BLOCKSEC

Security Audit

Report for Radiant UniswapV3 Helpers

Date: June 28, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Radiant Capital
Target	Radiant UniswapV3 Helpers

Version History

Version	Date	Description
1.0	June 28, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-12 08:50:11+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec 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 successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on the Radiant UniswapV3 Helpers of Radiant Capital [1]. Radiant UniswapV3 Helpers allows users to stake their RDNT and WETH tokens into the Uniswap V3-like pools, including Uniswap V3 and Velodrome.

Please note that only the `UniV3PoolHelper.sol` and the `UniV3TokenizedLp.sol` located within the `contracts/main/radiant/zap/helpers/` folder in the repository are included in the scope of this audit. Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Radiant UniswapV3 Helpers	`Version 1`	`aeb485411517983a20fb3a285c6129b4d2ec5e4d`
	`Version 2`	`b235c538ef72bd8ee8445ac57337cc0c8cd761eb`
	`Version 3`	`2f6eaadd32dd2d650499ee21e2b75824dd429ebf`

1.2 Disclaimer

This audit 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 audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit 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 one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/radiant-capital/v2-core

The scope of this audit is limited to the code mentioned in Section 1.1. 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.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 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

1.3.2 DeFi Security

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

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

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

1.4 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 [2] and Common Weakness Enumeration [3]. 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.

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

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

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **six** potential security issues. Besides, we have **one** recommendation and **three** notes.

- Medium Risk: 3
- Low Risk: 3
- Recommendation: 1
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Potential precision loss	Software Security	Confirmed
2	Low	Ineffective maximum deposit limit	Software Security	Confirmed
3	Medium	Potential DoS risk	Software Security	Fixed
4	Medium	Lack of checks on the token order	Software Security	Fixed
5	Low	Potential liquidity manipulation in the `autoRebalance` function	DeFi Security	Fixed
6	Medium	Manipulable return value from the `getLpPrice` function	DeFi Security	Fixed
7	-	Remove redundant checks	Recommendation	Fixed
8	-	Potential centralization risks	Note	-
9	-	Assumption on the `UniV3PoolHelper` contract	Note	-
10	-	Dependency on the block timestamp	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential precision loss

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The deposit and withdrawal progress in the `UniV3TokenizedLp` contract may suffer from precision losses.

Specifically, the assets in this contract are divided into two parts: liquidity in the Uniswap V3 pool position (denoted as part A) and the remaining token balances in the contract (denoted as part B). Shares minted for each deposit action are calculated as $amount * totalSupply/(A+B)$. In contrast, the returned amounts for a withdrawal action are calculated as $shares * A/totalSupply + shares * B/totalSupply$. The issue arises because A and B are calculated separately during the

withdrawal process, potentially leading to precision losses due to the summed total being much larger than the individual parts.

```
430   function deposit(
431       uint256 deposit0,
432       uint256 deposit1,
433       address to
434   ) external override nonReentrant returns (uint256 shares) {
435       if (!allowToken0 && deposit0 > 0) {
436           revert UniV3TokenizedLp_Token0NotAllowed();
437       }
438       if (!allowToken1 && deposit1 > 0) {
439           revert UniV3TokenizedLp_Token1NotAllowed();
440       }
441       if (deposit0 == 0 && deposit1 == 0) {
442           revert UniV3TokenizedLp_ZeroValue();
443       }
444       if (deposit0 > deposit0Max || deposit1 > deposit1Max) {
445           revert UniV3TokenizedLp_MoreThanMaxDeposit();
446       }
447       if (to == NULL_ADDRESS) revert UniV3TokenizedLp_ZeroAddress();
448
449       // Updates pending fees in pool state for inclusion when calling `getTotalAmounts()`
450       (uint128 baseLiquidity, , ) = _position(baseLower, baseUpper);
451       if (baseLiquidity > 0) {
452           // See IUniswapV3PoolActions.burn(...) interface docs, this call is used to update state
                  of fees
453           (uint256 burn0, uint256 burn1) = IUniswapV3Pool(pool).burn(baseLower, baseUpper, 0);
454           if (burn0 != 0 || burn1 != 0) {
455               revert UniV3TokenizedLp_UnexpectedBurn();
456           }
457       }
458
459       // Spot price of token1/token0
460       uint256 spotPrice = fetchSpot(token0, token1, PRECISION);
461       // External oracle price of token1/token0
462       uint256 oraclePrice = fetchOracle(token0, token1, PRECISION);
463
464       // If difference between spot and oracle is bigger than `hysteresis`, it
465       // checks the timestamp of the last `observation` at the pool
466       // to confirm if price has been manipulated in this block
467       uint256 delta = (spotPrice > oraclePrice)
468           ? ((spotPrice - oraclePrice) * PRECISION) / spotPrice
469           : ((oraclePrice - spotPrice) * PRECISION) / oraclePrice;
470       if (delta > hysteresis) require(_checkHysteresis(), "try later");
471
472       (uint256 pool0, uint256 pool1) = getTotalAmounts();
473
474       // Price the `deposit0` amount in token1 at oracle price
475       uint256 deposit0PricedInToken1 = (deposit0 * oraclePrice) / PRECISION;
476
477       if (deposit0 > 0) {
478           IERC20(token0).safeTransferFrom(msg.sender, address(this), deposit0);
```

```
479         }
480         if (deposit1 > 0) {
481             IERC20(token1).safeTransferFrom(msg.sender, address(this), deposit1);
482         }
483
484         // Shares in value of token1
485         shares = deposit1 + deposit0PricedInToken1;
486
487         if (totalSupply() != 0) {
488             // Price the pool0 in token1 at oracle price
489             uint256 pool0PricedInToken1 = (pool0 * oraclePrice) / PRECISION;
490             // Compute ratio of total shares to pool AUM in token1
491             shares = (shares * totalSupply()) / (pool0PricedInToken1 + pool1);
492         }
493         _mint(to, shares);
494         emit Deposit(msg.sender, to, shares, deposit0, deposit1);
495     }
```

Listing 2.1: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

```
514     function withdraw(
515         uint256 shares,
516         address to
517     ) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
518         if (shares == 0) revert UniV3TokenizedLp_ZeroValue();
519         if (to == NULL_ADDRESS) revert UniV3TokenizedLp_ZeroAddress();
520
521
522         // Withdraw share amount of liquidity from the Uniswap pool
523         // This call also updates fee state in the pool
524         (uint256 base0, uint256 base1) = _burnLiquidity(
525             baseLower,
526             baseUpper,
527             _liquidityForShares(baseLower, baseUpper, shares),
528             to,
529             false
530         );
531
532         // Compute proportion of unused balances in this contract relative to `shares`
533         // Note: Sending tokens directly to alter the balances of this address will result in a
                loss to the sender-caller.
534         uint256 _totalSupply = totalSupply();
535         uint256 unusedAmount0 = (IERC20(token0).balanceOf(address(this)) * (shares)) / _totalSupply
                ;
536         uint256 unusedAmount1 = (IERC20(token1).balanceOf(address(this)) * (shares)) / _totalSupply
                ;
537         if (unusedAmount0 > 0) IERC20(token0).safeTransfer(to, unusedAmount0);
538         if (unusedAmount1 > 0) IERC20(token1).safeTransfer(to, unusedAmount1);
539
540         amount0 = base0 + unusedAmount0;
541         amount1 = base1 + unusedAmount1;
542
543         _burn(msg.sender, shares);
```

```
544
545         emit Withdraw(msg.sender, to, shares, amount0, amount1);
546     }
```

Listing 2.2: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

Impact Potential precision loss may result in unexpected consequences.

Suggestion Revise the code accordingly.

Feedback from the Project This issue is acknowledged by the team, and we decide not to change the existing code. The lost precision results in losses to the user withdrawing of close to 1 wei unit, in favor of the remaining depositors.

2.1.2 Ineffective maximum deposit limit

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `deposit` function, the checks on the maximum deposit limit are ineffective, as users can bypass them by splitting a single deposit into multiple calls.

```
430     function deposit(
431         uint256 deposit0,
432         uint256 deposit1,
433         address to
434     ) external override nonReentrant returns (uint256 shares) {
435         if (!allowToken0 && deposit0 > 0) {
436             revert UniV3TokenizedLp_Token0NotAllowed();
437         }
438         if (!allowToken1 && deposit1 > 0) {
439             revert UniV3TokenizedLp_Token1NotAllowed();
440         }
441         if (deposit0 == 0 && deposit1 == 0) {
442             revert UniV3TokenizedLp_ZeroValue();
443         }
444         if (deposit0 > deposit0Max || deposit1 > deposit1Max) {
445             revert UniV3TokenizedLp_MoreThanMaxDeposit();
446         }
```

Listing 2.3: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

Impact The maximum deposit limit can be circumvented by splitting deposit actions into multiple calls.

Suggestion Revise the code accordingly.

Feedback from the Project If `deposit0Max` or `deposit1Max` are set to zero, it effectively stops the inflow of a specific type of token, even in further attempts. Setting a non-zero value can help control the proportion of inflow tokens; however, the team acknowledges that in several separate calls the purpose of inflow tokens in expected "max-ratio" can be not as effective.

2.1.3 Potential DoS risk

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 1`

Description There is a potential DoS risk in `UniV3TokenizedLp` contract. Specifically, user-deposited tokens are not immediately added to the Uniswap V3 pool, while the contract burns liquidity from the underlying Uniswap V3 pool immediately upon withdrawals requests. A malicious user can exploit this by repeatedly depositing and withdrawing from the contract, forcing it to burn liquidity into underlying tokens. In the most extreme case, this could leave only minimal liquidity in the contract, effectively reducing the contract's fee revenue.

Impact The DoS attack may reduce the liquidity of the contract's position.

Suggestion Revise the code accordingly.

2.1.4 Lack of checks on the token order

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description When initializing a new `UniV3TokenlizedLp` contract, the `UniV3PoolHelper` contract adds liquidity to the corresponding pool. However, the `initializePool` function lack checks on the token order, which may result in incorrect initialization.

```
112   function initializePool(InitializeParams memory params) public onlyOwner {
113       if (address(tokenizedLpToken) != address(0)) revert AlreadyInitialized();
114
115       (token0, token1) = rdntAddr < weth9Addr ? (rdntAddr, weth9Addr) : (weth9Addr, rdntAddr);
116
117       uint256 initRdntBal = IERC20(rdntAddr).balanceOf(address(this));
118       uint256 initWeth9Bal = IERC20(weth9Addr).balanceOf(address(this));
119       uint160 sqrtPriceX96 = UniV3PoolMath.encodePriceSqrtX96(
120           rdntAddr == token0 ? initRdntBal : initWeth9Bal,
121           rdntAddr == token1 ? initRdntBal : initWeth9Bal
122       );
123
124       if (params.factoryType == FactoryType.UniswapV3) {
125           pool = IUniswapV3Pool(params.uniV3Factory.getPool(token0, token1, DESIRED_FEE));
126           if (address(pool) == address(0)) {
127               // UniswapV3 vanilla factory implementation takes desired fee
128               pool = IUniswapV3Pool(params.uniV3Factory.createPool(token0, token1, DESIRED_FEE));
129               pool.initialize(sqrtPriceX96);
130           }
131       } else if (params.factoryType == FactoryType.Velodrome) {
132           ICLFactory clFactory = ICLFactory(address(params.uniV3Factory));
133           pool = IUniswapV3Pool(clFactory.getPool(rdntAddr, weth9Addr, DESIRED_TICK_SPACING));
134           if (address(pool) == address(0)) {
135               // Velodrome factory implementation takes desired tick spacing and initializes
                      srqtPriceX96
```

```
136             pool = IUniswapV3Pool(clFactory.createPool(rdntAddr, weth9Addr, DESIRED_TICK_SPACING
                    , sqrtPriceX96));
137         }
138     }
139
140     tokenizedLpToken = UniV3TokenizedLp(Clones.cloneDeterministic(params.tokenizedLpImpl, "
            (>'.'<)"));
141
142     {
143         (address oracle0, address oracle1) = rdntAddr == token0
144             ? (params.usdRdntOracle, params.usdWeth9Oracle)
145             : (params.usdWeth9Oracle, params.usdRdntOracle);
146         tokenizedLpToken.initialize(address(pool), true, true, oracle0, oracle1);
147         int24 tickSpacing = pool.tickSpacing();
148         int24 baseLower_ = UniV3PoolMath.roundTick(UniV3PoolMath.MIN_TICK, tickSpacing);
149         int24 baseUpper_ = UniV3PoolMath.roundTick(UniV3PoolMath.MAX_TICK, tickSpacing);
150         tokenizedLpToken.rebalance(baseLower_, baseUpper_, 0);
151
152         IERC20(rdntAddr).forceApprove(address(pool), initRdntBal);
153         IERC20(weth9Addr).forceApprove(address(pool), initWeth9Bal);
154
155         uint128 fullRangeliquidity = UniV3PoolMath.getLiquidityForAmounts(
156             sqrtPriceX96,
157             UniV3PoolMath.MIN_SQRT_RATIO,
158             UniV3PoolMath.MAX_SQRT_RATIO,
159             initRdntBal,
160             initWeth9Bal
161         );
162
163         pool.mint(address(this), baseLower_, baseUpper_, fullRangeliquidity, abi.encode(address(
                this)));
164     }
165
166     tokenizedLpToken.setApprovedRebalancer(address(this), false);
167     tokenizedLpToken.setApprovedRebalancer(msg.sender, true);
168     tokenizedLpToken.setFeeRecipient(msg.sender);
169     tokenizedLpToken.transferOwnership(msg.sender);
170
171     emit TokenizedLpInitialized(address(tokenizedLpToken));
172 }
```

Listing 2.4: contracts/main/radiant/zap/helpers/UniV3PoolHelper.sol

Impact Lack of token order may cause the function to fail.

Suggestion Revise the code accordingly.

2.2 DeFi Security

2.2.1 Potential liquidity manipulation in the `autoRebalance` function

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The `autoRebalance` function in the `UniV3TokenlizedLp` contract may encounter issues during the rebalance process. Specifically, the function first burns the position to withdraw all liquidity and then restructures tokens based on the spot price and oracle price. This mechanism functions effectively under certain conditions:

- The delta surpasses the threshold while the upper and lower ranges remain unchanged.
- The upper and lower ranges adjust when the oracle price moves significantly.

However, if the spot price exceeds the upper or lower range, causing the liquidity position to be filled predominantly with one token, the liquidity to be minted in the final step would be 0. Essentially, this empties the position's liquidity, preventing fee collection from the Uniswap V3 pool until the next proper rebalance.

```
545   function autoRebalance() public nonReentrant {
546       if (baseLower == 0 && baseUpper == 0) revert UniV3TokenizedLp_SetBaseTicksViaRebalanceFirst
              ();
547
548       (uint256 token0Bal, uint256 token1Bal) = _updateAndCollectPositionFees();
549
550       // Get spot and external oracle prices
551       uint256 spotPrice = fetchSpot(token0, token1, PRECISION);
552       uint256 oraclePrice = fetchOracle(token0, token1, PRECISION);
553
554       // Check if difference between spot and oraclePrice is too big
555       uint256 delta = (spotPrice > oraclePrice)
556           ? ((spotPrice - oraclePrice) * PRECISION) / oraclePrice
557           : ((oraclePrice - spotPrice) * PRECISION) / oraclePrice;
558
559       // Calculate the new baseLower and baseUpper ticks. It is required to encode the price into
              sqrtPriceX96
560       int24 baseLower_ = UniV3PoolMath.roundTick(
561           UniV3PoolMath.getTickAtSqrtRatio(
562               UniV3PoolMath.encodePriceSqrtX96(
563                   PRECISION,
564                   ((oraclePrice * (FULL_PERCENT - baseBpsRangeLower)) / FULL_PERCENT)
565               )
566           ),
567           tickSpacing
568       );
569       int24 baseUpper_ = UniV3PoolMath.roundTick(
570           UniV3PoolMath.getTickAtSqrtRatio(
571               UniV3PoolMath.encodePriceSqrtX96(
572                   PRECISION,
573                   ((oraclePrice * (FULL_PERCENT + baseBpsRangeUpper)) / FULL_PERCENT)
574               )
575           ),
576           tickSpacing
577       );
578
579       if (delta > hysteresis || baseLower != baseLower_ || baseUpper != baseUpper_) {
580           baseLower = baseLower_;
```

```
581         baseUpper = baseUpper_;
582
583         // Swap tokens if required to reach the target price
584         uint160 sqrtPriceCurrentX96 = currentSqrtPriceX96();
585         uint160 sqrtPriceTargetX96 = UniV3PoolMath.encodePriceSqrtX96(PRECISION, oraclePrice);
586         if (sqrtPriceCurrentX96 != sqrtPriceTargetX96) {
587             // Determine if it is a token0-to-token1 or opposite swap
588             bool zeroToOne = sqrtPriceCurrentX96 > sqrtPriceTargetX96;
589             IUniswapV3Pool(pool).swap(
590                 address(this),
591                 zeroToOne,
592                 zeroToOne ? int256(token0Bal) : int256(token1Bal),
593                 sqrtPriceTargetX96, // Swap through ticks until the target price is reached or
                        run out of tokens
594                 abi.encode(address(this))
595             );
596         }
597
598         uint128 liquidity = _liquidityForAmounts(
599             baseLower,
600             baseUpper,
601             IERC20(token0).balanceOf(address(this)),
602             IERC20(token1).balanceOf(address(this))
603         );
604         // Set the CL position at the new baseLower and baseUpper ticks
605         _mintLiquidity(baseLower, baseUpper, liquidity);
606     } else {
607         // Since price difference was less than `hysteresis`, set the CL position back at
608         // the previous baseLower and baseUpper ticks
609         uint128 baseLiquidity = _liquidityForAmounts(baseLower, baseUpper, token0Bal, token1Bal)
                ;
610         _mintLiquidity(baseLower, baseUpper, baseLiquidity);
611     }
612 }
```

Listing 2.5: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

Impact Malicious users could potentially deplete the contract's position, thereby preventing the contract from earning fees.

Suggestion Revise the code accordingly.

2.2.2 Manipulable return value from the `getLpPrice` function

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 1`

Description There is a `getLpPrice` function in the `UniV3PoolHelper` contract that returns the price of the underlying `UniV3TokenizedLp` token.

```
230 function getLpPrice(uint256) public view returns (uint256 priceInEth) {
231     (uint256 rdntReserve, uint256 wethReserve, uint256 lpSupply) = getReserves();
```

```
232         uint256 wethForRdnt = tokenizedLpToken.fetchOracle(rdntAddr, weth9Addr, rdntReserve);
233         uint256 allReservesInWeth = wethReserve + wethForRdnt;
234         priceInEth = (allReservesInWeth * 1e8) / lpSupply;
235     }
```

Listing 2.6: contracts/main/radiant/zap/helpers/UniV3PoolHelper.sol

However, the return value of this function can be manipulated. Specifically, the `getReserves` function converts the position of the `UniV3TokenizedLp` contract into underlying amounts of the two tokens. While the liquidity of the position is determined, the spot price is vulnerable to manipulation. Manipulating the spot price can ultimately affect the LP price, potentially resulting in unexpected consequences.

```
218     function getReserves() public view returns (uint256 rdntManaged, uint256 wethManaged, uint256
            lpTokenSupply) {
219         lpTokenSupply = tokenizedLpToken.totalSupply();
220         (uint256 total0, uint256 total1) = tokenizedLpToken.getTotalAmounts();
221         (wethManaged, rdntManaged) = weth9Addr == token0 ? (total0, total1) : (total1, total0);
222     }
```

Listing 2.7: contracts/main/radiant/zap/helpers/UniV3PoolHelper.sol

```
349     function getTotalAmounts() public view override returns (uint256 total0, uint256 total1) {
350         (, uint256 base0, uint256 base1) = getBasePosition();
351         total0 = IERC20(token0).balanceOf(address(this)) + base0;
352         total1 = IERC20(token1).balanceOf(address(this)) + base1;
353     }
```

Listing 2.8: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

```
361     function getBasePosition() public view returns (uint128 liquidity, uint256 amount0, uint256
            amount1) {
362         (uint128 positionLiquidity, uint128 tokensOwed0, uint128 tokensOwed1) = _position(baseLower
                , baseUpper);
363         (amount0, amount1) = _amountsForLiquidity(baseLower, baseUpper, positionLiquidity);
364         liquidity = positionLiquidity;
365         amount0 = amount0 + uint256(tokensOwed0);
366         amount1 = amount1 + uint256(tokensOwed1);
367     }
```

Listing 2.9: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

```
813     function _amountsForLiquidity(
814         int24 tickLower,
815         int24 tickUpper,
816         uint128 liquidity
817     ) internal view returns (uint256, uint256) {
818         (uint160 sqrtRatioX96, , , , , , ) = IUniswapV3Pool(pool).slot0();
819         return
820             UniV3PoolMath.getAmountsForLiquidity(
821                 sqrtRatioX96,
822                 UniV3PoolMath.getSqrtRatioAtTick(tickLower),
823                 UniV3PoolMath.getSqrtRatioAtTick(tickUpper),
```

```
824             liquidity
825         );
826     }
```

Listing 2.10: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

Impact A manipulated LP price may lead to unforeseen consequences for protocols relying on this function.

Suggestion Revise the code accordingly.

2.3 Additional Recommendation

2.3.1 Remove redundant checks

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the `_distributeFees` function, there are redundant checks on the parameters which have been checked in the function that sets these parameters.

```
681   function _distributeFees(uint256 fees0, uint256 fees1) internal {
682       // if there is no affiliate 100% of the baseFee should go to feeRecipient
683       uint256 baseFeeSplit_ = (affiliate == NULL_ADDRESS) ? PRECISION : baseFeeSplit;
684
685
686       if (baseFee > PRECISION) {
687           revert UniV3TokenizedLp_FeeMustBeLtePrecision();
688       }
689       if (baseFeeSplit_ > PRECISION) {
690           revert UniV3TokenizedLp_SplitMustBeLtePrecision();
691       }
692       if (feeRecipient == NULL_ADDRESS) {
693           revert UniV3TokenizedLp_ZeroAddress();
694       }
```

Listing 2.11: contracts/main/radiant/zap/helpers/UniV3TokenizedLp

Impact Redundant checks can cause extraneous gas usages.

Suggestion Remove the redundant checks.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description In both the `UniV3PoolHelper` and `UniV3TokenizedLp` contracts, there are multiple functions with privileges to set key parameters for the contracts. Altering these parameters can significantly alter the functionality of the contracts, potentially rendering them unusable or in an incorrect state.

2.4.2 Assumption on the `UniV3PoolHelper` contract

Introduced by `Version 1`

Description It is assumed that the `UniV3PoolHelper` contract does not hold any tokens.

Feedback from the Project The team confirms there is no intention for `UniV3PoolHelper` to hold any tokens at any moment. This note is acknowledged by the team.

2.4.3 Dependency on the block timestamp

Introduced by `Version 1`

Description In the `_checkHysteresis` function, the block timestamp is used to prevent price manipulation by ensuring no previous swaps occur in the same block. However, when deploying contracts on the Arbitrum blockchain, this check may be ineffective. Specifically, the Arbitrum blockchain can have very short block intervals. Although timestamps are derived from Layer 1 (Ethereum), it's feasible to monitor timestamp changes and manipulate prices before the switch, effectively circumventing the intended limitations.

```
801   function _checkHysteresis() private view returns (bool) {
802       (, , uint16 observationIndex, , , , ) = IUniswapV3Pool(pool).slot0();
803       (uint32 blockTimestamp, , , ) = IUniswapV3Pool(pool).observations(observationIndex);
804       return (block.timestamp != blockTimestamp);
805   }
```

Listing 2.12: contracts/main/radiant/zap/helpers/UniV3TokenizedLp.sol

Feedback from the Project The team acknowledges that `block.timestamp` of pool observations can be bypassed in subsequent blocks, specifically in "tight" timestamp blockchains such as layer-2 EVM-compatible blockchains. However, the check still guards against the usage of a flash loan as an "atomic" price manipulation mode, which we consider the main attack vector to defend against. As the liquidity of the underlying pool becomes higher, this form of attack becomes more costly.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS