSecond) {
219      uint256 length = emissionSchedule.length;
220      uint256 i = emissionScheduleIndex;
221      uint128 offset = uint128(block.timestamp.sub(startTime));
222      for (; i < length && offset >= emissionSchedule[i].startTimeOffset; i++) {}
223      if (i > emissionScheduleIndex) {
224        emissionScheduleIndex = i;
225        _massUpdatePools();
226        rewardsPerSecond = uint256(emissionSchedule[i - 1].rewardsPerSecond);
227      }
```

```
228     }
229   }
```

Listing 3.6: ChefIncentivesController.sol

Impact If the function `setScheduledRewardsPerSecond()` is not invoked for a long time, some promised rewards may not be distributed to the users.

Suggestion The function `setScheduledRewardsPerSecond()` is invoked inside function `claim()` and `_handleActionAfterForToken()`, so the only way the emissions schedule would be skipped would be for no people to interact with the protocol during an emissions epoch.

3.2.5 Changeable Exchange Rate during Migration

Severity Medium

Status Fixed in `Version 5`

Introduced by `Version 1`

Description The contract `Migration` is implemented for users to exchange from `tokenV1` to `tokenV2` with a specified `exchangeRate`. However, during the process of migration, this `exchangeRate` can still be adjusted by the `owner` via the function `setExchangeRate()`.

```
75    /**
76    * @notice Migrate from V1 to V2
77    * @param amount of V1 token
78    */
79   function exchange(uint256 amount) external whenNotPaused {
80       uint256 v1Decimals = tokenV1.decimals();
81       uint256 v2Decimals = tokenV2.decimals();
82
83       uint256 outAmount = amount.mul(1e4).div(exchangeRate).mul(10**v2Decimals).div(10**v1Decimals
             );
84       tokenV1.safeTransferFrom(_msgSender(), address(this), amount);
85       tokenV2.safeTransfer(_msgSender(), outAmount);
86
87       emit Migrate(_msgSender(), amount, outAmount);
88   }
```

Listing 3.7: Migration.sol

Impact It will be unfair to the other users, if the `exchangeRate` is changed during the migration process.

Suggestion Once the migration starts, the `exchangeRate` should be fixed.

3.2.6 Improper Implementation of _transfer() (I)

Severity High

Status Fixed in `Version 7`

Introduced by `Version 1`

Description In contract `IncentivizedERC20`, the function `_transfer()` does not consider the situation that the `sender` and the `recipient` can be the same account (so-called self transfer). Specifically, if the

`sender` equals to the `recipient`, the `sender`'s balance will be overwritten when updating the `recipient`'s balance. In this case, the hacker is able to increase his/her own balance infinitely by transferring to his/her own account repeatedly.

```
176   function _transfer(
177       address sender,
178       address recipient,
179       uint256 amount
180     ) internal virtual {
181       require(sender != address(0), 'ERC20: transfer from the zero address');
182       require(recipient != address(0), 'ERC20: transfer to the zero address');
183
184       _beforeTokenTransfer(sender, recipient, amount);
185
186       uint256 senderBalance = _balances[sender].sub(amount, 'ERC20: transfer amount exceeds
              balance');
187       uint256 recipientBalance = _balances[recipient].add(amount);
188
189       if (address(_getIncentivesController()) != address(0)) {
190         // uint256 currentTotalSupply = _totalSupply;
191         _getIncentivesController().handleActionBefore(sender);
192         if (sender != recipient) {
193           _getIncentivesController().handleActionBefore(recipient);
194         }
195       }
196
197       _balances[sender] = senderBalance;
198       _balances[recipient] = recipientBalance;
199
200       if (address(_getIncentivesController()) != address(0)) {
201         uint256 currentTotalSupply = _totalSupply;
202         _getIncentivesController().handleActionAfter(sender, senderBalance, currentTotalSupply);
203         if (sender != recipient) {
204           _getIncentivesController().handleActionAfter(recipient, recipientBalance,
                  currentTotalSupply);
205         }
206       }
207     }
```

Listing 3.8: IncentivizedERC20.sol

Impact Tokens can be minted infinitely.

Suggestion Implement the function `_transfer()` properly. For example, the standard `_transfer()` implementation of `ERC20` in `OpenZeppelin`.

```
1 _balances[sender] = _balances[sender].sub(amount, 'ERC20: transfer amount exceeds balance');
2 _balances[recipient] = _balances[recipient].add(amount);
```

Listing 3.9: ERC20.sol in OpenZeppelin

3.2.7 Lack of Check on Period in UniV2TwapOracle

Severity Low

Status Fixed in Version 9

Introduced by Version 1

Description In the contract UniV2TwapOracle, the attribute _period is not validated in the function initialize() and setPeriod().

```
35    function initialize(
36        address _pair,
37        address _rdnt,
38        address _ethChainlinkFeed,
39        uint _period,
40        uint _consultLeniency,
41        bool _allowStaleConsults
42    ) external initializer {
43        __Ownable_init();
44
45        pair = IUniswapV2Pair(_pair);
46        token0 = pair.token0();
47        token1 = pair.token1();
48        price0CumulativeLast = pair.price0CumulativeLast(); // Fetch the current accumulated price
              value (1 / 0)
49        price1CumulativeLast = pair.price1CumulativeLast(); // Fetch the current accumulated price
              value (0 / 1)
50        uint112 reserve0;
51        uint112 reserve1;
52        (reserve0, reserve1, blockTimestampLast) = pair.getReserves();
53        require(reserve0 != 0 && reserve1 != 0, 'UniswapPairOracle: NO_RESERVES'); // Ensure that
              there's liquidity in the pair
54
55        PERIOD = _period;
56        CONSULT_LENIENCY = _consultLeniency;
57        ALLOW_STALE_CONSULTS = _allowStaleConsults;
58
59        baseInitialize(_rdnt, _ethChainlinkFeed);
60    }
61
62    function setPeriod(uint _period) external onlyOwner {
63        PERIOD = _period;
64    }
```

Listing 3.10: UniV2TwapOracle.sol

Impact In this case, the oracle can return unexpected value if the _period is too small.

Suggestion Set a minimum limit on the _period in the function initialize and setPeriod.

3.2.8 Non-Refundable Dust Tokens

Severity Medium

Status Fixed in Version 5

Introduced by Version 1

Description In contract UniswapPoolHelper, the function zapWETH() is designed to help the user convert WETH tokens to LP tokens. It will invoke the function addLiquidityWETHOnly() to add liquidity in the pool for

LP tokens. In this process, there may exist dust tokens which should be returned back to users. However, the UniswapPoolHelper doesn't implement such functionality to handle these dust tokens.

```
83    function zapWETH(uint256 amount)
84    public
85    returns (uint256 liquidity)
86{
87    IWETH WETH = IWETH(wethAddr);
88    WETH.transferFrom(msg.sender, address(liquidityZap), amount);
89    liquidity = liquidityZap.addLiquidityWETHOnly(amount, address(this));
90    IERC20 lp = IERC20(lpTokenAddr);
91
92    liquidity = lp.balanceOf(address(this));
93    lp.safeTransfer(msg.sender, liquidity);
94}
```

Listing 3.11: UniswapPoolHelper.sol

Impact The dust tokens will remain in the contract, which can be extracted by others via the function zapTokens(0,0).

Suggestion Implement the function to return dust tokens after adding liquidity.

3.2.9 Improper Implementation of _transfer() (II)

Severity Medium

Status Fixed in Version 9

Introduced by Version 7

Description In contract IncentivizedERC20, the function _transfer() will invoke the function handleActionAfter() to update the status of the user in the contract ChefIncentivesController accordingly. However, the parameter senderBalance will not be updated if the sender equals the recipient, which is incorrect.

```
176    function _transfer(
177        address sender,
178        address recipient,
179        uint256 amount
180      ) internal virtual {
181        require(sender != address(0), 'ERC20: transfer from the zero address');
182        require(recipient != address(0), 'ERC20: transfer to the zero address');
183
184        _beforeTokenTransfer(sender, recipient, amount);
185
186        uint256 senderBalance = _balances[sender].sub(amount, 'ERC20: transfer amount exceeds
               balance');
187
188        if (address(_getIncentivesController()) != address(0)) {
189          // uint256 currentTotalSupply = _totalSupply;
190          _getIncentivesController().handleActionBefore(sender);
191          if (sender != recipient) {
192            _getIncentivesController().handleActionBefore(recipient);
193          }
```

```
194         }
195
196         _balances[sender] = senderBalance;
197         uint256 recipientBalance = _balances[recipient].add(amount);
198         _balances[recipient] = recipientBalance;
199
200         if (address(_getIncentivesController()) != address(0)) {
201           uint256 currentTotalSupply = _totalSupply;
202           _getIncentivesController().handleActionAfter(sender, senderBalance, currentTotalSupply);
203           if (sender != recipient) {
204             _getIncentivesController().handleActionAfter(recipient, recipientBalance,
                  currentTotalSupply);
205           }
206         }
207       }
```

Listing 3.12: IncentivizedERC20.sol

Impact When users transfer to themselves, their state in contract `ChefIncentivesController` will not be updated properly, which will bring further issues for the rewards.

Suggestion Correct the `senderBalance` in the function `handleActionAfter()`.

3.2.10 Manipulatable Compound Rewards

Severity Medium

Status Fixed in `Version 10`

Introduced by `Version 5`

Description In `MFDPlus` contract, the function `_convertPendingRewardsToWeth()` swaps the user's rewards to `WETH` through the `Uniswap` router for relocking. However, there is no slippage check after the swapping.

```
385     IERC20(underlying).safeApprove(uniRouter, removedAmount);
386     uint256[] memory amounts = IUniswapV2Router02(uniRouter)
387     .swapExactTokensForTokens(
388         removedAmount,
389         0, // slippage handled after this function
390         mfdHelper.getRewardToBaseRoute(underlying),
391         address(this),
392         block.timestamp + 10
393     );
```

Listing 3.13: MFDPlus.sol

Impact The attacker can front-run the transaction to manipulate the price and gain the profit.

Suggestion Add the slippage check in function `claimCompound()`.

3.2.11 Lack of Access Control in setLeverager()

Severity Medium

Status Fixed in Version 9

Introduced by Version 1

Description Function setLeverager() in the contract LendingPool has no access control.

```
904    uint256[] memory amounts = IUniswapV2Router02(uniRouter)
905    .swapExactTokensForTokens(
906        removedAmount,
907        0, // slippage handled after this function
908        mfdHelper.getRewardToBaseRoute(underlying),
909        address(this),
910        block.timestamp + 10
911     );
```

Listing 3.14: LendingPool.sol

Impact If the leverager was not set at the beginning, an attacker could set the leverager to any address, thereby gaining control over the logic of the function depositWithAutoDLP().

Suggestion Set the leverager in the function initialize() or add the access control for function setLeverager().

3.2.12 No Slippage Check in addLiquidityWETHOnly()

Severity Medium

Status Confirmed

Introduced by Version 1

Description The user can use either borrowed WETH tokens (or his/her own ETH tokens) or vesting RDNT tokens in MFD contracts to get LP tokens (i.e., WETH-RDNT).

However, when adding the liquidity to the pool, the calculation of the required tokens is based on the amount of reserves in the pool, which can be manipulated. In this case, if the user only has WETH tokens, the function addLiquidityWETHOnly() will be invoked to swap half of the WETH tokens to RDNT tokens in the unbalanced pool without checking slippage.

```
92function addLiquidityWETHOnly(uint256 _amount, address payable to)
93    public
94    returns (uint256 liquidity)
95{
96    require(to != address(0), "LiquidityZAP: Invalid address");
97    uint256 buyAmount = _amount.div(2);
98    require(buyAmount > 0, "LiquidityZAP: Insufficient ETH amount");
99
100   (uint256 reserveWeth, uint256 reserveTokens) = getPairReserves();
101   uint256 outTokens = UniswapV2Library.getAmountOut(
102       buyAmount,
103       reserveWeth,
104       reserveTokens
105   );
106
107   _WETH.transfer(_tokenWETHPair, buyAmount);
108
```

```
109     (address token0, address token1) = UniswapV2Library.sortTokens(
110         address(_WETH),
111         _token
112     );
113     IUniswapV2Pair(_tokenWETHPair).swap(
114         _token == token0 ? outTokens : 0,
115         _token == token1 ? outTokens : 0,
116         address(this),
117         ""
118     );
119
120     return _addLiquidity(outTokens, buyAmount, to);
121 }
```

Listing 3.15: LiquidityZap.sol

```
43     function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure returns (
           uint amountOut) {
44         require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
45         require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
46         uint amountInWithFee = amountIn.mul(997);
47         uint numerator = amountInWithFee.mul(reserveOut);
48         uint denominator = reserveIn.mul(1000).add(amountInWithFee);
49         amountOut = numerator / denominator;
50     }
```

Listing 3.16: UniswapV2Library.sol

Impact The attacker can front-run the transaction to manipulate the price and gain the profit.

Suggestion Check slippage in the function `addLiquidityWETHOnly()` or ensure it can only be invoked by `UniswapPoolHelper`.

3.2.13 Lack of Check of borrowRatio in loopETH()

Severity Low

Status Fixed in `Version 10`

Introduced by `Version 1`

Description Function `loopETH()` is used for leverage borrowing and receives a parameter `borrowRatio` to specify the borrow ratio. However, the `borrowRatio` is not checked before the loop starts.

```
212     function loopETH(
213         uint256 interestRateMode,
214         uint256 borrowRatio,
215         uint256 loopCount
216     ) external payable {
217         uint16 referralCode = 0;
218         uint256 amount = msg.value;
219         if (IERC20(address(weth)).allowance(address(this), address(lendingPool)) == 0) {
220             IERC20(address(weth)).safeApprove(address(lendingPool), type(uint256).max);
221         }
222         if (IERC20(address(weth)).allowance(address(this), address(treasury)) == 0) {
```

```
223             IERC20(address(weth)).safeApprove(treasury, type(uint256).max);
224         }
225
226         uint256 fee = amount.mul(feePercent).div(RATIO_DIVISOR);
227         _safeTransferETH(treasury, fee);
228
229         amount = amount.sub(fee);
230
231         weth.deposit{value: amount}();
232         lendingPool.deposit(address(weth), amount, msg.sender, referralCode);
233
234         for (uint256 i = 0; i < loopCount; i += 1) {
235             amount = amount.mul(borrowRatio).div(RATIO_DIVISOR);
236             lendingPool.borrow(address(weth), amount, interestRateMode, referralCode, msg.sender);
237             weth.withdraw(amount);
238
239             fee = amount.mul(feePercent).div(RATIO_DIVISOR);
240             _safeTransferETH(treasury, fee);
241
242             weth.deposit{value: amount.sub(fee)}();
243             lendingPool.deposit(address(weth), amount.sub(fee), msg.sender, referralCode);
244         }
245
246         zapWETHWithBorrow(wethToZap(msg.sender), msg.sender);
247     }
```

Listing 3.17: Leverager.sol

Impact The `borrowRatio` may be higher than `RATIO_DIVISOR` which is inconsistent with the original design.

Suggestion Make sure that `borrowRatio` is less or equal to `RATIO_DIVISOR`.

3.2.14 Lack of Check of Length between assets and poolIDs in setPoolIDs()

Severity Low

Status Fixed in `Version 10`

Introduced by `Version 1`

Description The function `setPoolIDs()` allows the owner to set different poolIDs for different assets. However, the lengths of these two arrays are not checked to be equal.

```
158     // Set pool ids of assets
159     function setPoolIDs(address[] memory assets, uint256[] memory poolIDs) external onlyOwner {
160         for (uint256 i = 0; i < assets.length; i += 1) {
161             poolIdPerChain[assets[i]] = poolIDs[i];
162         }
163         emit PoolIDsUpdated(assets, poolIDs);
164     }
```

Listing 3.18: StarBorrow.sol

Impact The `assets` will not be assigned to correct `poolIDs`.

Suggestion Make sure the lengths of `assets` and `poolIDs` are equal.

3.2.15 Lack of mint Privilege Revoke in addBountyContract()

Severity Low

Status Confirmed

Introduced by Version 1

Description Function addBountyContract() is used to set the new BountyManager. However, the original bounty contract still holds the mint privilege, which is against the original design.

```
250    function addBountyContract(address _bounty) external onlyOwner {
251        BountyManager = _bounty;
252        minters[_bounty] = true;
253    }
```

Listing 3.19: Leverager.sol

Impact The deprecated BountyManager still has mint privileges.

Suggestion Revoke the mint privilege of origin BountyManager contract.

Feedback The function addBountyContract will only be called once to initialize the BountyManager.

3.2.16 Minters Can Only be Assigned Once

Severity Low

Status Confirmed

Introduced by Version 1

Description The minters is used to record those who have the permission to access the function mint() and addReward(). However, when one of the minters (e.g., the contract ChefIncentivesController) is updated, the outdated minters can not be removed.

```
242    function setMinters(address[] memory _minters) external onlyOwner {
243        require(!mintersAreSet);
244        for (uint256 i; i < _minters.length; i++) {
245            minters[_minters[i]] = true;
246        }
247        mintersAreSet = true;
248    }
```

Listing 3.20: MultiFeeDistribution.sol

Impact The outdated minters can not be removed when they are upgraded.

Suggestion Implement a privileged function to modify minters.

Feedback Because the BountyManager, ChefIncentivesController and MultiFeeDistribution will be upgradable, so minters always keep the same proxy address.

3.3 Additional Recommendation

3.3.1 Gas Optimization (zapVestingToLp() in Mfd)

Status Fixed in Version 10

Introduced by Version 1

Description The function zapVestingToLp() can only be invoked by the contract LockZap to transfer the locked earning of the user out. It iterates the earnings array of the user starting from the index 0, and checks whether the unlockTime is larger than the current timestamp. If so, this earning will be removed from the array and transferred out. However, since the unlockTime in the array is increasing with the index, it will be more efficient to start the iteration from the end of array to the beginning. If the unlockTime is smaller than the current timestamp, the loop can be broken.

```
1204    function zapVestingToLp(address _user)
1205        external
1206        override
1207        returns (uint256 zapped)
1208    {
1209        require(msg.sender == lockZap);
1210
1211        LockedBalance[] storage earnings = userEarnings[_user];
1212        uint256 length = earnings.length;
1213
1214        for (uint256 i = 0; i < length; ) {
1215            // only vesting, so only look at currently locked items
1216            if (earnings[i].unlockTime > block.timestamp) {
1217                zapped = zapped.add(earnings[i].amount);
1218                // remove + shift array size
1219                earnings[i] = earnings[earnings.length - 1];
1220                earnings.pop();
1221                length = length.sub(1);
1222            } else {
1223                i = i.add(1);
1224            }
1225        }
1226
1227        rdntToken.safeTransfer(lockZap, zapped);
1228
1229        Balances storage bal = balances[_user];
1230        bal.earned = bal.earned.sub(zapped);
1231        bal.total = bal.total.sub(zapped);
1232
1233        return zapped;
1234    }
```

Listing 3.21: MultiFeeDistribution.sol

Suggestion Start the iteration from the end of earnings to the beginning. If the unlockTime is smaller than the current timestamp, the loop can be broken.

3.3.2 Non-empty Bounty Reserve in BountyManager

Status Fixed in Version 10

Introduced by Version 1

Description In function _sendBounty(), if there are not enough RDNT tokens for the transfer in the contract

BountyManager, the event BountyReseveEmpty() will be emitted, and the contract will be paused. However, it's possible that there are still some RDNT tokens left, which is inconsistent with the emitted event.

```
354  function _sendBounty(address _to, uint256 _amount)
355    internal
356    returns (uint256)
357  {
358    if (_amount == 0) {
359      return 0;
360    }
361
362    uint256 bountyReserve = IERC20(rdnt).balanceOf(address(this));
363    if(_amount > bountyReserve) {
364      emit BountyReserveEmpty(bountyReserve);
365      _pause();
366    } else {
367      IERC20(rdnt).safeTransfer(address(mfd), _amount);
368      IMFDPlus(mfd).mint(_to, _amount, true);
369      return _amount;
370    }
371  }
```

Listing 3.22: BountyManager.sol

Suggestion Transfer the left RDNT tokens out even if it's not enough.

3.3.3 Inconsistent Naming in requiredUsdValue()

Status Confirmed

Introduced by Version 1

Description The function requiredUsdValue() is used to check the required locked value of the user who wants to be qualified to earn rewards by holding RTokens. The calculation is based on the collateral value of the user, which is returned from the function getUserAccountData(). However, the returned value is named as totalCollateralETH, which is inconsistent with that in the function requiredUsdValue() (i.e., totalCollateralUSD).

Suggestion Standardize the naming conventions of functions with the right token name. For example, rename requiredUsdValue() to requiredEthValue().

Feedback We'd rather keep the AAVE contracts as similar as possible so we didn't update the name.

3.4 Notes

3.4.1 Depreciated MFDPlus

Status Confirmed

Introduced by version 10

Description The contract MFDPlus is no longer used. The logic of compounding is moved into the contract AutoCompounder and other logic is moved into the contract MiddleFeeDistribution.

Chapter 4 Appendix

4.1 Automated Static Security Testing Results

Table 4.1: Automated Static Security Testing Results. Found indidates the number of issues reported by the tools. FP means the number of false positives after our manual verification.

ID	Detector	Description	Impact	Found	FP	Result
1	arbitrary-send-erc20	Calling `transferFrom` with arbitrary `from`	High	1	1	Passed
2	array-by-reference	Modifying storage array by value	High	0	0	Passed
3	incorrect-shift	Incorrect order of parameters in a shift instruction	High	0	0	Passed
4	multiple-constructors	Multiple constructor schemes	High	0	0	Passed
5	name-reused	Reusing contract's name	High	0	0	Passed
6	protected-vars	Modifying variables directly without access control	High	0	0	Passed
7	rtlo	Using Right-To-Left-Override control character	High	0	0	Passed
8	shadowing-state	State variables shadowing	High	1	1	Passed
9	suicidal	Functions allowing anyone to destruct the contract	High	0	0	Passed
10	uninitialized-state	Uninitialized state variables	High	3	3	Passed
11	uninitialized-storage	Uninitialized storage variables	High	0	0	Passed
12	unprotected-upgrade	Unprotected upgradeable contract	High	1	1	Passed
13	arbitrary-send-erc20-permit	`transferFrom` uses arbitrary from with `permit`	High	0	0	Passed
14	arbitrary-send-eth	Functions that send Ether to arbitrary destinations	High	0	0	Passed
15	controlled-array-length	Tainted array length assignment	High	0	0	Passed
16	controlled-delegatecall	Controlled `delegatecall` destination	High	0	0	Passed
17	delegatecall-loop	Payable functions using `delegatecall` inside a loop	High	0	0	Passed
18	msg-value-loop	Using `msg.value` inside a loop	High	0	0	Passed

ID	Detector	Description	Impact	Found	FP	Result
19	reentrancy-eth	Reentrancy vulnerabilities (theft of ethers)	High	5	5	Passed
20	storage-array	Signed storage integer array compiler bug	High	0	0	Passed
21	unchecked-transfer	Unchecked tokens `transfer`	High	12	12	Passed
22	weak-prng	Weak PRNG	High	0	0	Passed
23	domain-separator-collision	Detects ERC20 tokens that have a function whose signature collides with EIP-2612's `DOMAIN_SEPARATOR()`	Medium	0	0	Passed
24	enum-conversion	Detects dangerous enum conversion	Medium	0	0	Passed
25	erc20-interface	Incorrect ERC20 interfaces	Medium	0	0	Passed
26	erc721-interface	Incorrect ERC721 interfaces	Medium	0	0	Passed
27	incorrect-equality	Dangerous strict equalities	Medium	23	23	Passed
28	locked-ether	Contracts that lock ether	Medium	1	1	Passed
29	mapping-deletion	Deletion on mapping containing a structure	Medium	0	0	Passed
30	shadowing-abstract	State variables shadowing from abstract contracts	Medium	0	0	Passed
31	tautology	Tautology or contradiction	Medium	0	0	Passed
32	write-after-write	Unused write	Medium	3	3	Passed
33	boolean-cst	Misuse of Boolean constant	Medium	0	0	Passed
34	constant-function-asm	Constant functions using assembly code	Medium	0	0	Passed
35	constant-function-state	Constant functions changing the state	Medium	0	0	Passed
36	divide-before-multiply	Imprecise arithmetic operations order	Medium	20	20	Passed
37	reentrancy-no-eth	Reentrancy vulnerabilities (no theft of ethers)	Medium	12	12	Passed
38	reused-constructor	Reused base constructor	Medium	0	0	Passed
39	tx-origin	Dangerous usage of `tx.origin`	Medium	1	1	Passed
40	unchecked-lowlevel	Unchecked low-level calls	Medium	0	0	Passed
41	unchecked-send	Unchecked `send`	Medium	0	0	Passed
42	uninitialized-local	Uninitialized local variables	Medium	33	33	Passed
43	unused-return	Unused return values	Medium	19	19	Passed

4.2 Automated Dynamic Security Testing Results

Table 4.2: Tested Properties for Lending related Logic

ID	Property	Result
1	Calling `deposit` never leads to a decrease of `onBehalfOf`'s `RToken` amount	Passed
2	Calling `withdraw` never leads to an increase of `msg.sender`'s `RToken` amount	Passed
3	Calling `borrow` with stable interest rate mode never leads to a decrease of `onBehalfOf`'s `StableDebtToken`.	Passed
4	Calling `borrow` with variable interest rate mode never leads to a decrease of `onBehalfOf`'s `VariableDebtToken`.	Passed
5	Calling `borrow` with `onBehalfOf` that does not equal to `msg.sender` never leads to an increase of `msg.sender`'s borrow allowance.	Passed
6	Calling `repay` with stable interest rate mode never leads to an increase of `onBehalfOf`'s `StableDebtToken`.	Passed
7	Calling `repay` with variable interest rate mode never leads to an increase of `onBehalfOf`'s `VariableDebtToken`.	Passed
8	`liquidityIndex` will never decrease.	Passed
9	`liquidityIndex` will remain constant within the same block.	Passed
10	`variableBorrowIndex` will never decrease.	Passed
11	`variableBorrowIndex` will remain constant within the same block.	Passed
12	Decreasing collateral amounts will never lead to health factor less than 1.	Passed
13	Increasing borrowing amounts will never lead to health factor less than 1.	Passed

Table 4.3: Tested Properties for Staking related Logic

ID	Property	Result
1	User's `total` balance always equals the sum of `locked` balance, `unlocked` balance and `earned` balance.	Passed
2	User's `locked` balance always equals the sum of `userLocks amount`	Passed
3	User's `lockedWithMultiplier` balance always equals the sum of `userLocks amount` times `userLocks multiplier`	Passed
4	`lockedSupply` always equals the sum of users' `locked` balance	Passed
5	`lockedSupplyWithMultiplier` always equals the sum of users' `lockedWithMultiplier` balance	Passed
6	`rewardPerTokenStored` never decreases.	Passed
7	`rewardPerTokenStored` will remain constant within the same block.	Passed
8	`totalSupply` always equals the sum of users' `amount`	Passed
9	`accRewardPerShare` never decreases.	Passed
10	`accRewardPerShare` will remain constant within the same block.	Passed

Table 4.4: Tested Properties for Other Features

ID	Property	Result
1	`WETH` and `RDNT` balance of the contract `LockedZap` will always be zero.	Passed
2	`WETH` and `RDNT` balance of the contract `LiquidityZap` will always be zero.	Passed
3	`WETH` and `RDNT` balance of the contract `BalancerPoolHelper` will always be zero.	Passed
4	`WETH` and `RDNT` balance of the contract `UniswapPoolHelper` will always be zero.	Passed
5	Calling `loop` will always lead to user eligible for rewards	Passed
6	Calling `loopETH` will always lead to user eligible for rewards	Passed
7	Calling `executeBounty` with `_execute` equals `false` will never lead to storage change.	Passed
8	Calling `transfer` with `sender` equals to `receiver` never leads to balance change.	Failed in Version 1. Passed in Version 7