Radiant Security Review

Pashov Audit Group

Conducted by: HickupHH3, SpicyMeatball

February 28th 2024 - March 1st 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **v2-core** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Radiant

Radiant is an omnichain money market where users can deposit any major asset on any major chain and borrow various supported assets across multiple chains. Radiant v2 migrates the current ERC-20 RDNT token to the LayerZero OFT format. Moreover, LockZap and Compounder contracts now utilize UniswapV3 instead of UniswapV2 for their swaps.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - e7271904c8e951bc73c1235872b0ec0ab5bb27c0

fixes review commit hash - 11447f34f417ad0c7b2401d26b3725ad2f147c38

Scope

The following smart contracts were in scope of the audit:

- `Compounder`
- `LockZap`

7. Executive Summary

Over the course of the security review, HickupHH3, SpicyMeatball engaged with Radiant to review Radiant. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Radiant
Repository	https://github.com/radiant-capital/v2-core
Date	February 28th 2024 - March 1st 2024
Protocol Type	Omnichain money market

Findings Count

Severity	**Amount**
Critical	2
High	1
Low	5
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect token order passed for WETH -> RDNT swaps	Critical	Resolved
[C-02]	_wethToRdnt() will certainly revert as rdntOut is always zero	Critical	Resolved
[H-01]	Incorrect quoteWETH() implementation (out-of-scope)	High	Resolved
[L-01]	Unused code	Low	Resolved
[L-02]	Two WETH addresses are stored in the same contract	Low	Resolved
[L-03]	LockZap.quoteFromToken() uses incorrect API for estimations	Low	Resolved
[L-04]	UniswapPoolHelper doesn't inherit IUniswapPoolHelper	Low	Resolved
[L-05]	Routes can be more strictly checked	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect token order passed for WETH -> RDNT swaps

Severity

Impact: High

Likelihood: High

Description

The `BalancerPoolHelper.swapWethToRdnt()` is intended to swap WETH for RDNT. However, the input and output tokens are in reverse:

```
_swap(
  RDNT_ADDRESS,
  REAL_WETH_ADDR,
  _wethAmount,
  _minAmountOut,
  WETH_RDNT_POOL_ID,
  msg.sender
);
```

resulting in an attempted RDNT -> WETH swap instead.

Recommendations

```
- _swap
- (RDNT_ADDRESS, REAL_WETH_ADDR, _wethAmount, _minAmountOut, WETH_RDNT_POOL_ID, msg.se
+ _swap
+ (wethAddr, RDNT_ADDRESS, _wethAmount, _minAmountOut, WETH_RDNT_POOL_ID, msg.sender);
```

[C-02] `_wethToRdnt()` will certainly revert as `rdntOut` is always zero

Severity

Impact: High

Likelihood: High

Description

`BalancerPoolHelper._swap()` always returns `0` because it lacks a return statement or return parameter assignment. Hence, `rdntOut = poolHelper_.swapWethToRdnt(_wethIn, 0);` will always be 0, causing the slippage check to fail.

Recommendations

```
- IVault(vaultAddr).swap(singleSwap, funds, _minAmountOut, block.timestamp);
+ return IVault(vaultAddr).swap
+ (singleSwap, funds, _minAmountOut, block.timestamp);
```

8.2. High Findings

[H-01] Incorrect `quoteWETH()` implementation (out-of-scope)

Severity

Impact: High

Likelihood: Medium

Description

`UniswapPoolHelper.quoteWETH()` is used to calculate the WBNB (denoted as WETH) required for LP-ing into the WBNB-RDNT pool on BSC. There are 2 issues with its implementation:

`neededWeth` is derived from the wrong reserve

```
uint256 weth = lpToken.token0() != address(rdntAddr) ? reserve0 : reserve1;
uint256 rdnt = lpToken.token0() == address(rdntAddr) ? reserve0 : reserve1;
uint256 lpTokenSupply = lpToken.totalSupply();

uint256 neededWeth = (rdnt * lpAmount) / lpTokenSupply;
```

The `neededWeth` should be using `weth` instead of `rdnt`.

Required amounts are derived from pool amounts before swap, not after

Doing 1-sided liquidity is akin to swapping half of the amount for the other token, then adding liquidity with the remaining half and swapped amounts.

The implementation uses the pool reserves before the swap to calculate the amounts needed, but it should use the altered reserves from the swap where `weth` increases and `rdnt` decreases.

Recommendations

The suggested implementation is below.

```
uint256 neededRdnt = (lpAmount * rdnt) / (lpAmount + lpTokenSupply);
uint256 neededRdntInWeth = router.getAmountIn(neededRdnt, weth, rdnt);
uint256 neededWeth = (weth - neededRdntInWeth) * lpAmount / lpTokenSupply;
return neededWeth + neededRdntInWeth;
```

8.3. Low Findings

[L-01] Unused code

`LockZap.sol`

```
/// @notice Swap uniswap v2 routes from token0 to token1
        mapping(address => mapping(address => address[])) internal _uniV2Route;
```

`BalancerPoolHelper.sol`

```
/**
         * @notice Swaps WETH to RDNT
         * @param _wethAmount the amount of RDNT to sell
         * @param _minAmountOut the minimum RDNT amount to accept without reverting
         */
        function swapWethToRdnt
   (uint256 _wethAmount, uint256 _minAmountOut) external returns (uint256) {
                if (_wethAmount == 0) revert ZeroAmount();

>>              uint256 usdcBalanceAfter = IERC20(USDC_ADDRESS).balanceOf(address(this
                return _swap(
    RDNT_ADDRESS,
    REAL_WETH_ADDR,
    _wethAmount,
    _minAmountOut,
    WETH_RDNT_POOL_ID,
    msg.sender
  );
        }
```

[L-02] Two WETH addresses are stored in the same contract

`BalancerPoolHelper.sol` stores two WETH address values, one is hardcoded

```
address public constant REAL_WETH_ADDR = address
  (0x82aF49447D8a07e3bd95BD0d56f35241523fBab1);
```

and another is specified on the initialization

```
function initialize(
                address _inTokenAddr,
                address _outTokenAddr,
                address _wethAddr,
                address _vault,
                IWeightedPoolFactory _poolFactory
        ) external initializer {
                ...
                __Ownable_init();
                inTokenAddr = _inTokenAddr;
                outTokenAddr = _outTokenAddr;
>>              wethAddr = _wethAddr;
                vaultAddr = _vault;
                poolFactory = _poolFactory;
        }
```

Both are used in the contract.

```
function swapWethToRdnt
   (uint256 _wethAmount, uint256 _minAmountOut) external returns (uint256) {
                if (_wethAmount == 0) revert ZeroAmount();

                uint256 usdcBalanceAfter = IERC20(USDC_ADDRESS).balanceOf(address(this
>>              return _swap(
  RDNT_ADDRESS,
  REAL_WETH_ADDR,
  _wethAmount,
  _minAmountOut,
  WETH_RDNT_POOL_ID,
  msg.sender
);
        }

        function quoteWethToRdnt(uint256 _wethAmount) external view returns (uint256)
                if (_wethAmount == 0) revert ZeroAmount();
>>              return _quote(wethAddr, RDNT_ADDRESS, _wethAmount, WETH_RDNT_POOL_ID);
        }
```

Consider leaving only one WETH address variable/constant.

[L-03] `LockZap.quoteFromToken()` uses incorrect API for estimations

Users are able to specify another `token` to be paired with RDNT tokens for adding liquidity. `quoteFromToken(address _token, uint256 _amount)` assists with estimating the `token` amount needed for this pairing.

```
if (_token != weth_) {
  uint256 wethAmount = poolHelper.quoteFromToken(_amount);
  return _quoteUniswap(weth_, _token, wethAmount);
}

function _quoteUniswap(
  address_tokenIn,
  address_tokenOut,
  uint256_amountIn
) internal view returns (uint256
  bytes memory route = _uniV3Route[_tokenIn][_tokenOut];
  uint256 amountOut = uniV3Quoter.quoteExactInput(route, _amountIn);
  return amountOut;
}
```

The quoted amount is derived from swapping the `wethAmount` to `_token`, but in practice, the swap is done the other way round (zap `_token` to `WETH`, then pair with `RDNT _amount`), resulting in a smaller estimate than the actual required amount.

Recommend switching to `quoteExactOutput()`.

```
- uniV3Quoter.quoteExactInput(route, _amountIn);
+ uniV3Quoter.quoteExactOutput
+ (route, _amountIn); // consider renaming to _amountOut
```

Note that the `route` is unchanged because the expected path is to be encoded in reverse (= swap exact input)

[L-04] `UniswapPoolHelper` doesn't inherit `IUniswapPoolHelper`

`UniswapPoolHelper` is expected to conform to the underlying `IPoolHelper`, but doesn't inherit it. Recommend doing so to ensure compatibility with other contracts using it.

[L-05] Routes can be more strictly checked

```
if (_route.length < MIN_UNIV3_ROUTE_LENGTH) revert WrongRoute
  (_tokenIn, _tokenOut);
```

The only check performed when setting a new route is that it must be `>= MIN_UNIV3_ROUTE_LENGTH`. However, there aren't checks to ensure that it's properly encoded.

There are 2 possible checks to validate the route:

1. ensure `(_route.length - 20) % 23) == 0` <=> `(_route.length - ADDR_SIZE) % NEXT_OFFSET) == 0`
2. call `uniV3Quoter.quoteExactInput(_route, someAmountIn) != 0;` but this is reliant on `uniV3Quoter` being set properly and having sufficient liquidity for the pools to be used