QuantAMM V1 Audit Report

Prepared by Cyfrin

Version 1.2

Lead Auditors

0kage

immeas

December 17, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

QuantAMM is the first protocol to utilize Temporal Function Market Making as a trade execution mechanism for on-chain quantitative asset management strategies. Built on Balancer V3, each pool receives periodic updates with new weights derived from on-chain trading strategies. These weights are then linearly adjusted using multipliers until the next update is executed. This process dynamically alters the composition of the pool balances based on the optimal allocation determined by the strategies.

4.1 Actors and Roles

1. **Actors:**
   - **Portfolio Managers:** Deposit into QuantAMM pools to leverage various strategies.
   - **Arbitrageurs:** Utilize weight changes to arbitrage and adjust the pool composition to align with the strategy's target.
   - **QuantAMM:** Provides and monitors pools, and can, in emergencies, use break-glass tools to adjust internal pool parameters, ensuring strategies and pools function as intended.
2. **Roles:**
   - `quantammAdmin`: A protocol-managed account that can add or remove trusted oracles, manage pool permissions, and use break-glass tools to adjust strategy parameters and pool weights when necessary.

4.2 Key Components

1. **QuantAMMWeightedPool:**
   An extension of the Balancer V3 Weighted Pool that supports up to 8 tokens. Within a single block, it behaves like a standard Balancer V3 Weighted Pool. Between blocks, the weights are adjusted using multipliers provided by the pool's trading strategy.

2. **UpdateWeightRunner:**
   A singleton contract responsible for managing weight updates for all pools. When called for a specific pool, it calculates the new adjusted weights based on the pool's strategy and updates the pool accordingly.

3. **Rules:**
   Trading strategies currently supported are *Momentum*, *Anti-Momentum*, *Power Channel*, and *Minimum Variance*. Each rule calculates new weights and multipliers for a pool based on the latest oracle prices. Weight updates are subject to guardrails that prevent extreme weight values and large amplitude changes. This helps mitigate potential multiblock MEV that could otherwise occur if weights shifted significantly between blocks. Further reading can be found in their paper on this topic.

5 Audit Scope

Cyfrin conducted an audit of One World Project based on the code present in the repository commit hash `7213401`.

The following contracts were included in the scope of the audit:

```
- pkg/pool-quantamm/contracts/QuantAMMWeightedPoolFactory.sol
- pkg/pool-quantamm/contracts/MultiHopOracle.sol
- pkg/pool-quantamm/contracts/ChainlinkOracle.sol
- pkg/pool-quantamm/contracts/QuantAMMWeightedPool.sol
- pkg/pool-quantamm/contracts/UpdateWeightRunner.sol
- pkg/pool-quantamm/contracts/QuantAMMBaseAdministration.sol
- pkg/pool-quantamm/contracts/QuantAMMStorage.sol
- pkg/pool-quantamm/contracts/DaoOperations.sol
- pkg/pool-quantamm/contracts/rules/MomentumUpdateRule.sol
- pkg/pool-quantamm/contracts/rules/MinimumVarianceUpdateRule.sol
- pkg/pool-quantamm/contracts/rules/AntimomentumUpdateRule.sol
- pkg/pool-quantamm/contracts/rules/PowerChannelUpdateRule.sol
- pkg/pool-quantamm/contracts/rules/UpdateRule.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammCovarianceBasedRule.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammMathMovingAverage.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammMathGuard.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammGradientBasedRule.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammVarianceBasedRule.sol
- pkg/pool-quantamm/contracts/rules/base/QuantammBasedRuleHelpers.sol
```

6 Executive Summary

Over the course of 20 days, the Cyfrin team conducted an audit on the QuantAMM V1 smart contracts provided by QuantAMM. In this period, a total of 19 issues were found.

The audit identified four high-severity findings. Three of these were related to the gas optimization method used to pack weights and multipliers for up to 8 tokens into just two storage slots. This optimization introduced significant complexity in handling the packing and accessing of weights. Consequently, errors occurred in pools with more than 4 tokens, where swaps, deposits, and withdrawals could either revert or use incorrect weights for balance and invariant calculations. Many of these issues can be explained by that the Balancer team decided to change their decision to support only four tokens in favor of eight instead. This lead to the addition of 4+ tokens late in the development process. The fourth high-severity finding concerned the handling of negative multipliers (for weight decreases between blocks). An erroneous conversion to unsigned integers caused swaps involving tokens with negative multipliers to revert.

The audit also identified seven medium-severity findings. Three of these findings pertained to the `QuantAMM-BaseAdministration` contract, which was designed to consolidate all break-glass calls into a single contract. Ultimately, this contract was deemed unnecessary. The call transformations previously handled by the admin contract were removed, and the protocol chose to manage the `quantammAdmin` account directly with an OpenZeppelin timelock contract. Of the remaining four medium-severity findings, two involved faulty access controls, one was related to an error in the packing of weights, and the last highlighted the risk of using stale oracle prices.

The test suite was comprehensive, particularly for 2-token pools. The protocol extended Balancer's existing regression tests to cover the QuantAMMWeightedPool. However, the coverage for pools with more than 2 tokens proved insufficient, as evidenced by the high-severity findings. To address this, the Cyfrin team built upon the existing tests by creating a stateless fuzz suite for pools of any size and configuration. These tests were provided to the protocol at the end of the audit.

At the end of the audit, the QuantAMM team highlighted a potential attack vector. The concern was that slight price changes caused by each weight update could be exploited through unbalanced deposits and withdrawals, similar to the multi-block trade attack discussed in their paper (mitigated by maximum trade sizes and guard rails). The QuantAMM team developed a mathematical simulation demonstrating that the attack would not be profitable under reasonable pool parameters. To further verify this, the Cyfrin team created a single-token in/out fuzz test. The results confirmed that with reasonable parameters (a 3% guard rail and an epsilon max of 1%), the attack would not be profitable. This fuzz test was handed over to the QuantAMM team.

Summary

Project Name	QuantAMM V1
Repository	QuantAMM-V1
Commit	7213401491f6...
Audit Timeline	Nov 18th - Dec 13th
Methods	Manual Review, Stateless Fuzzing

Issues Found

Critical Risk	0
High Risk	4
Medium Risk	7
Low Risk	3
Informational	5
Gas Optimizations	0
Total Issues	19

Summary of Findings

[H-1] Misconfigured index boundaries prevent certain swaps in QuantAMMWeightedPool	Resolved
[H-2] Token Index error in `getNormalizedWeights` calculates incorrect weight leading to incorrect pool asset allocation	Resolved
[H-3] Incorrect handling of negative multipliers in `QuantAMMWeightedPool` leads to underflow in weight calculation	Resolved

[H-4] Mismatch in multiplier position causes incorrect weight calculations in `QuantAMMWeightedPool`	Resolved
[M-1] Incorrect range validation in `quantAMMPackEight32` allows invalid values to be packed	Resolved
[M-2] Lack of permission check allows unauthorized updates to `lastPoolUpdateRun`	Resolved
[M-3] Faulty permission check in `UpdateWeightRunner::getData` allows unauthorized access	Resolved
[M-4] Stale Oracle prices accepted when no backup oracles available	Acknowledged
[M-5] Missing admin functions prevent execution of key `UpdateWeightRunner` actions	Resolved
[M-6] `QuantAMMBaseAdministration::onlyExecutor` modifier does not enforce a time lock on actions	Resolved
[M-7] Unauthorized role grants prevent `QuantAMMBaseAdministration` deployment	Resolved
[L-1] `UpdateWeightRunner::performUpdate` reverts on underflow when new multiplier is zero	Resolved
[L-2] Missing weight sum validation in `setWeightsManually` function	Acknowledged
[L-3] Extreme weight ratios combined with large balances can cause denial-of-service for unbalanced liquidity operations	Resolved
[I-1] Use of solidity `assert` instead of foundry asserts in test suite	Resolved
[I-2] Consider adding a getter for `QuantAMMWeightedPool.poolDetails`	Resolved
[I-3] `AntimomentumUpdateRule.parameterDescriptions` initialized too long	Resolved
[I-4] `TODO`s left in rule parameter descriptions	Resolved
[I-5] Unnecessary `_initialWeights` sum check	Resolved

7 Findings

7.1 High Risk

7.1.1 Misconfigured index boundaries prevent certain swaps in QuantAMMWeightedPool

Description: Balancer pools can handle up to 8 tokens in weighted pools. QuantAMM uses this feature but optimizes storage by using only two slots. As a result, the first four tokens and the last four tokens are treated separately.

For swaps, this logic is implemented in QuantAMMWeightedPool::onSwap:

```
// if both tokens are within the first storage element
if (request.indexIn < 4 && request.indexOut < 4) {
    QuantAMMNormalisedTokenPair memory tokenWeights = _getNormalisedWeightPair(
        request.indexIn,
        request.indexOut,
        timeSinceLastUpdate,
        totalTokens
    );
    tokenInWeight = tokenWeights.firstTokenWeight;
    tokenOutWeight = tokenWeights.secondTokenWeight;
} else if (request.indexIn > 4 && request.indexOut < 4) {
    // if the tokens are in different storage elements
    QuantAMMNormalisedTokenPair memory tokenWeights = _getNormalisedWeightPair(
        request.indexOut,
        request.indexIn,
        timeSinceLastUpdate,
        totalTokens
    );
    tokenInWeight = tokenWeights.firstTokenWeight;
    tokenOutWeight = tokenWeights.secondTokenWeight;
} else {
    tokenInWeight = _getNormalizedWeight(request.indexIn, timeSinceLastUpdate, totalTokens);
    tokenOutWeight = _getNormalizedWeight(request.indexOut, timeSinceLastUpdate, totalTokens);
}
```

The issue arises in the second `else if` block, which is intended to handle cases where both tokens are in the second slot. However, the condition `request.indexOut < 4` is incorrect; it should be `request.indexOut >= 4`. Consequently, in _getNormalisedWeightPair, the code will revert:

```
function _getNormalisedWeightPair(
    uint256 tokenIndexOne, // @audit indexOut
    uint256 tokenIndexTwo, // @audit indexIn
    // ...
) internal view virtual returns (QuantAMMNormalisedTokenPair memory) {
    uint256 firstTokenIndex = tokenIndexOne; // @audit indexOut
    uint256 secondTokenIndex = tokenIndexTwo; // @audit indexIn
    // ...
    if (tokenIndexTwo > 4) { // @audit indexIn will always be > 4 from the branch above
        firstTokenIndex = tokenIndexOne - 4; // @audit indexOut, must be < 4 hence underflow
        secondTokenIndex = tokenIndexTwo - 4;
        totalTokensInPacked -= 4;
        targetWrappedToken = _normalizedSecondFourWeights;
    } else {
```

Here, `tokenIndexOne - 4` will underflow due to the incorrect condition.

In QuantAMMWeightedPool::_getNormalizedWeight, there is a similar issue where the condition `tokenIndex > 4` should be `tokenIndex >= 4`:

```
if (tokenIndex > 4) { // @audit should be >=
    // get the index in the second storage element
    index = tokenIndex - 4;
    targetWrappedToken = _normalizedSecondFourWeights;
    tokenIndexInPacked -= 4;
} else {
    // @audit first slot
    if (totalTokens > 4) {
        tokenIndexInPacked = 4;
    }
    targetWrappedToken = _normalizedFirstFourWeights;
```

This causes a failure when fetching the multiplier in `QuantAMMWeightedPool::_calculateCurrentBlockWeight`:

```
int256 blockMultiplier = tokenWeights[tokenIndex + (tokensInTokenWeights)];
```

In this case, `tokenWeights` contains 8 entries, but `tokenIndex + (tokensInTokenWeights)` equals 8, causing an array out-of-bounds error.

The same issue is also present in the following locations, though they currently have no impact:

- In `QuantAMMWeightedPool::onSwap`:

  ```
  } else if (request.indexIn > 4 && request.indexOut < 4) { // @audit should be >=
  ```

- In `QuantAMMWeightedPool::_getNormalisedWeightPair`:

  ```
  if (tokenIndexTwo > 4) { // @audit should be >=
  ```

Impact:

1. Swaps where `indexIn > 4` and `indexOut < 4` are impossible.
2. Swaps involving token index 4 cannot be performed.
3. If swaps with `indexIn > 4` and `indexOut < 4` would be possible, the swap amounts would be calculated incorrectly.

Proof of Concept: Add the following tests to `pkg/pool-quantamm/test/foundry/QuantAMMWeightedPool8Token.t.sol` to demonstrate the issues:

```
// cross swap not working correctly
function testGetNormalizedWeightOnSwapOutGivenInNBlocksAfterToken7Token3() public {
    testParam memory firstWeight = testParam(7, 0.1e18, 0.001e18); // indexIn > 4
    testParam memory secondWeight = testParam(3, 0.15e18, 0.001e18); // indexOut < 4
    // will revert on underflow
    _onSwapOutGivenInInternal(firstWeight, secondWeight, 2, 1.006410772600252500e18);
}
// index >= 4 not working correctly
function testGetNormalizedWeightOnSwapOutGivenInInitialToken0Token4() public {
    testParam memory firstWeight = testParam(0, 0.1e18, 0.001e18);
    testParam memory secondWeight = testParam(4, 0.15e18, 0.001e18);
    // fails with `panic: array out-of-bounds access`
    _onSwapOutGivenInInternal(firstWeight, secondWeight, 0, 0.499583703357018000e18);
}
// index >= 4 not working correctly
function testGetNormalizedWeightOnSwapOutGivenInInitialToken4Token0() public {
    testParam memory firstWeight = testParam(4, 0.1e18, 0.001e18);
    testParam memory secondWeight = testParam(0, 0.15e18, 0.001e18);
```

```
    // fails with `panic: array out-of-bounds access`
    _onSwapOutGivenInInternal(firstWeight, secondWeight, 0, 0.887902403682279000e18);
}
```

Recommended Mitigation:

1. **Refactor the** `if` **block** in `QuantAMMWeightedPool::onSwap` as follows:

```
if (request.indexIn < 4 && request.indexOut < 4 || request.indexIn >= 4 && request.indexOut >=
↪   4) {
    // Same slot; _getNormalisedWeightPair handles the correct logic
    tokenWeights = _getNormalisedWeightPair(...);
} else {
    // Cross-slot handling
    tokenWeights = _getNormalizedWeight(...);
}
```

2. **Update** conditions `> 4` to `>= 4` at the following lines:
   - Line 320
   - Line 383

QuantAmm: Fixed in `414d4bc`

Cyfrin: Verfied.

7.1.2 Token Index error in `getNormalizedWeights` calculates incorrect weight leading to incorrect pool asset allocation

Description: The `_getNormalizedWeights()` function in `QuantAMMWeightedPool.sol` contains a critical indexing error when handling multipliers for pools with more than 4 tokens. The error causes incorrect multiplier values to be used when calculating interpolated weights for tokens 4-7, potentially leading to severe asset allocation errors.

The issue occurs in the _getNormalizedWeights() function when handling the second storage slot for pools with more than 4 tokens:

```
function _getNormalizedWeights() internal view virtual returns (uint256[] memory) {
    // ...
    uint256 tokenIndex = totalTokens;
    if (totalTokens > 4) {
        tokenIndex = 4;  // @audit Sets to 4 for first slot multipliers
    }

    // First slot handles correctly
    normalizedWeights[0] = calculateBlockNormalisedWeight(
        firstFourWeights[0],
        firstFourWeights[tokenIndex],  // Uses indices 4-7 for multipliers
        timeSinceLastUpdate
    );
    // ...

    // Second slot has indexing error
    if (totalTokens > 4) {
        tokenIndex -= 4;  // @audit Resets to 0, breaking multiplier indices
        int256[] memory secondFourWeights = quantAMMUnpack32(_normalizedSecondFourWeights);
        normalizedWeights[4] = calculateBlockNormalisedWeight(
            secondFourWeights[0],
            secondFourWeights[tokenIndex],  // @audit Uses wrong indices 0-3 for multipliers ->
            ↪   essentially weight and multiplier are same
            timeSinceLastUpdate
        );
        // ...
    }
}
```

When processing the second slot of tokens (indices 4-7):

- First tokenIndex is set to 4 if total tokens > 4
- tokenIndex is decremented by 4, resetting to 0
- For all subsequent weight calculations, the weight & multiplier are effectively the same

Impact:

- Incorrect weight calculations for tokens 4-7 in pools with more than 4 tokens
- Significant deviation from intended pool asset allocation ratios

Recommended Mitigation: When there are more than 4 tokens in the pool, do not decrement the tokenIndex by 4.

QuantAMM: Fixed in `60815f2`

Cyfrin: Verified

7.1.3 Incorrect handling of negative multipliers in `QuantAMMWeightedPool` leads to underflow in weight calculation

Description: In `QuantAMMWeightedPool::calculateBlockNormalisedWeight`:

```
if (multiplier > 0) {
    return uint256(weight) + FixedPoint.mulDown(uint256(multiplierScaled18), timeSinceLastUpdate);
} else {
    // @audit silent overflow, uint256(multiplierScaled18) of a negative value will result in a very
    ↪  large value
    return uint256(weight) - FixedPoint.mulUp(uint256(multiplierScaled18), timeSinceLastUpdate);
}
```

Impact: All swaps with negative multiplier which aren't on the same block that the update happened (`timeSinceLastUpdate != 0`) will fail due to underflow.

Proof of Concept: Add the following test to `pkg/pool-quantamm/test/foundry/QuantAMMWeightedPool8Token.t.sol`:

```
function testGetNormalizeNegativeMultiplierOnSwapOutGivenInInitialToken0Token1() public {
    testParam memory firstWeight = testParam(0, 0.1e18, 0.001e18);
    testParam memory secondWeight = testParam(1, 0.15e18, -0.001e18);
    _onSwapOutGivenInInternal(firstWeight, secondWeight, 2, 1.332223208952048000e18);
}
```

Recommended Mitigation: Consider multiplying by `-1` first:

```
- return uint256(weight) - FixedPoint.mulUp(uint256(multiplierScaled18), timeSinceLastUpdate);
+ return uint256(weight) - FixedPoint.mulUp(uint256(-multiplierScaled18), timeSinceLastUpdate);
```

QuantAMM: Fixed in commit `31636cf`

Cyfrin: Verified.

7.1.4 Mismatch in multiplier position causes incorrect weight calculations in `QuantAMMWeightedPool`

Description: The primary feature of QuantAMM weighted pools is to periodically update weights based on various trading rules set during pool creation. These weights are updated by a singleton contract, `UpdateWeightRunner`, which provides new weights and multipliers. The multipliers help to adjust the weights between blocks until the next update.

To optimize gas usage compared to Balancer weighted pools, QuantAMM packs weights and multipliers into just two storage slots.

When an update occurs, `UpdateWeightRunner` sends the new weights to `QuantAMMWeightedPool::setWeights`. The weights and multipliers are sent in a different format than the pool expects though, as described in the NatSpec comment for `QuantAMMWeightedPool::_splitWeightAndMultipliers`:

```
/// @dev Update weight runner gives all weights in a single array shaped like
↪   [w1,w2,w3,w4,w5,w6,w7,w8,m1,m2,m3,m4,m5,m6,m7,m8], we need it to be
↪   [w1,w2,w3,w4,m1,m2,m3,m4,w5,w6,w7,w8,m5,m6,m7,m8]
```

The data is then split into two storage slots as follows:

- **First slot:** `[w1, w2, w3, w4, m1, m2, m3, m4]`
- **Second slot:** `[w5, w6, w7, w8, m5, m6, m7, m8]`

For a pool with just two tokens, `UpdateWeightRunner` sends the weights and multipliers as `[w1, w2, m1, m2]`, which are stored in this same order (with zeros padded at the end). However, for a pool with more tokens (e.g., 6 tokens), more complex logic is required to split the data into two slots. This is handled by `QuantAMMWeightedPool::_splitWeightAndMultipliers`.

When `UpdateWeightRunner` sends the weights and multipliers for 6 tokens as `[w1, w2, w3, w4, w5, w6, m1, m2, m3, m4, m5, m6]`, the first slot is populated like this:

```
uint256 tokenLength = weights.length / 2;
splitWeights = new int256[][](2);
splitWeights[0] = new int256[](8);
for (uint i; i < 4; ) {
    splitWeights[0][i] = weights[i];
    splitWeights[0][i + 4] = weights[i + tokenLength];


    unchecked {
        i++;
    }
}
```

This results in the first slot being `[w1, w2, w3, w4, m1, m2, m3, m4]`.

The second slot is handled with the following logic:

```
splitWeights[1] = new int256[](8);
uint256 moreThan4Tokens = tokenLength - 4;
for (uint i = 0; i < moreThan4Tokens; ) {
    uint256 i4 = i + 4;
    splitWeights[1][i] = weights[i4];
    splitWeights[1][i4] = weights[i4 + tokenLength];

    unchecked {
        i++;
    }
}
```

For 6 tokens, this results in the second slot being `[w5, w6, 0, 0, m5, m6, 0, 0]`.

The issue arises when these values are read in `QuantAMMWeightedPool::_getNormalizedWeight`:

```
uint256 tokenIndexInPacked = totalTokens;

if (tokenIndex > 4) {
    //get the index in the second storage int
    index = tokenIndex - 4;
    targetWrappedToken = _normalizedSecondFourWeights;
    tokenIndexInPacked -= 4;
} else {
```

Here, `tokenIndexInPacked` becomes `6 - 4 = 2`, which is then used in `QuantAMMWeightedPool::_calculate-CurrentBlockWeight`:

```
int256 blockMultiplier = tokenWeights[tokenIndex + (tokensInTokenWeights)];
```

For token index 5, or index 1 in second slot, this attempts to access the multiplier at position `1 + 2 = 3`, whereas the multiplier is actually in the fifth position. This mismatch causes incorrect multipliers or no multipliers to be used.

Impact: The incorrect reading of multipliers causes the wrong weight to be applied during token swaps. As a result, swap amounts may be incorrect, leading to unintended and potentially harmful outcomes for users.

Proof of Concept: Add this test to `pkg/pool-quantamm/test/foundry/QuantAMMWeightedPool8Token.t.sol`:

```
function testGetNormalizedWeightsInitial6Tokens() public {
    int256 weight = int256(1e18) / 6 + 1;
    PoolRoleAccounts memory roleAccounts;
    IERC20[] memory tokens = [
        address(dai),
        address(usdc),
        address(weth),
        address(wsteth),
        address(veBAL),
        address(waDAI)
    ].toMemoryArray().asIERC20();
    MockMomentumRule momentumRule = new MockMomentumRule(owner);

    int256[] memory initialWeights = new int256[](6);
    initialWeights[0] = weight;
    initialWeights[1] = weight;
    initialWeights[2] = weight - 1;
    initialWeights[3] = weight - 1;
    initialWeights[4] = weight;
    initialWeights[5] = weight;

    uint256[] memory initialWeightsUint = new uint256[](6);
    initialWeightsUint[0] = uint256(weight);
    initialWeightsUint[1] = uint256(weight);
    initialWeightsUint[2] = uint256(weight - 1);
    initialWeightsUint[3] = uint256(weight - 1);
    initialWeightsUint[4] = uint256(weight);
    initialWeightsUint[5] = uint256(weight);

    uint64[] memory lambdas = new uint64[](1);
    lambdas[0] = 0.2e18;

    int256[][] memory parameters = new int256[][](1);
    parameters[0] = new int256[](1);
    parameters[0][0] = 0.2e18;
```

```
address[][] memory oracles = new address[][](1);
oracles[0] = new address[](1);
oracles[0][0] = address(chainlinkOracle);

QuantAMMWeightedPoolFactory.NewPoolParams memory params = QuantAMMWeightedPoolFactory.NewPoolParams(
    "Pool With Donation",
    "PwD",
    vault.buildTokenConfig(tokens),
    initialWeightsUint,
    roleAccounts,
    MAX_SWAP_FEE_PERCENTAGE,
    address(0),
    true,
    false, // Do not disable unbalanced add/remove liquidity
    ZERO_BYTES32,
    initialWeights,
    IQuantAMMWeightedPool.PoolSettings(
        new IERC20[](6),
        IUpdateRule(momentumRule),
        oracles,
        60,
        lambdas,
        0.01e18,
        0.01e18,
        0.01e18,
        parameters,
        address(0)
    ),
    initialWeights,
    initialWeights,
    3600,
    0,
    new string[][](0)
);

address quantAMMWeightedPool = quantAMMWeightedPoolFactory.create(params);

uint256[] memory weights = QuantAMMWeightedPool(quantAMMWeightedPool).getNormalizedWeights();

int256[] memory newWeights = new int256[](12);
newWeights[0] = weight;
newWeights[1] = weight;
newWeights[2] = weight - 1;
newWeights[3] = weight - 1;
newWeights[4] = weight;
newWeights[5] = weight;
newWeights[6] = 0.001e18;
newWeights[7] = 0.001e18;
newWeights[8] = 0.001e18;
newWeights[9] = 0.001e18;
newWeights[10] = 0.001e18;
newWeights[11] = 0.001e18;

vm.prank(address(updateWeightRunner));
QuantAMMWeightedPool(quantAMMWeightedPool).setWeights(
    newWeights,
    quantAMMWeightedPool,
    uint40(block.timestamp + 5)
);

uint256[] memory balances = new uint256[](6);
```

```
    balances[0] = 1000e18;
    balances[1] = 1000e18;
    balances[2] = 1000e18;
    balances[3] = 1000e18;
    balances[4] = 1000e18;
    balances[5] = 1000e18;

    PoolSwapParams memory swapParams = PoolSwapParams({
        kind: SwapKind.EXACT_IN,
        amountGivenScaled18: 1e18,
        balancesScaled18: balances,
        indexIn: 0,
        indexOut: 1,
        router: address(router),
        userData: abi.encode(0)
    });

    vm.warp(block.timestamp + 5);

    vm.prank(address(vault));
    uint256 swap1Balance = QuantAMMWeightedPool(quantAMMWeightedPool).onSwap(swapParams);

    swapParams.indexOut = 5;

    vm.prank(address(vault));
    uint256 swap5Balance = QuantAMMWeightedPool(quantAMMWeightedPool).onSwap(swapParams);

    assertEq(swap1Balance, swap5Balance);
}
```

Recommended Mitigation: Consider storing the second slot the same way as the first one:

```
- splitWeights[1][i4] = weights[i4 + tokenLength];
+ splitWeights[1][i + moreThan4Tokens] = weights[i4 + tokenLength];
```

This will also require changes to `getNormalizedWeights` as it assumes a distance of 4 in the second weight slot.

[Project]: Fixed in commit 31636cf and 8a6b3b2

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Incorrect range validation in `quantAMMPackEight32` allows invalid values to be packed

Description: The `ScalarQuantAMMBaseStorage::quantAMMPackEight32` function contains incorrect validation logic that fails to properly check minimum bounds for six out of eight input parameters. The require statement incorrectly reuses `_firstInt` for minimum bound checks instead of checking each respective input parameter.

```
require(
    _firstInt <= MAX32 &&
        _firstInt >= MIN32 &&
        _secondInt <= MAX32 &&
        _secondInt >= MIN32 &&
        _thirdInt <= MAX32 &&
        _firstInt >= MIN32 &&  // @audit should be _thirdInt
        _fourthInt <= MAX32 &&
        _firstInt >= MIN32 &&  // @audit should be _fourthInt
        _fifthInt <= MAX32 &&
        _firstInt >= MIN32 &&  // @audit should be _fifthInt
        _sixthInt <= MAX32 &&
        _firstInt >= MIN32 &&  // @audit should be _sixthInt
        _seventhInt <= MAX32 &&
        _firstInt >= MIN32 &&  // @audit should be _seventhInt
        _eighthInt <= MAX32 &&
        _firstInt >= MIN32,    // @audit should be _eighthInt
    "Overflow"
);
```

This function is called by `quantAMMPack32Array` which inturn is used in the `QuantAMMWeightedPool::setWeights` function to pack weight values.

Impact: While the `setWeights` function is protected by access controls (only callable by `updateWeightRunner`), allowing invalid values to be packed could lead to unexpected behavior in weight calculations.

Proof of Concept: TBD

Recommended Mitigation: Consider updating the require statement to properly validate minimum bounds for all input parameters:

QuantAmm: We use the pack in setInitialWeights which is used in the initialise function. Given the absolute guard rail logic etc I do not think it is mathematically possible for this to be triggered during running of the pool I think they may only fail the deployment of the pool if a pool is initialised with bad weights. Fixed in commit `408ba20`.

Cyfrin: Verified.

7.2.2 Lack of permission check allows unauthorized updates to `lastPoolUpdateRun`

Description: To apply the weight changes calculated by the rules, any user can call `UpdateWeightRunner::performUpdate`.

Each pool has a set of permissions, configured during deployment in `QuantAMMWeightedPool.poolRegistry`. Each bit in this registry represents a specific permission. To execute `performUpdate`, the `MASK_POOL_PERFORM_UPDATE` bit must be set.

To prevent `performUpdate` from being called too frequently, which could impact the smoothing of the algorithms used, each pool has a configured `updateInterval` that dictates the minimum time between updates.

An admin can override this restriction by calling `UpdateWeightRunner::InitialisePoolLastRunTime`.

The ability to call `InitialisePoolLastRunTime` is restricted based on permissions. The pool requires either the `MASK_POOL_DAO_WEIGHT_UPDATES` or the `MASK_POOL_OWNER_UPDATES` bit to be set, and it verifies the identity of the caller accordingly:

```
// Current breakglass settings allow DAO or pool creator to trigger updates. This is subject to review.
if (poolRegistryEntry & MASK_POOL_DAO_WEIGHT_UPDATES > 0) {
    address daoRunner = QuantAMMBaseAdministration(quantammAdmin).daoRunner();
    require(msg.sender == daoRunner, "ONLYDAO");
} else if (poolRegistryEntry & MASK_POOL_OWNER_UPDATES > 0) {
    require(msg.sender == poolRuleSettings[_poolAddress].poolManager, "ONLYMANAGER");
}
poolRuleSettings[_poolAddress].timingSettings.lastPoolUpdateRun = _time;
```

If none of these permissions are set, anyone can call `InitialisePoolLastRunTime`.

Impact: A malicious user can:

1. **Prevent legitimate updates** by modifying `lastPoolUpdateRun` to an inappropriate time.
2. **Bypass** `updateInterval` **restrictions** by setting `lastPoolUpdateRun` to a time far in the past, allowing updates to be performed sooner than intended.

Proof of Concept: Add the following test to `pkg/pool-quantamm/test/foundry/UpdateWeightRunner.t.sol`:

```
function testAnyoneCanUpdatePoolLastRunTime() public {
    // pool lacks MASK_POOL_DAO_WEIGHT_UPDATES and MASK_POOL_OWNER_UPDATES but has
    ↪  MASK_POOL_PERFORM_UPDATE
    mockPool.setPoolRegistry(1);

    vm.startPrank(owner);
    // Deploy oracles with fixed values and delay
    chainlinkOracle1 = deployOracle(1001, 0);
    chainlinkOracle2 = deployOracle(1002, 0);

    updateWeightRunner.addOracle(chainlinkOracle1);
    updateWeightRunner.addOracle(chainlinkOracle2);
    vm.stopPrank();

    address[][] memory oracles = new address[][](2);
    oracles[0] = new address[](1);
    oracles[0][0] = address(chainlinkOracle1);
    oracles[1] = new address[](1);
    oracles[1][0] = address(chainlinkOracle2);

    uint64[] memory lambda = new uint64[](1);
    lambda[0] = 0.0000000005e18;
    vm.prank(address(mockPool));
    updateWeightRunner.setRuleForPool(
        IQuantAMMWeightedPool.PoolSettings({
            assets: new IERC20[](0),
            rule: IUpdateRule(mockRule),
            oracles: oracles,
            updateInterval: 10,
            lambda: lambda,
            epsilonMax: 0.2e18,
            absoluteWeightGuardRail: 0.2e18,
            maxTradeSizeRatio: 0.2e18,
            ruleParameters: new int256[][](0),
            poolManager: addr2
        })
    );

    address anyone = makeAddr("anyone");

    vm.prank(anyone);
    updateWeightRunner.InitialisePoolLastRunTime(address(mockPool), uint40(block.timestamp));

    vm.expectRevert("Update not allowed");
    updateWeightRunner.performUpdate(address(mockPool));
}
```

Recommended Mitigation: Add an `else` clause to revert when no appropriate permissions are set. This ensures only authorized users can modify `lastPoolUpdateRun`:

```
}
+ else {
+     revert("No permission to set lastPoolUpdateRun");
+ }
poolRuleSettings[_poolAddress].timingSettings.lastPoolUpdateRun = _time;
```

QuantAMM: Fixed in `a69c06f` and `84f506f`

Cyfrin: Verified. Calls to `InitialisePoolLastRunTime` now revert for an unauthorized user.

7.2.3 Faulty permission check in `UpdateWeightRunner::getData` allows unauthorized access

Description: Certain actions on a pool are protected by permissions. One such permission is `POOL_GET_DATA`, which governs access to the `UpdateWeightRunner::getData` function:

```
bool internalCall = msg.sender != address(this);
require(internalCall || approvedPoolActions[_pool] & MASK_POOL_GET_DATA > 0, "Not allowed to get data");
```

The issue lies in how the `internalCall` condition is enforced. The variable `internalCall` is intended to check if the call is made internally by the contract. However, this logic behaves incorrectly:

- `internalCall` will always be `true` because `UpdateWeightRunner` does not call `getData` in a way that changes the call context (i.e., `address(this).getData()`). As a result, `msg.sender` is never equal to `address(this)`.
- Consequently, the statement `msg.sender != address(this)` will always evaluate to `true`, allowing the `require` condition to pass regardless of the pool's permissions.

As a result, the `POOL_GET_DATA` permission is effectively bypassed.

Impact: The `UpdateWeightRunner::getData` function can be called on pools that lack the `POOL_GET_DATA` permission.

Proof of Concept: Add the following test to `pkg/pool-quantamm/test/foundry/UpdateWeightRunner.t.sol`:

```
function testUpdateWeightRunnerGetDataPermissionCanByBypassed() public {
    vm.prank(owner);
    updateWeightRunner.setApprovedActionsForPool(address(mockPool), 1);

    assertEq(updateWeightRunner.getPoolApprovedActions(address(mockPool)) & 2 /* MASK_POOL_GET_DATA */,
    ↪  0);

    // this should revert as the pool is not allowed to get data
    updateWeightRunner.getData(address(mockPool));
}
```

Recommended Mitigation: Consider removing the `internalCall` boolean entirely. This change will ensure that only calls with the correct `POOL_GET_DATA` permission are allowed. Additionally, this modification allows `getData` to be a `view` function, as the event emission at the end becomes unnecessary.

Here is the recommended code change:

```diff
- function getData(address _pool) public returns (int256[] memory outputData) {
+ function getData(address _pool) public view returns (int256[] memory outputData) {
-     bool internalCall = msg.sender != address(this);
-     require(internalCall || approvedPoolActions[_pool] & MASK_POOL_GET_DATA > 0, "Not allowed to get
↪   data");
+     require(approvedPoolActions[_pool] & MASK_POOL_GET_DATA > 0, "Not allowed to get data");
// ...
-     if(!internalCall){
-         emit GetData(msg.sender, _pool);
-     }
```

QuantAMM: Fixed in a1a333d

Cyfrin: Verified. Code refactored, `internalCall` now passed as a parameter to a new `UpdateWeightRunner::_getData`. The original `UpdateWeightRunner::getData` requires permission but the internal call during update does not.

7.2.4 Stale Oracle prices accepted when no backup oracles available

Description: The `UpdateWeightRunner::getData` function performs staleness checks on oracle prices to ensure price data is fresh. However, when the optimized (primary) oracle returns stale data and there are no backup oracles configured, the function silently accepts and returns the stale price data instead of reverting.

This occurs because the logic checks for staleness on the primary oracle, but if stale, it moves to a backup oracle check section. When no backup oracles exist, this section is skipped due to array length checks, and the function proceeds to use the stale data from the primary oracle.

```solidity
// getData function
function getData(address _pool) public returns (int256[] memory outputData) {
         // .... code
        outputData = new int256[](oracleLength);
        uint oracleStalenessThreshold = IQuantAMMWeightedPool(_pool).getOracleStalenessThreshold();

        for (uint i; i < oracleLength; ) {
            // Asset is base asset
            OracleData memory oracleResult;
            oracleResult = _getOracleData(OracleWrapper(optimisedOracles[i]));
            if (oracleResult.timestamp > block.timestamp - oracleStalenessThreshold) {
                outputData[i] = oracleResult.data; //@audit skips this correctly when data is stale
            } else {
                unchecked {
                    numAssetOracles = poolBackupOracles[_pool][i].length;
                }

                for (uint j = 1 /*0 already done via optimised poolOracles*/; j < numAssetOracles; ) {
                ↪   //@audit for loop is skipped when no backup oracle
                    oracleResult = _getOracleData(
                        // poolBackupOracles[_pool][asset][oracle]
                        OracleWrapper(poolBackupOracles[_pool][i][j])
                    );
                    if (oracleResult.timestamp > block.timestamp - oracleStalenessThreshold) {
                        // Oracle has fresh values
                        break;
                    } else if (j == numAssetOracles - 1) {
                        // All oracle results for this data point are stale. Should rarely happen in
                        ↪  practice with proper backup oracles.

                        revert("No fresh oracle values available");
                    }
                    unchecked {
                        ++j;
                    }
                }
                outputData[i] = oracleResult.data; //@audit BUG - here it accepts the same stale data
                ↪  that was rejected before
            }

            unchecked {
                ++i;
            }
        }

        if(!internalCall){
            emit GetData(msg.sender, _pool);
        }
    }
```

Impact: Stale oracle prices could be used to calculate pool weight updates, leading to incorrect weight adjustments based on outdated market information.

Recommended Mitigation: Consider reverting in the `else` block when `numAssetOracles == 1`

```
 else {
    unchecked {
        numAssetOracles = poolBackupOracles[_pool][i].length;
    }

    if (numAssetOracles == 1) {  // @audit revert when No backups available (1 accounts for primary
    ↪   oracle)
        revert("No fresh oracle values available");
    }

    for (uint j = 1; j < numAssetOracles; ) {
        // ... rest of the code
    }
}
```

QuantAMM: A stale update is better than none because of the way the estimators encapsulate a weighted price and a certain level of smoothing. The longer away a price is, the less important. So triggering an update may mean its with a current stale price however it keeps all the historical prices baked into the exponential smoothing on track in terms of their importance.

Cyfrin: Acknowledged.

7.2.5 Missing admin functions prevent execution of key `UpdateWeightRunner` actions

Description: Several functions in `UpdateWeightRunner` require the `quantammAdmin` role, but these functions are missing from `QuantAMMBaseAdministration`. The affected functions are:

- `UpdateWeightRunner::addOracle`
- `UpdateWeightRunner::removeOracle`
- `UpdateWeightRunner::setApprovedActionsForPool`
- `UpdateWeightRunner::setETHUSDOracle`

Since these functions rely on `quantammAdmin`, they cannot be executed unless the corresponding calls are included in `QuantAMMBaseAdministration`.

Impact: The `QuantAMMBaseAdministration` contract cannot execute the above functions. As a result, administrative actions such as adding or removing oracles, setting approved actions for pools, and updating the ETH/USD oracle cannot be performed, limiting the functionality and flexibility of the system.

Recommended Mitigation: Consider one of the following approaches:

1. Add the Missing Functions to `QuantAMMBaseAdministration`
2. Use OpenZeppelin's `TimeLock` Contract Directly: If `QuantAMMBaseAdministration` is redundant, replace it with OpenZeppelin's `TimeLock` contract to manage administrative functions securely and effectively.

QuantAMM: Fixed in a299ce7

Cyfrin: Verified. `QuantAMMBaseAdministration` is removed.

7.2.6 `QuantAMMBaseAdministration::onlyExecutor` modifier does not enforce a time lock on actions

Description: In `QuantAMMBaseAdministration`, certain functions are restricted using the `onlyExecutor` modifier:

```
// Modifier to check for EXECUTOR_ROLE using `timelock`
modifier onlyExecutor() {
    require(timelock.hasRole(timelock.EXECUTOR_ROLE(), msg.sender), "Not an executor");
    _;
}
```

The issue with this modifier is that it does not enforce a timelock on the actions. Instead, it only verifies that the caller has the `EXECUTOR_ROLE` on the timelock contract. As seen in the OpenZeppelin `TimelockController` code, this check allows anyone with the `EXECUTOR_ROLE` to bypass the timelock and execute commands directly on `QuantAMMBaseAdministration`.

Impact: Anyone with permission to execute actions on the timelock contract can bypass the timelock and execute the same commands directly on the `QuantAMMBaseAdministration` contract. This undermines the purpose of the timelock, which is to provide a delay for critical actions.

Recommended Mitigation:

1. **Modify the `onlyExecutor` Modifier to Enforce Timelock:** Change the `onlyExecutor` modifier to require that the caller is the timelock contract itself:

   ```
   modifier onlyTimelock() {
       require(address(timelock) == msg.sender, "Only timelock");
       _;
   }
   ```

2. **Make `timelock` Public:** To allow querying the timelock's address for proposing actions, make the `timelock` variable `public`:

   ```
   - TimelockController private timelock;
   + TimelockController public timelock;
   ```

3. **Alternative Solution:** Instead of using `QuantAMMBaseAdministration`, consider using the OpenZeppelin `TimelockController` contract directly as `quantammAdmin`. This approach streamlines the process and ensures timelock enforcement without redundant checks.

QuantAMM: Fixed in a299ce7

Cyfrin: Verified. `QuantAMMBaseAdministration` is removed.

7.2.7 Unauthorized role grants prevent `QuantAMMBaseAdministration` deployment

Description: `QuantAMMBaseAdministration` is a contract that manages certain break-glass features and facilitates calls to other QuantAMM contracts.

However, this contract fails to deploy due to an issue in the `QuantAMMBaseAdministration` constructor, where roles are granted to proposers and executors:

```
for (uint256 i = 0; i < proposers.length; i++) {
    timelock.grantRole(timelock.PROPOSER_ROLE(), proposers[i]);
}

for (uint256 i = 0; i < executors.length; i++) {
    timelock.grantRole(timelock.EXECUTOR_ROLE(), executors[i]);
}
```

The issue arises because the `QuantAMMBaseAdministration` contract needs to have the `DEFAULT_ADMIN_ROLE` in the `timelock` contract for these `grantRole` calls to succeed. Since it does not possess this role, the calls will revert with an `AccessControlUnauthorizedAccount` error.

Impact: The deployment of `QuantAMMBaseAdministration` will fail due to unauthorized role assignment, preventing the contract from being deployed successfully.

Proof of Concept: Add the following test file to the `pkg/pool-quantamm/test/foundry/` folder:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "forge-std/Test.sol";
import "../../contracts/QuantAMMBaseAdministration.sol";

contract QuantAMMBaseAdministrationTest is Test {

    address daoRunner = makeAddr("daoRunner");

    address proposer = makeAddr("proposer");
    address executor = makeAddr("executor");

    QuantAMMBaseAdministration admin;

    function testSetupQuantAMMBaseAdministration() public {
        address[] memory proposers = new address[](1);
        proposers[0] = proposer;

        address[] memory executors = new address[](1);
        executors[0] = executor;

        vm.expectRevert(); // AccessControlUnauthorizedAccount(address(admin),DEFAULT_ADMIN_ROLE)
        admin = new QuantAMMBaseAdministration(
            daoRunner,
            1 hours,
            proposers,
            executors
        );
    }
}
```

Recommended Mitigation: Remove the role grants in the constructor, as these assignments are already handled in the `TimeLockController` constructor:

QuantAMM: Fixed in `a299ce7`

Cyfrin: Verified. `QuantAMMBaseAdministration` is removed.

7.3 Low Risk

7.3.1 `UpdateWeightRunner::performUpdate` reverts on underflow when new multiplier is zero

Description: In `UpdateWeightRunner::_calculateMultiplerAndSetWeights`, the new multiplier is determined by calculating the change in weight over time.

To protect against unexpected values, a "last valid timestamp" for the multiplier is calculated in the section here.

When there is no change in weight, the multiplier becomes zero. This scenario is handled by a special case in the following code:

```
int256 currentLastInterpolationPossible = type(int256).max;

for (uint i; i < local.currentWeights.length; ) {
    // ...

    if (blockMultiplier > int256(0)) {
        weightBetweenTargetAndMax = upperGuardRail - local.currentWeights[i];
        // ...
        blockTimeUntilGuardRailHit = weightBetweenTargetAndMax / blockMultiplier;
    } else if (blockMultiplier == int256(0)) {
        blockTimeUntilGuardRailHit = type(int256).max; // @audit if blockMultiplier is 0 this is max
    } else {
        weightBetweenTargetAndMax = local.currentWeights[i] - local.absoluteWeightGuardRail18;
        // ...
        blockTimeUntilGuardRailHit = weightBetweenTargetAndMax / int256(uint256(-1 * blockMultiplier));
    }
    // ...
}
// ...

//next expected update + time beyond that
currentLastInterpolationPossible += int40(uint40(block.timestamp));
```

If the weights remain unchanged and the multiplier is `0`, `currentLastInterpolationPossible` is set to `type(int256).max`. Consequently, the statement `currentLastInterpolationPossible += int40(uint40(block.timestamp))` will revert due to integer overflow.

The same overflow can also potentially happen if the multiplier is very small in the other branches as `weightBetweenTargetAndMax / blockMultiplier` could blow up.

Impact: Whenever the new multiplier is `0` (or extremely large), the call to `UpdateWeightRunner::performUpdate` will revert. This prevents updates from being executed when weights remain constant or under certain edge cases.

Proof of Concept: Add the following test to `pkg/pool-quantamm/test/foundry/UpdateWeightRunner.t.sol`:

```
function testUpdatesFailsWhenMultiplierIsZero() public {
    vm.startPrank(owner);
    updateWeightRunner.setApprovedActionsForPool(address(mockPool), 3);
    vm.stopPrank();
    int256[] memory initialWeights = new int256[](4);
    initialWeights[0] = 0;
    initialWeights[1] = 0;
    initialWeights[2] = 0;
    initialWeights[3] = 0;

    // Set initial weights
    mockPool.setInitialWeights(initialWeights);

    int216 fixedValue = 1000;
    chainlinkOracle = deployOracle(fixedValue, 3601);
    vm.startPrank(owner);
    updateWeightRunner.addOracle(OracleWrapper(chainlinkOracle));
    vm.stopPrank();

    vm.startPrank(address(mockPool));

    address[][] memory oracles = new address[][](1);
    oracles[0] = new address[](1);
    oracles[0][0] = address(chainlinkOracle);

    uint64[] memory lambda = new uint64[](1);
    lambda[0] = 0.0000000005e18;
    updateWeightRunner.setRuleForPool(
        IQuantAMMWeightedPool.PoolSettings({
            assets: new IERC20[](0),
            rule: IUpdateRule(mockRule),
            oracles: oracles,
            updateInterval: 1,
            lambda: lambda,
            epsilonMax: 0.2e18,
            absoluteWeightGuardRail: 0.2e18,
            maxTradeSizeRatio: 0.2e18,
            ruleParameters: new int256[][](0),
            poolManager: addr2
        })
    );
    vm.stopPrank();

    vm.startPrank(addr2);
    updateWeightRunner.InitialisePoolLastRunTime(address(mockPool), 1);
    vm.stopPrank();

    vm.warp(block.timestamp + 10000000);
    mockRule.CalculateNewWeights(
        initialWeights,
        new int256[](0),
        address(mockPool),
        new int256[][](0),
        new uint64[](0),
        0.2e18,
        0.2e18
    );
    vm.expectRevert();
    updateWeightRunner.performUpdate(address(mockPool));
}
```

Recommended Mitigation: Consider guarding against overflow:

```
  //next expected update + time beyond that
+ if(currentLastInterpolationPossible < int256(type(int40).max)) {
      currentLastInterpolationPossible += int40(uint40(block.timestamp));
+ }
```

QuantAMM: Fixed in 9b7a998

Cyfrin: Verified. Overflow now prevented.

7.3.2 Missing weight sum validation in `setWeightsManually` function

Description: The `UpdateWeightRunner::setWeightsManually` allows privileged users (DAO, pool manager, or admin based on pool registry) to manually set weights of the pool without validating if the weights sum to 1. This check is correctly enforced the first time when weights are initialized in `_setInitialWeights`:

```
// In QuantAMMWeightedPool.sol - _setInitialWeights
// Ensure that the normalized weights sum to ONE
if (uint256(normalizedSum) != FixedPoint.ONE) {
    revert NormalizedWeightInvariant();
}
```

A similar validation is missing in setWeightsManually in UpdateWeightRunner.sol, creating an inconsistency in how weight validations are handled when weights are entered manually.

Impact: While this function is access controlled, allowing privileged users to set weights that don't sum to 1 could cause:

- Incorrect pool valuation
- Unfair trading due to miscalculated swap amounts

It is noteworthy that although `setInitialWeights` also has privileged access, weight sum check exists in that function.

Recommended Mitigation: Consider adding a weight sum validation in `setWeightsManually`

QuantAMM: In practice and in theory the weights do not have to sum to one. Trading is done on the ratio not the absolute values of the weights. Everyone does it but if you set it to 50/50 instead of 0.5/0.5 nothing would change in what swaps were accepted by the mathematics of the pool. Our general practice with break glass measures is to include no validation as which glass needs breaking could be an unknown unknown. However, we reviewed the potential issues and actually given we use fixed point math libs that expect 1>x>0 (especially greater than 0) it does make sense to add the check so really maybe just a change in description of the issue. We will not check guard rails as that could be a valid break glass in some strange unknown.

Cyfrin: Acknowledged.

7.3.3 Extreme weight ratios combined with large balances can cause denial-of-service for unbalanced liquidity operations

Description: The weighted pool math can potentially overflow when making unbalanced liquidity adjustments to a pool tokens with very small weights and large balances.

The issue occurs in `WeightedMath::computeBalanceOutGivenInvariant`:

```
 newBalance = oldBalance * (invariantRatio ^ (1/weight))
```

When weights are small and invariant ratio is close to the maximum value (300%), even realistic balance changes can cause computations to overflow.

An example here taken from the fuzz test:

```
// Balance computation scenario:
balance = 7500e21  (7.5 million tokens)
weight = 0.01166 (1.166%) // Just above absoluteWeightGuardRail minimum of 1% proposed for Balancer
↪  pools
invariantRatio = 3.0 // maximum value

calculation = 7500e21 * (3.0 ^ (1/0.01166))
           = 7500e21 * (3.0 ^ 85.76)
```

```
            = OVERFLOW
```

`QuantAMMWeightedPool` allows weights down to `0.1 %`, as seen in `QuantAMMWeightedPool::_setRule`:

```
//applied both as a max (1 - x) and a min, so it cant be more than 0.49 or less than 0.01
//all pool logic assumes that absolute guard rail is already stored as an 18dp int256
require(
    int256(uint256(_poolSettings.absoluteWeightGuardRail)) <
        PRBMathSD59x18.fromInt(1) / int256(uint256((_initialWeights.length))) &&
        int256(uint256(_poolSettings.absoluteWeightGuardRail)) > 0.001e18, // @audit minimum guard rail
        ↪  allowed is 0.1 %
    "INV_ABSWGT"
); //Invalid absoluteWeightGuardRail value
```

Impact: Liquidity operations can revert for tokens with small weights and large balances. As a consequence, pool can become temporarily unusable as weights approach guard rail minimums. It is worth highlighting that users can always choose to proportionately add/remove liquidity to mitigate this scenario.

Recommended Mitigation: Risk of overflow exists when the minimum weight can go up to 1%. Even a modest increase of minimum weight to 3% effectively prevents this scenario. Consider changing the lowest enforced guard rail to be higher.

QuantAMM: Response from the balancer team and we agree:

> computeBalance is only used for unbalanced liquidity ops (add single token exact out, or remove single token exact in). Add exact out is only used in batch swaps. Remove exact in could be used if you want to exit single sided I guess. But in any case you can always exit proportional
>
> The example above uses an invariant ratio of 3, which is an add liquidity (>1). Compute balance for that ratio only happens for add single token exact out. You won't ever use that one unless you nest your pools (which is not supported anyways by default for [quantamm] weighted pools).

1% enforced as lowest guard rail in `4beffda`

Cyfrin: Verified.

7.4 Informational

7.4.1 Use of solidity `assert` instead of foundry asserts in test suite

Description: In QuantAMMWeightedPool8Token tests `assert` is used instead of foundry `assertEq`

Foundry assertEq provides better error information when failing such as the two values compared. Consider using `assertEq` instead of solidity `assert`

QuantAMM: Fixed in 414d4bc and ca1e441

Cyfrin: Verified

7.4.2 Consider adding a getter for `QuantAMMWeightedPool.poolDetails`

Description: QuantAMMWeightedPool.poolDetails is a nested array:

```
string[][] public poolDetails;
```

Nested array needs both indexes to access an element. To query for the whole array a custom getter would be needed. This could be useful for off-chain monitoring.

Reported by the protocol during audit.

QuantAMM: Fixed in ee1fbb8

Cyfrin: Verified.

7.4.3 `AntimomentumUpdateRule.parameterDescriptions` initialized too long

Description: AntimomentumUpdateRule.parameterDescriptions is initialized as length 3 but only two entries are used:

```
parameterDescriptions = new string[](3); // @audit should be 2
parameterDescriptions[0] = "Kappa: Kappa dictates the aggressiveness of response to a signal change
↪   TODO";
parameterDescriptions[1] = "Use raw price: 0 = use moving average, 1 = use raw price to be used as the
↪   denominator";
```

Consider initializing it as `new string[](2)`.

QuantAMM: Fixed in 91241ca

Cyfrin: Verified.

7.4.4 `TODO`s left in rule parameter descriptions

Description: The three rules AntimomentumUpdateRule, MinimumVarianceUpdateRule and MomentumUpdateRule have `TODO`s left in their parameter descriptions.

Consider finalizing the descriptions of these parameters.

QuantAMM: Fixed in e3e0d5e, 8c3a4f3, f56796e

Cyfrin: Verified.

7.4.5 Unnecessary `_initialWeights` sum check

Description: When first initializing a `QuantAMMWeightedPool` the pool creator provides the initial weights. These initial weights should sum up to 1 (`1e18`). The issue is however that this is verified both in QuantAMMWeighted-Pool::_setRule:

```
int256 sumWeights;
for (uint i; i < _initialWeights.length; ) {
    sumWeights += int256(_initialWeights[i]);
    unchecked {
        ++i;
    }
}

require(
    sumWeights == 1e18, // 1 for int32 sum
    "SWGT!=1"
); //Initial weights must sum to 1
```

And in `QuantAMMWeightedPool::_setInitialWeights`:

```
int256 normalizedSum;
int256[] memory _weightsAndBlockMultiplier = new int256[](_weights.length * 2);
for (uint i; i < _weights.length; ) {
    if (_weights[i] < int256(uint256(absoluteWeightGuardRail))) {
        revert MinWeight();
    }

    _weightsAndBlockMultiplier[i] = _weights[i];
    normalizedSum += _weights[i];
    //Initially register pool with no movement, first update will come and set block multiplier.
    _weightsAndBlockMultiplier[i + _weights.length] = int256(0);
    unchecked {
        ++i;
    }
}

// Ensure that the normalized weights sum to ONE
if (uint256(normalizedSum) != FixedPoint.ONE) {
    revert NormalizedWeightInvariant();
}
```

Consider removing the check in `_setRule` as that function is for verifying the rules and also the check in `_setInitialWeights` is more thorough, as it also checks they adhere to the `absoluteWeightGuardRail` requirement.

QuantAMM: Fixed in 6b891e8

Cyfrin: Verified.