

PARAPET LABS

COMPETITIVE AUDIT

QuantAMM

Final Report · Optimism / EVM

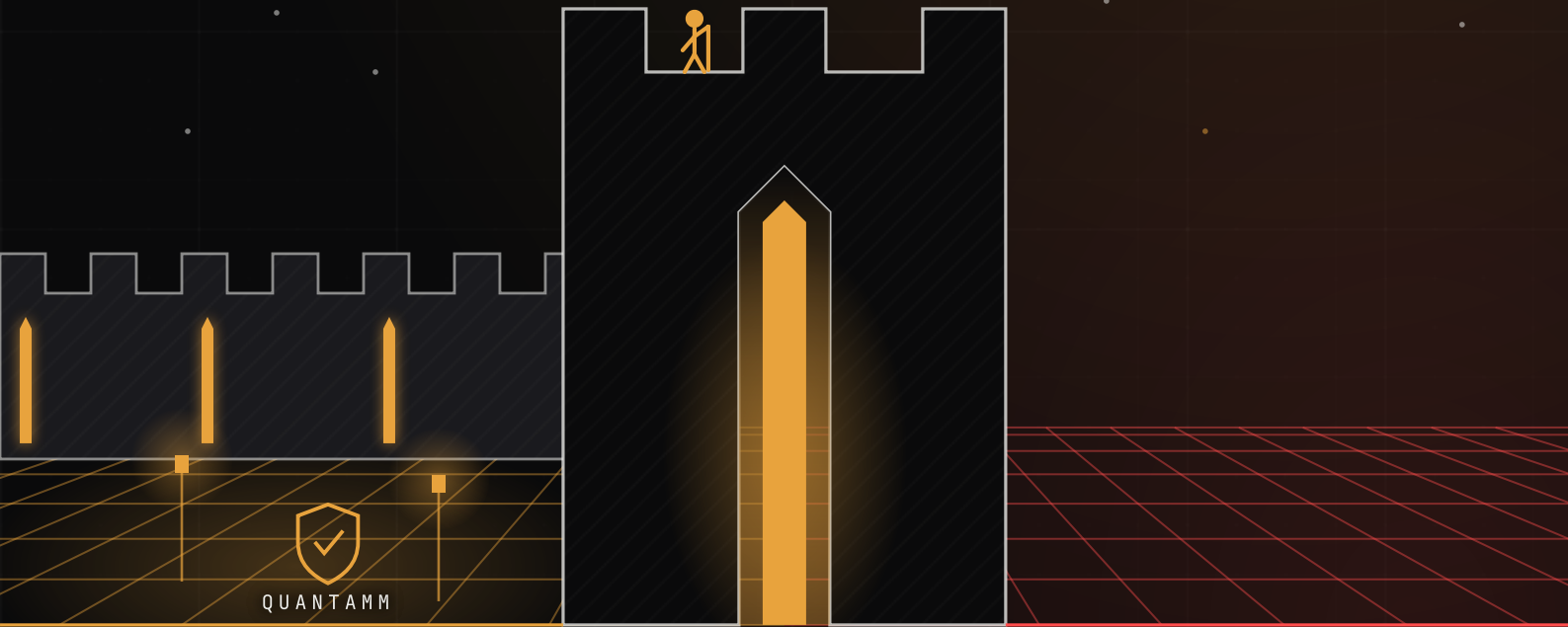
PREPARED FOR
QuantAMM

VERSION
0.09 · Final Report

WEBSITE
quantamm.fi/

COMMIT
a775db4273

AUDIT PERIOD
Dec 20, 2024 to Jan 15, 2025



INSIDE

THE WALL

OUTSIDE

STAND BEHIND THE PARAPET.

[PARAPETLABS.NET](https://ParapetLabs.net)

Contents

Document Control	4
Report information	4
Executive Summary	6
Protocol overview	6
Engagement overview	6
Summary of results	6
Disclaimer	7
Scope and Methodology	7
Scope	7
Methodology	9
Severity Classification	9
Impact	9
Likelihood	10
Severity levels	10
Finding status	10
Findings Overview	11
Summary of findings	11
Observations	13
Code maturity	13
Privileged roles and trust assumptions	13
Findings	14
Critical Severity	14
● C-01 · Donations are sandwichable to steal funds from LP	14
● C-02 · Fees sent to QuantAMMAdmin are stuck forever as there is no function to retrieve them	
● C-03 · Hook fees (ownerFee) are stuck in UpliftOnlyExample and non-retrievable	18
● C-04 · Improper access control in onAfterRemoveLiquidity() risks loss of funds by attacker	
High Severity	21
● H-01 · In onAfterRemoveLiquidity(), maxAmountsIn during admin fee distribution can cause txn reverts	
● H-02 · Truncation due to bad order of division causes higher fees sent to the admin	
● H-03 · Uplift fees are applied on depositAmount instead of the uplifted (profit) value	
● H-04 · Users can avoid paying fees on uplifts by utilising the afterUpdate logic	26
● H-05 · Wrong calculation of gradient	27
● H-06 · Wrong calculation of variance	29
Medium Severity	30
● M-01 · Attacker can completely prevent users from withdrawing in UpliftOnlyExample	
● M-02 · Division before multiplication in lpTokenDepositValueChange causes loss of fees	
● M-03 · In UpliftOnlyExample, the immutable updateWeightRunner address prevents seamless contract migration	

- M-04 · Last withdrawer will donate fees to empty pool, allowing MEV and leaving stuck funds in the pool
- M-05 · Moving average length validation prevents admin override for rules requiring historical data
- M-06 · Pool weights are non-updatable after updating the updateWeightRunner address
- M-07 · Single storage variable used for both swap and uplift fees 39
- M-08 · Users transferring their NFT position will retroactively get the new upliftFeeBps

Low Severity 42

- L-01 · Missing implementation of a function to change upliftFee 42
- L-02 · Wrong value added to blockTimestampDeposit in afterUpdate 43
- L-03 · minWithdrawalFeeBps is not added to upliftFeeBps, causing loss of fees and allowing MEV actions

Appendix 45

Tools used 45

Document Control

Report information

ITEM	DETAIL
Client	QuantAMM
Protocol	QuantAMM
Report	Final Report v0.09
Audit period	Dec 20, 2024 to Jan 15, 2025
Repository	https://github.com/Cyfrin/2024-12-quantamm
Commit reviewed	<code>a775db4273eb36e7b4536c5b60207c9f17541b92</code>
Fix-review commit	Absent
Language / chain	Solidity / Optimism / EVM
Website	https://quantamm.fi/

FOR PROTOCOL TEAMS READING THIS REPORT

This is what standing behind the parapet looks like.

Every finding in this report came with a recommended fix, but since this engagement is special, there was no fix verified based on the nature on the engagement. Your codebase can stand behind the same/better wall.

THIS REPORT

THIS ENGAGEMENT

Competitive Audit

Findings from the public Codehawks competitive audit of QuantAMM (December 2024 - January 2025) by two of our senior researcher submissions.

EARLY CODE • TIGHT BUDGET

AI Scan

A fast AI-assisted pass that flags suspect code before a human reads a line. The quickest way to catch obvious flaws while they are cheap to fix.

LAUNCH IS CLOSE • OR THE STAKES ARE HIGH

Smart Contract Audit

Senior auditors read your code the way an attacker would, then stay until every fix is verified. No juniors on your code.

STILL ON THE WHITEBOARD

Protocol Design Review

Threat modelling and architecture review before the code exists, so the flaw never ships.

BREAK IT BEFORE THEY DO

Fuzzing Suite

An adversarial fuzzing campaign built around your invariants, run until the code stops breaking.

PROVE THE CORE LOGIC

Formal Verification

Mathematical proof that your core invariants hold in every reachable state, not just the ones tests reach.

Every day your code sits unreviewed, an attacker can read it first.

contact@parapetlabs.net • [Telegram @mrzizah](https://t.me/mrzizah)

GET A FIXED QUOTE AT PARAPETLABS.NET

NOT READY FOR A QUOTE YET?

Read how exploits actually happen.

Security notes and new solutions from the wall. No noise, no spam, unsubscribe anytime.

[GET THE DISPATCHES](#)

Know a team that needs an audit?

Send them our way and get rewarded: one warm intro, no paperwork. If they book an audit, you earn a referral reward.

[REFER A TEAM](#)

Executive Summary

Protocol overview

QuantAMM is an AMM protocol built as a hook on top of balancerV3 contracts.

the weights of the pool change to take advantage of current underlying price movements and therefore can overcome MEV and Impermanent Loss. QuantAMM does this in a continuous, responsive way with on-chain TradFi-style strategies. QuantAMM also allows community members with strategy ideas to use protocol-provided highly efficient strategy building blocks to build their own automated strategy pool.

The market making variant where weights change continuously over time is called Temporal Function Market Making (TFMM)

Engagement overview

PROJECT	QuantAMM
REPOSITORY	https://github.com/Cyfrin/2024-12-quantamm
COMMIT HASH	a775db4273eb36e7b4536c5b60207c9f17541b92
MITIGATION HASH	Absent
AUDIT TIMELINE	Dec 20, 2024 to Jan 15, 2025
LANGUAGE / CHAIN	Solidity / Optimism / EVM

Parapet Labs was involved through a competitive audit for **QuantAMM** to perform a security review of their **Block chain traded funds** (<https://github.com/Cyfrin/2024-12-quantamm>), at commit

[a775db4273eb36e7b4536c5b60207c9f17541b92](#), between Dec 20, 2024 and Jan 15, 2025. The review covered the contracts listed in the [scope](#) using the methods described in [Methodology](#).

Summary of results

The review identified **21 findings**: 4 critical, 6 high, 8 medium and 3 low. 0 of 21 are marked as resolved at the time of writing.

4 CRITICAL	6 HIGH	8 MEDIUM	3 LOW	0 INFORMATIONAL	0 GAS OPTIMIZATION
---------------	-----------	-------------	----------	--------------------	--------------------------

Disclaimer

This report is the result of a time-boxed competitive audit carried out by Parapet Labs for QuantAMM. It is provided for information purposes only, and it is not investment, legal, financial or tax advice.

Competitive Audit. Findings from the public Codehawks competitive audit of QuantAMM (December 2024 - January 2025) 21 unique vulnerabilities identified by two of our senior researcher submissions and they were able to find all high severity issues in the code along a lot more valuable bugs

No guarantee of security. A security review reduces risk; it cannot remove it. The findings in this report reflect the code at the commit stated in [Document Control](#) and the knowledge and tooling available to the audit team during the engagement. The absence of a finding in an area does not mean that area is free of vulnerabilities.

Scope. Only the files and functionality listed in the [scope](#) were reviewed. Third-party contracts, off-chain infrastructure, front ends, keys and operational procedures are out of scope unless explicitly stated. Behaviour of external dependencies, oracles, bridges and other protocols that the system integrates with was assumed to match their documentation.

Changes after the review. Code that is modified, upgraded or redeployed after the reviewed commit has not been assessed. Deployed bytecode should be checked against the reviewed source before relying on this report.

Reliance. This report may be relied on only by the party it was prepared for and only for the purpose stated in the engagement. Parapet Labs accepts no liability for losses arising from the use of the audited code or from decisions made on the basis of this report.

Scope and Methodology

Scope

FILE	NOTES
<code>pkg/pool-quantamm/contracts/ChainlinkOracle.sol</code>	
<code>pkg/pool-quantamm/contracts/MultiHopOracle.sol</code>	
<code>pkg/pool-quantamm/contracts/UpdateWeightRunner.sol</code>	
<code>pkg/pool-quantamm/contracts/QuantAMMStorage.sol</code>	
<code>pkg/pool-quantamm/contracts/QuantAMMWeightedPoolFactory.sol</code>	

FILE	NOTES
pkg/pool-quantamm/contracts/QuantAMMWeightedPool.sol	
pkg/pool-quantamm/contracts/rules/AntimomentumUpdateRule.sol	
pkg/pool-quantamm/contracts/base/QuantammBasedRuleHelpers.sol	
pkg/pool-quantamm/contracts/base/QuantammCovarianceBasedRule.sol	
pkg/pool-quantamm/contracts/base/QuantammGradientBasedRule.sol	
pkg/pool-quantamm/contracts/base/QuantammMathGuard.sol	
pkg/pool-quantamm/contracts/base/QuantammMathMovingAverage.sol	
pkg/pool-quantamm/contracts/base/QuantammVarianceBasedRule.sol	
pkg/pool-quantamm/contracts/rules/ChannelFollowingUpdateRule.sol	
pkg/pool-quantamm/contracts/rules/DifferenceMomentumUpdateRule.sol	
pkg/pool-quantamm/contracts/rules/MinimumVarianceUpdateRule.sol	
pkg/pool-quantamm/contracts/rules/MomentumUpdateRule.sol	
pkg/pool-quantamm/contracts/rules/PowerChannelUpdateRule.sol	
pkg/pool-quantamm/contracts/rules/UpdateRule.sol	
pkg/pool-hooks/contracts/hooks-quantamm/LPNFT.sol	
pkg/pool-hooks/contracts/hooks-quantamm/UpliftOnlyExample.sol	
pkg/interfaces/contracts/pool-quantamm/IQuantAMMWeightedPool.sol	
pkg/interfaces/contracts/pool-quantamm/IUpdateRule.sol	
pkg/interfaces/contracts/pool-quantamm/IUpdateWeightRunner.sol	
pkg/interfaces/contracts/pool-quantamm/OracleWrapper.sol	

REFERENCE	VALUE
Repository	https://github.com/Cyfrin/2024-12-quantamm
Reviewed commit	a775db4273eb36e7b4536c5b60207c9f17541b92
Fix-review commit	Absent
Language / chain	Solidity / Optimism / EVM

Methodology

Every engagement follows the same four steps.

1. **Scope.** We read the repository, agree what is in and out of scope, and fix the commit under review.
2. **Adversarial review.** The code is attacked from the point of view of someone trying to drain it. This engagement used:
 - AI-assisted scan of the in-scope code
 - Manual line-by-line review
 - Fuzzing and invariant testing
3. **Report.** Findings are ranked by severity, and each one comes with a recommended fix. Where we could, a proof of concept demonstrates the issue.
4. **Fix review.** We re-check every patch and update the status of each finding.

Severity Classification

Each finding is rated on two axes: the **impact** if it is exploited, and the **likelihood** that it can be exploited in practice. The combination gives the severity.

		IMPACT		
		High	Medium	Low
LIKELIHOOD	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

Impact

LEVEL	MEANING
High	Loss or permanent lock-up of user or protocol funds, takeover of privileged functionality, or a break of a core invariant.
Medium	Temporary loss of funds or service, value leakage, or a broken feature that has a workaround.
Low	Minor leakage, edge-case misbehaviour or inefficiency with limited consequence.

Likelihood

LEVEL	MEANING
High	Exploitable by anyone at low cost, or likely to occur in normal use.
Medium	Needs specific conditions, capital or timing, but is realistic.
Low	Needs an unlikely combination of events, a privileged actor, or an uneconomic attack.

Severity levels

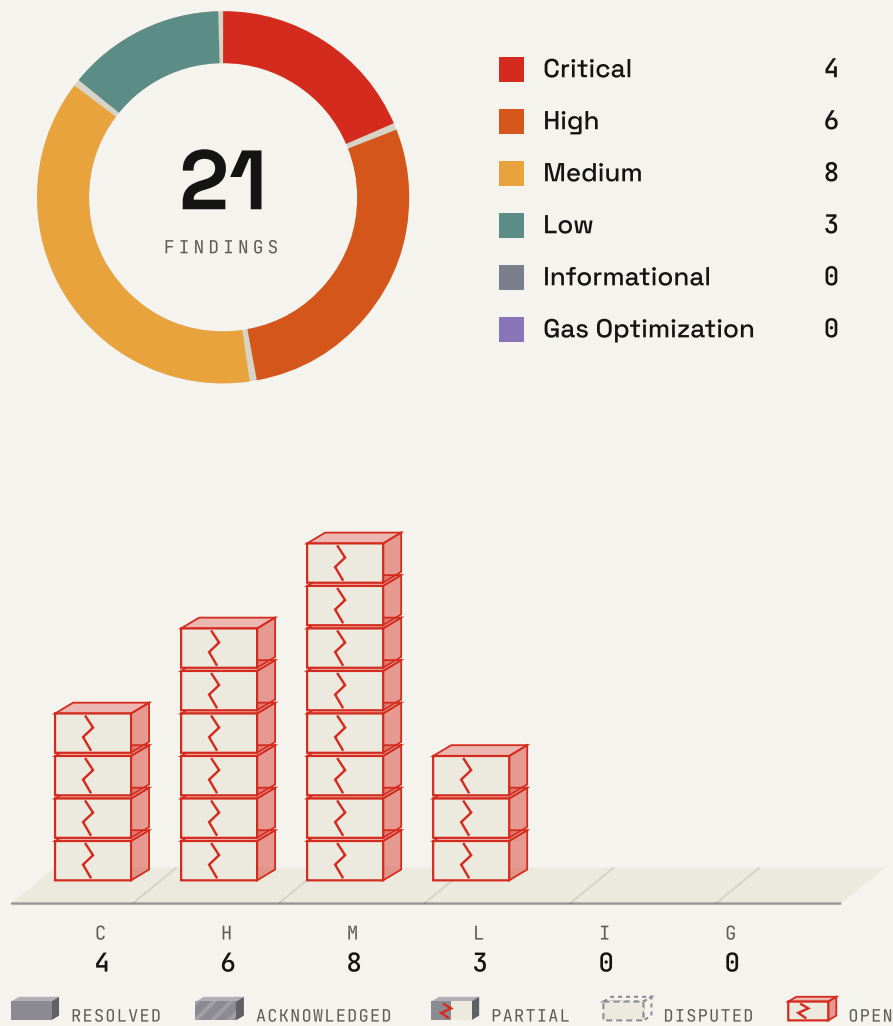
SEVERITY	DESCRIPTION
Critical	Direct and likely loss of funds. Fix before deployment.
High	Serious impact or a likely path to loss. Fix before deployment.
Medium	Real but conditional risk. Should be fixed.
Low	Limited impact. Fix when practical.
Informational	No direct security impact. Code quality, clarity and best practice.
Gas Optimization	Changes that reduce gas cost without changing behaviour.

Finding status

STATUS	MEANING
Resolved	The fix was reviewed and addresses the issue.
Partially Resolved	The fix reduces but does not remove the risk.
Acknowledged	The team accepts the risk or will address it later.
Disputed	The team disagrees that this is an issue.
Open	No response or fix has been reviewed yet.

Findings Overview

The review identified **21 findings**: 4 critical, 6 high, 8 medium and 3 low. 0 of 21 are marked as resolved at the time of writing.



Summary of findings

ID	TITLE	SEVERITY	STATUS
C-01	Donations are sandwichable to steal funds from LP	CRITICAL	OPEN
C-02	Fees sent to QuantAMMAdmin are stuck forever as there is no function to retrieve them	CRITICAL	OPEN
C-03	Hook fees (ownerFee) are stuck in UpliftOnlyExample and non-retrievable	CRITICAL	OPEN

ID	TITLE	SEVERITY	STATUS
C-04	Improper access control in onAfterRemoveLiquidity() risks loss of funds by attacker	CRITICAL	OPEN
H-01	In onAfterRemoveLiquidity(), maxAmountsIn during admin fee distribution can cause txn reverts	HIGH	OPEN
H-02	Truncation due to bad order of division causes higher fees sent to the admin	HIGH	OPEN
H-03	Uplift fees are applied on depositAmount instead of the uplifted (profit) value	HIGH	OPEN
H-04	Users can avoid paying fees on uplifts by utilising the afterUpdate logic	HIGH	OPEN
H-05	Wrong calculation of gradient	HIGH	OPEN
H-06	Wrong calculation of variance	HIGH	OPEN
M-01	Attacker can completely prevent users from withdrawing in UpliftOnlyExample	MEDIUM	OPEN
M-02	Division before multiplication in lpTokenDepositValueChange causes loss of fees	MEDIUM	OPEN
M-03	In UpliftOnlyExample, the immutable updateWeightRunner address prevents seamless contract migration	MEDIUM	OPEN
M-04	Last withdrawer will donate fees to empty pool, allowing MEV and leaving stuck funds in the pool	MEDIUM	OPEN
M-05	Moving average length validation prevents admin override for rules requiring historical data	MEDIUM	OPEN
M-06	Pool weights are non-updatable after updating the updateWeightRunner address	MEDIUM	OPEN
M-07	Single storage variable used for both swap and uplift fees	MEDIUM	OPEN
M-08	Users transferring their NFT position will retroactively get the new upliftFeeBps	MEDIUM	OPEN
L-01	Missing implementation of a function to change upliftFee	LOW	OPEN
L-02	Wrong value added to blockTimestampDeposit in afterUpdate	LOW	OPEN
L-03	minWithdrawalFeeBps is not added to upliftFeeBps, causing loss of fees and allowing MEV actions	LOW	OPEN

Observations

Code maturity

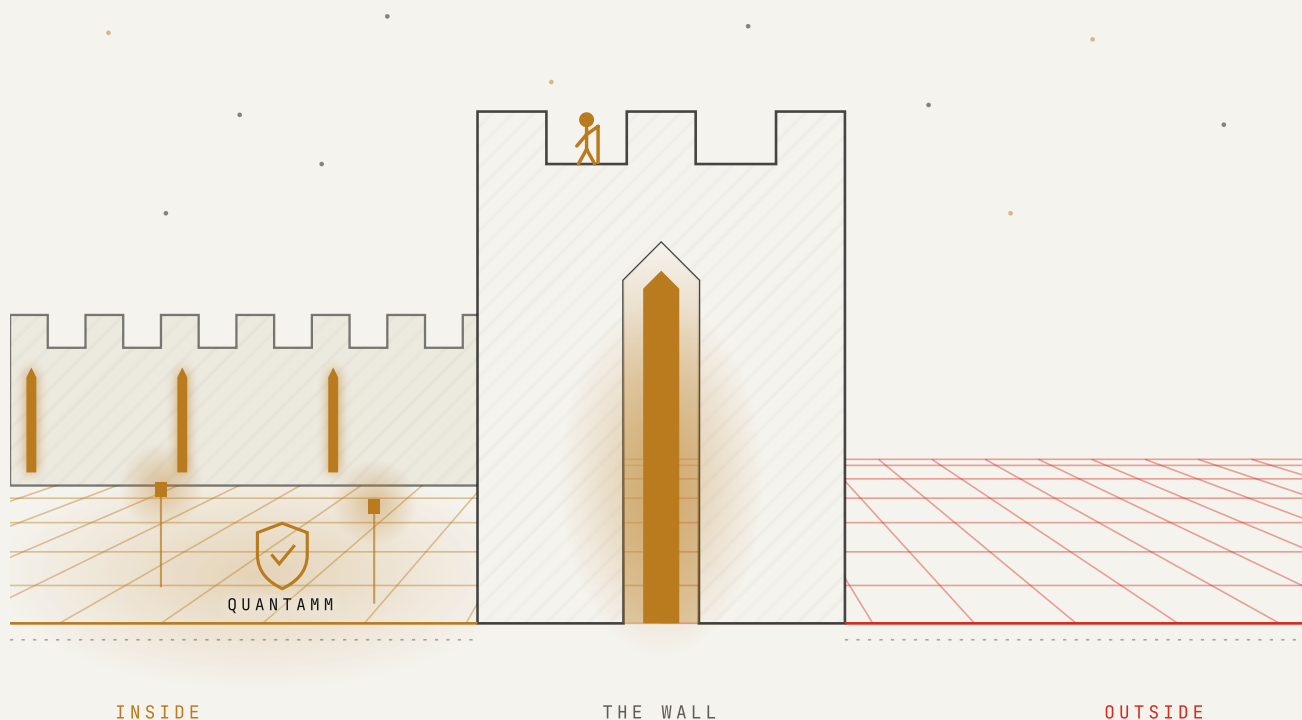
CATEGORY	SUMMARY	RESULT
Arithmetic	Core rule math is incorrect	MODERATE
Access control	Hook to balancer vault access control was wrong	MISSING
Decentralization	Pool owners and protocol admin may adversely affect users	WEAK
Testing	Good unit tests with fuzz tests but with alot of mocked state	SATISFACTORY

Privileged roles and trust assumptions

ROLE	POWERS	WHAT USERS MUST TRUST
QuantAMM Admin	set/change critical protocol variables	a timelock
Pool Owner	Pool owner	may or may not have admin powers based on creation configs
QuantAMM pool runner	Permissionless update the pool weight based on the strategy and the chainlink prices	can be a chainlink keeper workder

Findings

Findings are grouped by severity. Each one lists its status, the affected code, a description of the issue, its impact and a recommendation.



Critical Severity

C-01 • Donations are sandwichable to steal funds from LP

CRITICAL

● OPEN

UpliftOnlyExample.sol#L555-L567

Summary

Large Fees donations are sandwichable in `UpliftOnlyExample` stealing funds from the deserving actual LP providers

Vulnerability Details

UpLift fees taken from `removeLiquidityProportional` during `onAfterRemoveLiquidity` are donated to the pool in stepwise matter,

```
File: UpliftOnlyExample.sol
555:     if (localData.adminFeePercent  $\neq$  1e18) {
```

```

556:          // Donates accrued fees back to LPs.
557:      _vault.addLiquidity(
558:          AddLiquidityParams({
559:              pool: localData.pool,
560:              to: msg.sender, // It would mint BPTs to router, but it's a donation so no
BPT is minted
561:              maxAmountsIn: localData.accruedFees, // Donate all accrued fees back to the
pool (i.e. to the LPs)
562:              minBptAmountOut: 0, // Donation does not return BPTs, any number above 0
will revert
563:              kind: AddLiquidityKind.DONATION,
564:              userData: bytes("") // User data is not used by donation, so we can set it
to an empty string
565:          })
566:      );
567:  }

```

- This allow MEV (on L1) or racing txns to get those values that they don't deserve, basically stealing them from genuine LP providers, then immediately removing the liquidity after

An example of a whale removing liquidity with huge upLift fees donated:

1- Old whale in wETH-USDC Pool having his balance got upLifted 1000%

- from 2,500,000 USD/100wETH → 25,000,000 USD/100wETH
- Total value in the Pool is 26,000,000 (whale has most of the pool)

2- `upLiftFeeBps` is set to 5% * 10 (value change) (11,500,000) as fees donated to a pool of 1,000,000

3- a MEV sees the whale liquidity removal and sandwich it (flashloan or use his own funds) add liquidity worth 1,000,000 (having 50% of the pool and getting 50% of the donation)

4- When the MEV submit a withdrawal, He will own (6,750,000 from donation + his own 1 million = 7,750,000)

5- his uplift is 5% * ~8 = 40% = 3,100,000

6- MEV leave the pool with 4,650,000 having a profit of 4,650,000

The above numbered example may have exaggerated numbers only to show the feasibility of the MEV attack and the fee is not enough, attacks can be carried on smaller whales too, the idea stay the same

NOTE! Its worth mentioning that above i assumed that `upLiftFeeBps` is applied in the whole withdrawal value and not value increase only, thats how the code actually works, but that was another Bug

Assuming that `upLiftFeeBps` is only applied on the value increase, then the attack will be completely easy to be carried by MEV, not requiring large whale % of pool getting out immediately Here is what happens

1- whale in wETH-USDC Pool having his balance got upLifted 500%

- from 2,500,000 USD/100wETH → 12,500,000 USD/100wETH
- Total value in the Pool is 100,000,000

2- `upLiftFeeBps` is set to $5\% * 5 * 10,000,000$ (value increase) (2,500,000) as fees donated to a pool of 87,500,000

3- a MEV sees the whale liquidity removal and sandwich it (flashloan or use his own funds) add liquidity worth 50,000,000 (having ~50% of the pool and getting ~50% of the donation)

4- When the MEV submit a withdrawal, He will own (1,750,000 from donation + his own 50 million = 51,750,000) (2% upLift)

5- his uplift is $5\% * \sim 2/100 = 0.1\% = 0.1 * 1,750,000 / 100 = 1750$

6- MEV leave the pool with ~51,749,000 having a profit of 1,749,000

In the above second example, there can be small inaccuracies coming from small percentage change and uncertainty of how upLift Fees would actually apply to only upLift value

Impact

Stealing of funds from LP providers

Tools Used

Manual review

Recommendations

implement timelock on LP providing, or implement vesting logic for donation

C-02 · Fees sent to QuantAMMAdmin are stuck forever as there is no function to retrieve them

CRITICAL

● OPEN

UpliftOnlyExample.sol#L536-L545

UpliftOnlyExample.sol#L265-L275

Summary

When a user removes liquidity from the pool he pays fees and a portion of the fees is liquidity added back for `QuantAMMAdmin`

Those funds are stuck in, and can't be removed as there is no mapping or NFT minted for those shares so any attempt to remove liquidity will revert

Vulnerability Details

Here is the call inside `onAfterREmoveLiquidity` to add fees to `quantAMMAdmin`

```
File: UpliftOnlyExample.sol
536:         if (localData.adminFeePercent > 0) {
537:             _vault.addLiquidity(
538:                 AddLiquidityParams({
539:                     pool: localData.pool,
540:                     to: IUpdateWeightRunner(_updateWeightRunner).getQuantAMMAdmin(),
541:                     maxAmountsIn: localData.accruedQuantAMMFees,
542:                     minBptAmountOut: localData.feeAmount.mulDown(localData.adminFeePercent) /
543:                     1e18,
```

```

543:                kind: AddLiquidityKind.PROPORTIONAL,
544:                userData: bytes("")
545:            })

```

In `removeLiquidityProportional` there is a check for the length of `poolsFeeData` in this case, would be zero and any attempt to remove will revert

```

File: UpliftOnlyExample.sol
265:    function removeLiquidityProportional(
266:        uint256 bptAmountIn,
267:        uint256[] memory minAmountsOut,
268:        bool wethIsEth,
269:        address pool
270:    ) external payable saveSender(msg.sender) returns (uint256[] memory amountsOut) {
271:        uint depositLength = poolsFeeData[pool][msg.sender].length;
272:
273:        if (depositLength == 0) {
274:            revert WithdrawalByNonOwner(msg.sender, pool, bptAmountIn);
275:        }

```

POC

Add this anywhere in `UpliftExample.t.sol`

```

function testRemoveLiquidity_Stuck_Admin_Fees() public {
    vm.prank(address(vaultAdmin));
    updateWeightRunner.setQuantAMMUpliftFeeTake(0.5e18);
    vm.stopPrank();
    // Add liquidity so bob has BPT to remove liquidity.
    uint256[] memory maxAmountsIn = [dai.balanceOf(bob), usdc.balanceOf(bob)].toMemoryArray();
    vm.prank(bob);
    upliftOnlyRouter.addLiquidityProportional(pool, maxAmountsIn, bptAmount, false, bytes(""));
    vm.stopPrank();
    uint256[] memory minAmountsOut = [uint256(0), uint256(0)].toMemoryArray();
    vm.startPrank(bob);
    upliftOnlyRouter.removeLiquidityProportional(bptAmount, minAmountsOut, false, pool);
    vm.stopPrank();
    BaseVaultTest.Balances memory balancesAfter =
    getBalances(updateWeightRunner.getQuantAMMAdmin());
    //trying to remove liquidity added to QuantAMMAdmin with the value added from bob removing
    liquidity the remove attempt will revert with `WithdrawalByNonOwner` error
    vm.prank(updateWeightRunner.getQuantAMMAdmin());
    vm.expectRevert();
    upliftOnlyRouter.removeLiquidityProportional(5000000000000000000, minAmountsOut, false,
pool);
    vm.stopPrank();
}

```

Impact

- **Loss of Funds:** Fees added as liquidity for `QuantAMMAdmin` are stuck and cannot be removed.

Tools Used

manual review

Recommendations

- create a new array while creating as done for normal addition

C-03 • Hook fees (ownerFee) are stuck in UpliftOnlyExample and non-retrievable

CRITICAL**● OPEN**

UpliftOnlyExample.sol#L342-L345

Summary

UpliftOnlyExample::onAfterSwap() some fees taken from the swap are not sent to quantAMMAdmin or Hook owner, but sent to UpliftOnlyExample it self with no retrievable way for the tokens

Vulnerability Details

after pool swap, there is a hook called onAfterSwap() in UpliftOnlyExample to deduct swap fees for the hook contract, (part of it is sent to the Quant admin)

```
File: UpliftOnlyExample.sol
293:     function onAfterSwap(
341:
342:         if (ownerFee > 0) {
343:             _vault.sendTo(feeToken, address(this), ownerFee);
344:
345:             emit SwapHookFeeChanged(address(this), feeToken, ownerFee);
350:     }
```

But the problem in the above code is that it sends the fees amount ownerFee to it self, with no function to retrieve those tokens

Impact

Loss of swap fees

Tools Used

Manual review

Recommendations

use the ownerFee to be donated to the pool like the logic in onAfterRemoveLiquidity Hook or if its intended to be sent to Pool owner, then send those tokens to him like the way you did with admin fees

C-04 • Improper access control in onAfterRemoveLiquidity() risks loss of funds by attacker

CRITICAL**● OPEN**

UpliftOnlyExample.sol#L431-L440 UpliftOnlyExample.sol#L493-L511

Summary

in `UpliftOnlyExample::onAfterRemoveLiquidity()` absent access control to only allow the vault calls opens up a path to cause loss of funds to LP providers by an attacker calling to with victim address to burn the victim NFTs

Vulnerability Details

`onAfterRemoveLiquidity` is not access controlled and can be called by any one as long as the passed router is the `UpliftOnlyExample` it self, which is false access control guard.

Also the function takes the user address that his NFT will be burnt as a passed variable.

What makes it worse is that `LPNFT::Burn()` doesn't have checks for tokens approvals or ownership, so `UpliftOnlyExample` can burn any user NFT

Here is what can happen:

1- Attacker calls `Unlock()` on balancerV3 vault to have a transient unlock of the vault for the txn

2- Attacker calls `onAfterRemoveLiquidity` after calling unlock passing the:

- router as `UpliftOnlyExample` to pass the `onlySelfRouter` modifier
- `userData` to be the victim address
- `bptAmountIn` to be the balance registered in `poolsFeeData` for the victim

3- The NFTs of the victim will be burnt due to this block

```
File: UpliftOnlyExample.sol
493:         if (feedataArray[i].amount ≤ localData.amountLeft) {
494:             uint256 depositAmount = feedataArray[i].amount;
495:             localData.feeAmount += (depositAmount * feePerLP);
496:
497:             localData.amountLeft -= feedataArray[i].amount;
498:
499:             lpNFT.burn(feedataArray[i].tokenID);
500:
501:             delete feedataArray[i];
502:             feedataArray.pop();
503:
504:             if (localData.amountLeft == 0) {
505:                 break;
506:             }
507:         } else {
508:             feedataArray[i].amount -= localData.amountLeft;
509:             localData.feeAmount += (feePerLP * localData.amountLeft);
510:             break;
511:         }
```

4- Attacker will need to call `settle()` to settle debt taken from the `UpliftOnlyExample` call to `addLiquidity()` (the fee distribution Logic)

```
File: UpliftOnlyExample.sol
536:         if (localData.adminFeePercent > 0) {
```

```

537:         _vault.addLiquidity(
538:             AddLiquidityParams({
539:                 pool: localData.pool,
540:                 to: IUpdateWeightRunner(_updateWeightRunner).getQuantAMMAdmin(),
541:                 maxAmountsIn: localData.accruedQuantAMMFees,
542:                 minBptAmountOut: localData.feeAmount.mulDown(localData.adminFeePercent) /
1e18,
543:                 kind: AddLiquidityKind.PROPORTIONAL,
544:                 userData: bytes("")
545:             })
546:         );
547:         emit ExitFeeCharged(
548:             userAddress,
549:             localData.pool,
550:             IERC20(localData.pool),
551:             localData.feeAmount.mulDown(localData.adminFeePercent) / 1e18
552:         );
553:     }
554:
555:     if (localData.adminFeePercent != 1e18) {
556:         // Donates accrued fees back to LPs.
557:         _vault.addLiquidity(
558:             AddLiquidityParams({
559:                 pool: localData.pool,
560:                 to: msg.sender, // It would mint BPTs to router, but it's a donation so no
BPT is minted
561:                 maxAmountsIn: localData.accruedFees, // Donate all accrued fees back to the
pool (i.e. to the LPs)
562:                 minBptAmountOut: 0, // Donation does not return BPTs, any number above 0
will revert
563:                 kind: AddLiquidityKind.DONATION,
564:                 userData: bytes("") // User data is not used by donation, so we can set it
to an empty string
565:             })
566:         );
567:     }
.....
File: Vault.sol
507:     function addLiquidity(
569:         (amountsIn, amountsInScaled18, bptAmountOut, returnData) = _addLiquidity(
570:             poolData,
571:             params,
572:             maxAmountsInScaled18
573:         );
574:     )
.....
610:     function _addLiquidity(
732:         // 3) Deltas: Debit of token[i] for amountInRaw.
733:         _takeDebt(token, amountInRaw);
734:     )

```

The loss for attacker is very smaller (fees of minimumSwapFees or UpLiftFee) compared to the complete loss of funds for the victim, since users can only retrieve funds through `removeLiquidityProportional` by their NFT and the `poolsFeeData`

Impact

Complete loss of funds

Tools Used

Manual Review

Recommendations

Add `OnlyVault` modifier

High Severity

H-01 • In `onAfterRemoveLiquidity()`, `maxAmountsIn` during admin fee distribution can cause txn reverts

HIGH**● OPEN**

UpliftOnlyExample.sol#L522-L542

Summary

In `onAfterRemoveLiquidity()` during `_vault.addLiquidity()` made to the admin, `maxAmountsIn` will cause reverts in multiple circumstances

Vulnerability Details

In `onAfterRemoveLiquidity()`, `amountsOutRaw` can round down when calculating `exitFee` here

```
File: UpliftOnlyExample.sol
522:         for (uint256 i = 0; i < localData.amountsOutRaw.length; i++) {
523:             uint256 exitFee = localData.amountsOutRaw[i].mulDown(localData.feePercentage);
```

That `exitFee` is then used to get `accruedQuantAMMFees` (used as `maxAmountIn` during `_vault.addLiquidity()`)

- Both of the above variables are the actual tokens of the Pool that will be used during liquidity addition
- `amountsOutRaw` array can have low decimal tokens like USDC for example that will most likely have slight truncation during fee calculation

Now during `minBptAmountOut` calculations in `_vault.addLiquidity()` to the admin

```
File: UpliftOnlyExample.sol
542:             minBptAmountOut: localData.feeAmount.mulDown(localData.adminFeePercent) /
1e18,
```

- There can be cases where BPT amounts are perfectly divisible and doesn't round down
- * This is likely to happen since during Liquidity addition BPT is calculated and then stay unchanged after that, but Pool reserves change by swapping and weight changes

* So during swapping balances corresponding to those BPT change by wei through the invariant swap formula

* BPT tokens are 18 decimals and less likely to round down in that calculation

The above condition will create a situation where during `_vault.addLiquidity()` the `minBptAmountOut` was accurate but the call will provide less than needed `maxAmountsIn` causing a revert of the txn

The above temporary DOS can be mitigated by user repeatedly trying to adjust `bptAmountIn` that rounds down (to have consistent double rounding down in `onAfterRemoveLiquidity()`) during his call to `removeLiquidityProportional()`

Pseudo Exemplified of numbered scenario

1- Token A (USDC, 6 decimals):

- `amountsOutRaw[0]` = 100.000123 USDC (100_000_123)

2- Token B (WETH, 18 decimals):

- `amountsOutRaw[1]` = 0.1000000000000000123 WETH (100_000_000_000_000_123)

Parameters:

- `feePercentage` = 0.02e18 (2%)
- `adminFeePercent` = 0.5e18 (50%)
- `feeAmount` = 20 (perfect number, no decimals)

Calculation flow with rounding:

1- For USDC:

- `exitFee` = $100_000_123 \times 0.02e18 / 1e18 = 2_000_002$ (rounded down)
- `accruedQuantAMMFees[0]` = $2_000_002 \times 0.5e18 / 1e18 = 1_000_001$

2- For WETH:

- `exitFee` = $100_000_000_000_000_123 \times 0.02e18 / 1e18 = 2_000_000_000_000_002$
- `accruedQuantAMMFees[1]` = $2_000_000_000_000_002 \times 0.5e18 / 1e18 = 1_000_000_000_000_001$

3- BPT calculation:

- `feeAmount` = 2e18 (perfect number)
- `minBptAmountOut` = $20 \times 0.5e18 / 1e18 = 1e18$ (no rounding needed)

Txn reverting due to `minBptAmountOut` not reached with the provided `accruedQuantAMMFees` as `maxAmountsIn`

Please notice that simply rounding Up won't work for the same reason (a variable will be perfect % and will not benefit from `mulUp` while other variable gets benefited from `mulUp` cause it would have been truncated otherwise) and this issue sequence will always happen and is different from issues simply stating that `adminFees` should `roundUp`

Impact

Multiple txn reverts in multiple circumstances affecting the availability of `removeLiquidityProportional()` (temporary DOS)

Tools Used

Manual review

Recommendations

Build a logic that

- Doesn't pass `maxAmountsIn` or pass a very high amount, doesn't matter
- uses returned `amountsIn` used from `addLiquidity()` and then use those to remove it from `hookAdjustedAmountsOutRaw`

H-02 • Truncation due to bad order of division causes higher fees sent to the admin

HIGH**OPEN**

UpliftOnlyExample.sol#L334-L335

Summary

precision loss in `UpliftOnlyExample::onAfterSwap()` fee calculation mechanism that leads to very high QuantAdmin fees percentage from `hookFee`.

Vulnerability Details

The fee calculation in `onAfterSwap()` performs division operations in a bad order:

```
if (quantAMMFeeTake > 0) {
    uint256 adminFee = hookFee / (1e18 / quantAMMFeeTake);
    ownerFee = hookFee - adminFee;
```

The issue arises from dividing `1e18` by `quantAMMFeeTake` first, which creates truncation that gets magnified in subsequent calculations.

Examples:

- For `quantAMMFeeTake = 0.3e18` (30%)
- $1e18 / 0.3e18 = 3.333...$ truncates to 3
- With `hookFee = 100e18`:

* Calculated: $\text{adminFee} = 100\text{e}18/3 = 33\text{e}18$

* Expected: $100\text{e}18 * 0.3 = 30\text{e}18$

* Difference: +10% overcharge

Another Example That will eat all the swap Fees (Works with any thing above $0.5\text{e}18$ as `quantAMMFeeTake`):

- For `quantAMMFeeTake` = $0.7\text{e}18$ (70%)

- $1\text{e}18 / 0.7\text{e}18 = 1.428\dots$ truncates to 1

- With `hookFee` = $100\text{e}18$:

* Calculated: $\text{adminFee} = 100\text{e}18/1 = 100\text{e}18$

* Expected: $100 * 0.7 = 70\text{e}18$

* Difference: +42.8% overcharge

NOTE!: The issues doesn't assume the admin to be behaving maliciously, but its just a valid fee % that is $\leq 1\text{e}18$ as checked here

File: `UpdateWeightRunner.sol`

```
141:     function setQuantAMMUpLiftFeeTake(uint256 _quantAMMUpLiftFeeTake) external{
142:         require(msg.sender == quantammAdmin, "ONLYADMIN");
143:         require(_quantAMMUpLiftFeeTake ≤ 1e18, "Uplift fee must be less than 100%");
144:         uint256 oldSwapFee = quantAMMSwapFeeTake;
145:         quantAMMSwapFeeTake = _quantAMMUpLiftFeeTake;
146:
147:         emit UpLiftFeeTakeSet(oldSwapFee, _quantAMMUpLiftFeeTake);
148:     }
```

Impact

Fees overcharged % from `hookFee` during `onAfterSwap()` Hook

Tools Used

- Manual code

Recommendations

Implement the following fix to maintain precision:

```
uint256 adminFee = (hookFee * quantAMMFeeTake) / 1e18;
```

H-03 • Uplift fees are applied on depositAmount instead of the uplifted (profit) value

HIGH

● OPEN

UpliftOnlyExample.sol#L480-L495

Summary

The `UpliftOnlyExample` contract's withdrawal fee calculation applies fees on the total withdrawal amount rather than just the uplift value, contradicting the whitepaper specification and potentially causing users to receive less than their initial deposit value.

Vulnerability Details

In `onAfterRemoveLiquidity()`, the fee calculation is performed as:

```
feePerLP = (uint256(localData.lpTokenDepositValueChange) * (uint256(feeDataArray[i].upliftFeeBps) * 1e18)) / 10000;
```

The issue arises because this fee is then applied to the total withdrawal amount rather than just the uplift portion:

```
localData.feeAmount += (depositAmount * feePerLP);
```

Example scenario:

1. User deposits 1e18 tokens
2. Value increases to 2e18 (100% uplift)
3. With 60% uplift fee, fee is calculated on entire 2e18
4. User receives 8e17 tokens (less than initial 1e18 deposit)

This contradicts the whitepaper which states fees should only apply to "increase in value LPs have received over the value they would have if they had HODLed".

Impact

- Loss of user funds below HODL value
- Contradict the white paper

Tools Used

Manual review

Recommendations

- 1- Modify fee calculation to only apply to the uplift portion:
- 2- Add explicit checks to ensure withdrawal amount after fees cannot fall below initial deposit value

H-04 • Users can avoid paying fees on uplifts by utilising the afterUpdate logic

HIGH

● OPEN

UpliftOnlyExample.sol#L609-L614

LPNFT.sol#L49-L56

Summary

Users can weaponize the `afterUpdate` hook during NFT transfers to dodge paying fees on Up Lifts, since during the hook, it override the `lpTokenDepositValue` that was previously registered by setting it to `lpTokenDepositValueNow` tricking the system to have no uplifts by calling `removeLiquidityProportional` in the same block from their second wallet

Vulnerability Details

During user adding Liquidity, we register the deposit value of LP token here

```
File: UpliftOnlyExample.sol
250:         poolsFeeData[pool][msg.sender].push(
251:             FeeData({
252:                 tokenID: tokenID,
253:                 amount: exactBptAmountOut,
254:                 //this rounding favours the LP
255:                 lpTokenDepositValue: depositValue,
256:                 //known use of timestamp, caveats are known.
257:                 blockTimestampDeposit: uint40(block.timestamp),
258:                 upliftFeeBps: upliftFeeBps
259:             })
260:         );
```

This is used as a reference value to see how much the value of LP token has increased since deposit when calling `removeLiquidityProportional` specifically in `onAfterRemoveLiquidity` Here

```
File: UpliftOnlyExample.sol
480:         if (localData.lpTokenDepositValueChange > 0) {
481:             feePerLP =
482:                 (uint256(localData.lpTokenDepositValueChange) *
483:                 (uint256(feeDataArray[i].upliftFeeBps) * 1e18)) /
484:                 10000;
```

But Users can avoid paying that fee by transferring the NFT to other wallet

```
File: LPNFT.sol
49:     function _update(address to, uint256 tokenId, address auth) internal override returns
(address previousOwner) {
50:         previousOwner = super._update(to, tokenId, auth);
51:         //_update is called during mint, burn and transfer. This functionality is only for
transfer
52:         if (to != address(0) && previousOwner != address(0)) {
53:             //if transferring the record in the vault needs to be changed to reflect the change
in ownership
54:             router.afterUpdate(previousOwner, to, tokenId);
55:         }
56:     }
.....
```

```
File: UpliftOnlyExample.sol
576:     function afterUpdate(address _from, address _to, uint256 _tokenId) public {
586:
587:         int256[] memory prices = IUpdateWeightRunner(_updateWeightRunner).getData(poolAddress);
609:         feeDataArray[tokenIdIndex].lpTokenDepositValue = lpTokenDepositValueNow;
610:         feeDataArray[tokenIdIndex].blockTimestampDeposit = uint32(block.number);
611:         feeDataArray[tokenIdIndex].upliftFeeBps = upliftFeeBps;
614:         poolsFeeData[poolAddress][_to].push(feeDataArray[tokenIdIndex]);
```

As you see, we retrieve the current LP Deposit value and override the old one with it and then assign those values to the new owner `to` in Line 614

Impact

Loss of funds to LP providers and QuantAdmin

Tools Used

Manual Review

Recommendations

Don't override deposit value when transferring, or charge fees during transfers (will require complexity since it will require `vault.unlock()` etc, more gas cost,)

H-05 • Wrong calculation of gradient

HIGH

● OPEN

QuantammGradientBasedRule.sol#L156-L157

Summary

In gradient calculation if we have for example 6 gradients they get packed into 3 in

`intermediateGradientStates`

Calculating the gradient incase of vector lambda it packs with the wrong index forwarding `i` instead of `locals.storageArrayIndex` for `intermediateGradientStates` which assign wrong values of the gradient.

The gradient calculation is to unpack `intermediateGradientStates` then enter a loop and calculate the new values and packing them again in `intermediateGradientStates` the loop go as follow

- getting the value for `i` and `i+1`
- Calculating then packing them with index of `storageArrayIndex` in `intermediateGradientStates`
- then `i+=2` and `++locals.storageArrayIndex`

a simple chart to simplify the process:

INTERMEDIATEGRADIENTSTATES	[0]	[1]	[2]
stored gradient [i]	0,1	2,3	4,5

incase of a vector wrong index is used for `intermediateGradientStates` `i` instead of `storageArrayIndex` so values is packed as follow

INTERMEDIATEGRADIENTSTATES	[0]	[1]	[2]	[3]	[4]
stored gradient	0,1	OldGradient (no new value stored)	2,3	0 (no value as length should be 3)	4,5

incase number of assets 7 the index is done correctly for `intermediateGradientStates` but overwrite one value due to wrong index used in the loop

INTERMEDIATEGRADIENTSTATES	[0]	[1]	[2]	[3]	[4]
stored gradient	0,1	OldGradient (no value stored)	2,3	Last IntermediateValue	4,5

Affected code

```
File: QuantammGradientBasedRule.sol
156:         intermediateGradientStates[_poolParameters.pool][i] = _quantAMMPackTwo128(
157:             locals.intermediateGradientState[i],
158:             locals.secondIntermediateValue
159:         );
160:         unchecked {
161:             i += 2;
162:             ++locals.storageArrayIndex;
163:         }
164:     }
```

Impact

- broken calculation giving wrong calculations affecting all function depends on it
 - QuantAMMGradientBasedRule
 - PowerChannelUpdateRule
 - MomentumUpdateRule
 - ChannelFollowingUpdateRule
 - AntiMomentumUpdateRule
- all roles will calculate wrong weights which is the core of the protocol and break all its logic

Tools Used

Manual review

Recommendations

Use the correct value for index

```
File: QuantammGradientBasedRule.sol
--         intermediateGradientStates[_poolParameters.pool][i] = _quantAMMPackTwo128(
++         intermediateGradientStates[_poolParameters.pool][locals.storageArrayIndex] =
    _quantAMMPackTwo128(
```

H-06 · Wrong calculation of variance

HIGH

● OPEN

QuantammVarianceBasedRule.sol#L196-L202

Summary

In variance calculation it packs each 2 variance in 1 and if number of assets is odd then last will be appended directly at the end.

In case of odd number and a vector lambda the check for `notDivisibleByTwo` is missing leading to wrong values being calculated

logic goes as follow

- if length is even ex: 8

INTERMEDIATEVARIANCESTATES	[0]	[1]	[2]	[3]
indexes packed	0,1	2,3	4,5	6,7

```
- loop go untill 7
  - with i being 0,2,4,6
  - secondIndex 1,3,5,7
```

- if length is 7

INTERMEDIATEVARIANCESTATES	[0]	[1]	[2]	[3]
indexes packed	0,1	2,3	4,5	6

```
- loop go untill 5
  - i is 0,2,4
  - secondIndex 1,3,5
- this store 6 values and the 7th stored separately out of the loop
```

With vector lambda subtracting 1 from `nMinusOne` is missing leading to loop go as follow

INTERMEDIATEVARIANCESTATES	[0]	[1]	[2]	[3]	[4]
indexes packed	0,1	2,3	4,5	6,7	8

with values of index 7,8 is zero leading to wrong calculation and mismatching of future variance calculation

- wrong values will be calculated

- admin can't assign the correct values as length won't match his inputted data

```
File: QuantammVarianceBasedRule.sol
196:    /// @param _numberOfAssets the number of assets in the pool
197:    function _setIntermediateVariance(
198:        address _poolAddress,
199:        int256[] memory _initialValues,
```

```

200:         uint _numberOfAssets
201:     ) internal {
202:         uint storeLength = intermediateVarianceStates[_poolAddress].length;

```

1. wrong values stored as in this example storage length is retrieved from tokens length and `%2 != 0` so value will be retrieved in this example `index[3]` that holds a packed value without unpacking and the wrong value will be used

Impact

- broken calculation giving wrong calculation affecting all function depends on it
 - `MinimumVarianceUpdateRule`

Tools Used

manual review

Recommendations

apply the correct check incase of odd number incsae of vector lambda

```

++         if (locals.notDivisibleByTwo) {
++             unchecked {
++                 --locals.nMinusOne;
++             }
++         }

```

Medium Severity

M-01 • Attacker can completely prevent users from withdrawing in UpliftOnlyExample

MEDIUM

● OPEN

UpliftOnlyExample.sol#L471-L512

UpliftOnlyExample.sol#L616-L622

Summary

Due to the first in last out nature of the `poolsFeeData`, attacker can prevent users from withdrawing liquidity permanently by transferring to them millions of dust NFT positions over filling the `feeData[]` and causing OOG errors during withdrawals

Vulnerability Details

during withdrawals in `UpliftOnlyExample::onAfterRemoveLiquidity()` we start looping from the last deposit (`localData.feeDataArrayLength - 1`)

```

File: UpliftOnlyExample.sol
471:         for (uint256 i = localData.feeDataArrayLength - 1; i ≥ 0; --i) {
493:             if (feeDataArray[i].amount ≤ localData.amountLeft) {
494:                 uint256 depositAmount = feeDataArray[i].amount;

```

```

495:         localData.feeAmount += (depositAmount * feePerLP);
501:         delete feeDataArray[i];
502:         feeDataArray.pop();
507:     } else {
508:         feeDataArray[i].amount -= localData.amountLeft;
509:         localData.feeAmount += (feePerLP * localData.amountLeft);
510:         break;
511:     }
512: }

```

This opens up an attack on large valuable positions by malicious BOTs (especially in L2s where gas price is very cheap), here what would happen:

- 1- Attacker sees very large valuable deposit position
- 2- Attacker deposit dust value positions and transfer the NFT to the victim (taking advantage of `afterUpdate` hook in NFT transfer, pushing the position to the end of `feeDataArray` of victim)
- 3- Attack is repeated millions of times, causing any attempt to withdraw the actual first position of the user to cause OOG reverts
- 4- The victim can't transfer his valuable LP NFT position to other wallet as a solution, since transfer will try to re-order the array of the sender that will cause OOG reverts too

```

File: UpliftOnlyExample.sol
576:     function afterUpdate(address _from, address _to, uint256 _tokenId) public {
590:         FeeData[] storage feeDataArray = poolsFeeData[poolAddress][_from];
616:         if (tokenIdIndex != feeDataArray.length - 1) {
617:             //Reordering the entire array could be expensive but it is the only way to
keep true FILO
618:             for (uint i = tokenIdIndex + 1; i < feeDataArray.length; i++) {
619:                 delete feeDataArray[i - 1];
620:                 feeDataArray[i - 1] = feeDataArray[i];
621:             }
622:         }

```

- 5- the victim will try to remove those dust positions batch by batch to reach his first position, but the malicious bot of the attacker will track the victim `poolsFeeData` after each block (no need to front run, valid attack path on L2s), adding more position to the victim if needed

Attack is feasible since its on L2s and using dust positions to DOS (very small loss overall to attacker to keep the attack running)

This was possible due to:

1. when transferring we don't check the array size of `to` if it has passed 100 or not
2. the FILO nature when withdrawing, and re-ordering when transferring
3. any one can enlarge the size of other people by transferring to them the NFT, and array will grow in `afterUpdate` hook
4. no minimum LP deposits to prevent dusts deposits

Impact

Freeze of funds to the victim

Tools Used

Manual Review

Recommendations

- implement minimum LP deposits
- check for length of `feedataArray` of the `to` during transfers and prevent it it will exceed
- give a way to withdraw specific position index (if FILO implementation is not very important)
- give the ability to guard against who can transfer his NFT position to me (set by user him self)

M-02 • Division before multiplication in `lpTokenDepositValueChange` causes loss of fees

MEDIUM

OPEN

`UpliftOnlyExample.sol#L474-L490`

Summary

in `UpliftOnlyExample::onAfterRemoveLiquidity` There is a division before multiplication in `lpTokenDepositValueChange` usage logic, causing any change of deposit value below 100% truncates to 0, and any change that is not perfect % of `lpTokenDepositValue` will have the remainder truncated

Vulnerability Details

in `UpliftOnlyExample::onAfterRemoveLiquidity` Hook, there is a calculation to check for `lpTokenDepositValue` during withdrawals to see if any uplift happened, according to that, its decided to use `upliftFeeBps` or `minWithdrawalFeeBps`

```
File: UpliftOnlyExample.sol
474:         localData.lpTokenDepositValueChange =
475:             (uint256(localData.lpTokenDepositValueNow) -
uint256(localData.lpTokenDepositValue)) /
476:             uint256(localData.lpTokenDepositValue);
477:
478:         uint256 feePerLP;
480:         if (localData.lpTokenDepositValueChange > 0) {
481:             feePerLP =
482:                 (uint256(localData.lpTokenDepositValueChange) *
(uint256(feedataArray[i].upliftFeeBps) * 1e18)) /
483:                 10000;
484:         }
486:         else {
489:             feePerLP = (uint256(minWithdrawalFeeBps) * 1e18) / 10000;
490:         }
```

But as you see in Line 475, we get the ratio of change by dividing by the old `lpTokenDepositValue` before we multiply this ration by 1e18 in Line 482

Here is an example of what would happen

- `lpTokenDepositValueNow` = 15e18
- `lpTokenDepositValue` = 10e18

$$\text{lpTokenDepositValueNow} - \text{lpTokenDepositValue} = 15e18 - 10e18 = 5e18$$

$$5e18 / \text{lpTokenDepositValue} = 5e18 / 10e18 = 0$$

$$\text{lpTokenDepositValueChange} = 0$$

Then the logic in else block will be the one to be executed, since the line 480 `if` evaluate to false

Also, if the uplift happened was 350%, then the 50% will truncate ($35e18 - 10e18 / 10e18 = 3e18$)

Impact

Loss of fees (funds) for LP providers and Quant AMM admin

Tools Used

Manual Review

Recommendations

multiply during `lpTokenDepositValueChange` calculation first (as seen below)

```
File: UpliftOnlyExample.sol
474:         localData.lpTokenDepositValueChange =
475:             (uint256(localData.lpTokenDepositValueNow) -
uint256(localData.lpTokenDepositValue) * 1e18) /
476:             uint256(localData.lpTokenDepositValue);
477:
478:         uint256 feePerLP;
479:         // if the pool has increased in value since the deposit, the fee is calculated
based on the deposit value
480:         if (localData.lpTokenDepositValueChange > 0) {
481:             feePerLP =
482:                 (uint256(localData.lpTokenDepositValueChange) *
(uint256(feeDataArray[i].upliftFeeBps))) /
483:                 10000;
484:         }
485:         // if the pool has decreased in value since the deposit, the fee is calculated
based on the base value - see wp
486:         else {
487:             //in most cases this should be a normal swap fee amount.
488:             //there always must be at least the swap fee amount to avoid deposit/withdraw
attack surgance.
489:             feePerLP = (uint256(minWithdrawalFeeBps) * 1e18) / 10000;
490:         }
```

M-03 • In UpliftOnlyExample, the immutable updateWeightRunner address prevents seamless contract migration

MEDIUM

● OPEN

UpliftOnlyExample.sol

Summary

The immutable `_updateWeightRunner` address causes overhead during migration to new `UpdateWeightRunner` contracts, forcing admins to maintain fee configurations across multiple deprecated contracts to support existing `UpliftOnlyExample`.

Vulnerability Details

When `_updateWeightRunner` is set as immutable in `UpliftOnlyExample`, hooks created with older `UpdateWeightRunner` contracts cannot be updated to point to newer versions. This creates a fragmented system where:

1. New hooks use the latest `UpdateWeightRunner`
2. Old hooks remain tied to deprecated `UpdateWeightRunner` contracts
3. Admins must maintain fee configurations across all versions to ensure consistent behavior since this is the only usage of `_updateWeightRunner` beside retrieving admin address

Impact

- Increased operational complexity for admins managing multiple `UpdateWeightRunner` instances
- Risk of inconsistent fee structures across different hook generations

Tools Used

Manual review

Recommendations

- Remove the immutable modifier and implement an upgradeable pattern:

```
address private _updateWeightRunner;
function setUpdateWeightRunner(address newRunner) external onlyOwner {
    _updateWeightRunner = newRunner;
    emit UpdateWeightRunnerChanged(newRunner);
}
```

M-04 • Last withdrawer will donate fees to empty pool, allowing MEV and leaving stuck funds in the pool

MEDIUM

● OPEN

UpliftOnlyExample.sol#L555-L567

Summary

in `onAfterRemoveLiquidity()` fees of the withdrawn tokens amounts (whatever its upLift or the minimum fees) of the last user withdrawing from the pool will be donated to empty pool.

Vulnerability Details

NOTE: the bug about `accruedQuantAMMFees` wrong distribution since Admin don't have position registered in `poolsFeeData` Is assumed to be solved by sending actual tokens to the admin

in `onAfterRemoveLiquidity()` The code calculate fees on the withdrawn tokens amounts (whatever its upLift or the minimum fees) and donates part of it to the pool and part of it to the QuantAMM admin, Here:

```
File: UpliftOnlyExample.sol
function onAfterRemoveLiquidity(
536:     if (localData.adminFeePercent > 0) {
537:         _vault.addLiquidity(
538:             AddLiquidityParams({
539:                 pool: localData.pool,
540:                 to: IUpdateWeightRunner(_updateWeightRunner).getQuantAMMAdmin(),
541:                 maxAmountsIn: localData.accruedQuantAMMFees,
542:                 minBptAmountOut: localData.feeAmount.mulDown(localData.adminFeePercent) /
1e18,
543:                 kind: AddLiquidityKind.PROPORTIONAL,
544:                 userData: bytes("")
545:             })
546:         );
547:         emit ExitFeeCharged(
548:             userAddress,
549:             localData.pool,
550:             IERC20(localData.pool),
551:             localData.feeAmount.mulDown(localData.adminFeePercent) / 1e18
552:         );
553:     }
554:
555:     if (localData.adminFeePercent != 1e18) {
556:         // Donates accrued fees back to LPs.
557:         _vault.addLiquidity(
558:             AddLiquidityParams({
559:                 pool: localData.pool,
560:                 to: msg.sender,
561:                 maxAmountsIn: localData.accruedFees,
562:                 minBptAmountOut: 0,
563:                 kind: AddLiquidityKind.DONATION,
564:                 userData: bytes("")
565:             })
566:         );
567:     }
```

But the problem is that There are alot of circumstances where user withdrawing is the last withdrawer of the pool, what happens is:

1. User will get charged fees, part of it added to the admin (in the correct way) and part of it is donated to the pool
2. this will create an empty pool that have tokens in it with no corresponding BPT

3. MEV can immediately add liquidity (to mint BPT) and remove liquidity (burning the BPT and getting the initially deposited tokens Plus the stuck tokens)
4. The problem is yet that on MEV interactions, they will be charged fees and donated to the again empty pool (after MEV removing liquidity)
5. This will create some tokens in the end that is never retrievable and always stuck in the pool

The above scenario describes what happens when the first mentioned bug get fixed by sending fees to the admin as actual tokens

Now if they decide to solve it by registering admin fee position in `poolsFeeData` then

1. Last withdrawer gets charged fees that part of it mint liquidity to admin, and part of it gets donated to the pool (that now have the admin liquidity, meaning that the admin owns all token Pools)
2. Admin comes to remove liquidity and he himself will be charged fee that again that of it mint liquidity to admin, and part of it gets donated to the pool
3. Again and again and again (yet pool will have stuck funds either way after last withdrawer)

Impact

Stuck funds in the pool

Tools Used

Manual Review

Recommendations

if the withdrawer is the last withdrawer (vault supply is 0) send all fees to the admin, or don't charge fees (like any pool does with the last withdrawer, they simply transfer all the balance of the pool to him)

M-05 • Moving average length validation prevents admin override for rules requiring historical data

MEDIUM

● OPEN

QuantammMathMovingAverage.sol

UpdateWeightRunner.sol

Summary

The `initialisePoolRuleIntermediateValues()` function in `UpdateRule` fails to properly handle rules like `MinimumVarianceUpdateRule` that require double the number of moving averages (current + previous) due to a length validation check in `_setInitialMovingAverages()`.

Vulnerability Details

The issue occurs in the following call flow:

```
// UpdateWeightRunner.sol
function setIntermediateValuesManually(
    address _poolAddress,
```

```

    int256[] memory _newMovingAverages,
    int256[] memory _newParameters,
    uint _numberOfAssets
) external {
    // ... auth checks ...
    rule.initialisePoolRuleIntermediateValues(_poolAddress, _newMovingAverages, _newParameters,
    _numberOfAssets);
}

```

```

// QuantammMathMovingAverage.sol
function _setInitialMovingAverages() internal {
    if (movingAverageLength == 0 || _initialMovingAverages.length == _numberOfAssets) {
        movingAverages[_poolAddress] = _quantAMMPack128Array(_initialMovingAverages);
    } else {
        revert("Invalid set moving avg");
    }
}

```

The `MinimumVarianceUpdateRule` requires double the assets count for moving averages:

```

uint16 private constant REQUIRES_PREV_MAVG = 1;

```

And this how storage should look for that rule by DEV comment

```

// QuantAMMMathMovingAverage.sol
// this can be just the moving averages per token, or if prev moving average is true then it is
[...moving averages, ...prev moving averages]
mapping(address => int256[]) public movingAverages;

```

The validation check `_initialMovingAverages.length == _numberOfAssets` prevents setting both current and previous moving averages, only allowing current ones to be set.

This will leave two options to the admin:

1. Not using the function
2. Use the function setting only moving average of the length of number of assets

If its the first one, then the availability of function is lost

If its the second option, then other integrators of that stored previous moving averages will be corrupted

Impact

- Admin cannot properly set/override moving averages for rules requiring previous values
- Loss of historical moving average data when manual override is needed
- Breaks integrations from other protocols relying on both current and previous moving average values

Tools Used

Manual review

Recommendations

Modify the validation in `_setInitialMovingAverages()` to handle rules requiring different moving average lengths or there can be a check to assure that new values are the same length as `movingAverageLength` retrieved from the mapping

M-06 • Pool weights are non-updatable after updating the `updateWeightRunner` address

MEDIUM

● OPEN

QuantAMMWeightedPool.sol

UpdateWeightRunner.sol

Summary

The `setUpdateWeightRunnerAddress` function in `QuantAMMWeightedPool` allows changing the `UpdateWeightRunner` contract address, but the pool has no way to set rules on the new contract since `_setRule()` is only callable during initialization. This is particularly problematic because `UpdateWeightRunner` maintains multiple pool-specific mappings keyed by `msg.sender`.

Vulnerability Details

The vulnerability exists because:

1- `UpdateWeightRunner` mappings keyed to `msg.sender` (pool address):

```
rules[msg.sender] = _poolSettings.rule; //Only pool can set its own rules
poolOracles[msg.sender] = optimisedHappyPathOracles; //Pool oracle settings
poolBackupOracles[msg.sender] = _poolSettings.oracles; //Pool backup oracles
poolRuleSettings[msg.sender] = PoolRuleSettings({ //Pool rule configuration
    lambda: _poolSettings.lambda,
    epsilonMax: _poolSettings.epsilonMax,
    absoluteWeightGuardRail: _poolSettings.absoluteWeightGuardRail,
    ruleParameters: _poolSettings.ruleParameters,
    timingSettings: PoolTimingSettings({ updateInterval: _poolSettings.updateInterval,
    lastPoolUpdateRun: 0 }),
    poolManager: _poolSettings.poolManager
});
```

2- Rules are set through `_setRule()` which is only called during `initialize()` function with initializer modifier

3- When `setUpdateWeightRunnerAddress()` changes the `UpdateWeightRunner`:

```
function setUpdateWeightRunnerAddress(address _updateWeightRunner) external override {
    require(msg.sender == quantammAdmin, "ONLYADMIN");
    updateWeightRunner = UpdateWeightRunner(_updateWeightRunner);
    emit UpdateWeightRunnerAddressUpdated(address(updateWeightRunner), _updateWeightRunner);
}
```

The pool has no mechanism to call `setRuleForPool()` on the new contract since:

- `_setRule()` is locked behind initializer

- No other function exists to set rules post-initialization

Since the pool can't call `setRuleForPool()`, then the new `UpdateWeightRunner` contract will have no way to interact or call the pool since almost in every action to a pool, it uses the variables from the mappings `rules` or `poolRuleSettings` etc, for example:

```
function performUpdate(address _pool) public {
    address rule = address(rules[_pool]);
    require(rule != address(0), "Pool not registered");
    PoolRuleSettings memory settings = poolRuleSettings[_pool];
```

Impact

When `UpdateWeightRunner` address is changed:

- Pool becomes permanently unable to update weights since:
 - * Rules cannot be set on new contract
 - * All pool configurations are lost (rules, oracles, settings)
 - * Only the pool itself can set its configurations (msg.sender mappings)
 - * No post-initialization setting mechanism exists
- Core pool functionality is broken

Tools Used

- Manual code review

Recommendations

1. Add a new function to allow setting rules after `UpdateWeightRunner` changes:

```
function migrateRules(
    int256[] memory _initialWeights,
    int256[] memory _ruleIntermediateValues,
    int256[] memory _initialMovingAverages,
    IQuantAMMWeightedPool.PoolSettings memory _poolSettings
) external {
    require(msg.sender == quantammAdmin, "ONLYADMIN");
    _setRule(_initialWeights, _ruleIntermediateValues, _initialMovingAverages, _poolSettings);
}
```

M-07 • Single storage variable used for both swap and uplift fees

MEDIUM

● OPEN

UpdateWeightRunner.sol

Summary

The `UpdateWeightRunner` contract incorrectly uses the same storage variable `quantAMMSwapFeeTake` for both swap fees and uplift fees, leading to incorrect fee calculations in the `UpliftOnlyExample` contract.

Vulnerability Details

In `UpdateWeightRunner` :

- Both `setQuantAMMUpLiftFeeTake()` and `setQuantAMMSwapFeeTake()` modify the same storage variable `quantAMMSwapFeeTake`
- `getQuantAMMUpLiftFeeTake()` returns `quantAMMSwapFeeTake`

In `UpLiftOnlyExample` :

- Line 331: Retrieves uplift fee using `getQuantAMMUpLiftFeeTake()`
- Line 519: Uses the same function for swap fee calculations

This means both uplift fees and swap fees will always be identical, which is likely not the intended behavior, (evidenced from the two setter and getter functions too)

Impact

- Incorrect fee calculations as uplift fees and swap fees cannot be set independently

Tools Used

Manual Review

Recommendations

1- Add separate storage variable for uplift fees:

```
uint256 public quantAMMUpLiftFeeTake;  
uint256 public quantAMMSwapFeeTake;
```

2- Update getter functions to return correct variables:

```
function getQuantAMMUpLiftFeeTake() external view returns (uint256) {  
    return quantAMMUpLiftFeeTake;  
}  
function getQuantAMMSwapFeeTake() external view returns (uint256) {  
    return quantAMMSwapFeeTake;  
}
```

3- Ensure setter functions modify their respective variables

M-08 • Users transferring their NFT position will retroactively get the new upliftFeeBps

MEDIUM

● OPEN

UpLiftOnlyExample.sol#L611-L614

Summary

When user transfer his NFT LP position to another wallet, the new `upliftFeeBps` is registered to `poolsFeeData` of the `to`, this contradict the intended design to have no retroactive `upliftFeeBps` applied to already open position and also opens an attack surface to dodge the old high fees for the current smaller ones

Vulnerability Details

NOTE: its worth mentioning that this is a different bug from the bug that talks about complete dodging of UpLift fees through `lpTokenDepositValue` overriding

When user adds liquidity, the `upliftFeeBps` is registered in his `poolsFeeData` so that any future change doesn't apply to his position retroactively

```
File: UpliftOnlyExample.sol
219:     function addLiquidityProportional()
249:
250:         poolsFeeData[pool][msg.sender].push(
251:             FeeData({
252:                 tokenID: tokenID,
253:                 amount: exactBptAmountOut,
254:                 //this rounding favours the LP
255:                 lpTokenDepositValue: depositValue,
256:                 //known use of timestamp, caveats are known.
257:                 blockTimestampDeposit: uint40(block.timestamp),
258:                 upliftFeeBps: upliftFeeBps
259:             })
260:         );
263:     }
```

But the problem is that in `afterUpdate` hook that is triggered on NFT transfers, the `upliftFeeBps` of `poolsFeeData` of `to` is set to the current `upliftFeeBps`

```
File: UpliftOnlyExample.sol
576:     function afterUpdate(address _from, address _to, uint256 _tokenId) public {
609:         feeDataArray[tokenIdIndex].lpTokenDepositValue = lpTokenDepositValueNow;
610:         feeDataArray[tokenIdIndex].blockTimestampDeposit = uint32(block.number);
611:         feeDataArray[tokenIdIndex].upliftFeeBps = upliftFeeBps;
612:
613:         //actual transfer not a afterTokenTransfer caused by a burn
614:         poolsFeeData[poolAddress][_to].push(feeDataArray[tokenIdIndex]);
```

this causes two problems:

1. New fees are applied retroactively, which is not the intention of the protocol (Broken functionality)
2. There can be scenarios where `upliftFeeBps` was set by the admin to lower values than the ones the user initially deposited at (ie, events of lower fees, or simply the admin wants to do it due to increased competition in the field that provided less fees, etc)

* When that happens, the user can transfer the NFT to other wallet of his own so that the `upliftFeeBps` variable gets overridden by the new one

Impact

- Non intended design
- User can decide to override their `upliftFeeBps` when they see it less costly to pay (overall profitable to them)

* Leading to loss of funds (Fees) to the Quant Admin and LP providers (since there should have been larger fees donated to their pool)

Tools Used

Manual review

Recommendations

Don't override that variable, simply remove Line 611

Low Severity

L-01 • Missing implementation of a function to change `upliftFee`

LOW● OPEN`UpliftOnlyExample.sol`

Summary

`upliftFeeBps` Is the uplift fee in basis points (1/10000) for the pool the pool creator should have the ability to change this value for future deposits as stated by the sponsor [here](#) but there is no function anywhere to do so It is also stated by the sponsorto flag any missing implementation

scenario

pool creator set the `upliftFeeBps` to 0 as a start then adjust it later then after deployment and some deposits he found out he couldn't change the value

this could be all the way around with fees being too high and not being set back to decent value

Impact

- function not implemented breaks contract design forcing the creator to static `upliftFeeBps`
- loss of (funds / fees) to the creator

Tools Used

manual review

Recommendations

implement a function to change `upliftFeeBps`

```
++ function changeUpliftFeeBps(uint256 newUpliftFeeBps) external onlyowner {
++     uint256 oldUpliftFeeBps = upliftFeeBps;
++     require(newUpliftFeeBps ≤ 10000, "upliftFeeBps must be less than 10000")
++     upliftFeeBps = newUpliftFeeBps;
++     emit upliftFeeBpsChanged(upliftFeeBps,oldUpliftFeeBps)
++ }
```

L-02 · Wrong value added to blockTimestampDeposit in afterUpdate

LOW

● OPEN

UpliftOnlyExample.sol#L610

Summary

In `afterUpdate` function the `feeDataArray` holds `blockTimestampDeposit` which is the timestamp of when the index of this array has been started

In `addLiquidityProportional` it hold `block.timestamp` value but

In `afterUpdate` it hold `block.number`

Impact

- Function is broken and not working as intended

Tools Used

Manual review

Recommendations

Use `block.timestamp`

L-03 · minWithdrawalFeeBps is not added to upliftFeeBps, causing loss of fees and allowing MEV actions

LOW

● OPEN

UpliftOnlyExample.sol#L480-L490

Summary

Note!: This bug assumes that `upliftFeeBps` is applied in the upLifted value only as intended in the whitePaper and assumes the rounding down to 0 of `lpTokenDepositValueChange` is solved

The `UpliftOnlyExample` contract's uplift fee calculation can result in fees lower than the intended minimum when small uplifts occur, potentially enabling MEV attacks that were meant to be prevented by `minWithdrawalFeeBps`.

Vulnerability Details

Current implementation uses an if/else block that chooses between fees types during liquidity removal:

```
if (localData.lpTokenDepositValueChange > 0) {
    feePerLP = (uint256(localData.lpTokenDepositValueChange) *
        (uint256(feeDataArray[i].upliftFeeBps) * 1e18)) / 10000;
} else {
    feePerLP = (uint256(minWithdrawalFeeBps) * 1e18) / 10000;
}
```

When calculating fees for uplifted positions:

```
if (localData.lpTokenDepositValueChange > 0) {
    feePerLP = (uint256(localData.lpTokenDepositValueChange) *
        (uint256(feeDataArray[i].upliftFeeBps) * 1e18)) / 10000;
}
```

Example scenario:

1. `minWithdrawalFeeBps` = 0.5% (50 bps)
2. `upliftFeeBps` = 50% (5000 bps)
3. MEV deposits 100e18
4. Gets 1% uplift (1e18)
5. Fee calculation: $1e18 * 50\% = 5e17$ (0.05% of total deposit)
6. Actual fee (0.05%) < `minWithdrawalFeeBps` (0.5%)

This creates a gap where MEV can extract value while paying less than the intended minimum fee.

Impact

The vulnerability enables:

1. MEV attacks with fees below intended minimum (for example, just in time liquidity for large swaps and feeless swaps attacks, etc)
2. Potential value extraction through rapid deposit/withdraw cycles attacks

Tools Used

Manual review

Recommendations

add the `minWithdrawalFeeBps` to the `upliftFeeBps`

Appendix

Tools used

- Foundry

About Parapet Labs

Parapet Labs does smart contract audits for teams that can't afford to get drained. The audit ends, but the wall doesn't: we offer priority re-audits, upgrade reviews and a direct line to the auditors who know your code.

<https://parapetlabs.net>