Security Assessment Report

Uniswap V4 Hook – TWAMM v1.1

January 2025

Prepared for Zaha Studio

certora

Table of content

Project Summary **3**
- Project Scope 3
- Project Overview 3
  - Protocol Overview 3
- Findings Summary 4
- Severity Matrix 4

Detailed Findings **5**
- Critical Severity Issues 6
- C-01 Uniswap protocol fees and LP fees are not implemented correctly 6
- C-02 Squaring of sqrtPriceX96 causes overflow in unchecked block 7
- **Medium Severity Issues** **8**
- M-01 Wrong calculation of tick bit position 8
- M-02 Matching order exhaustion doesn't average pool price history 9
- **Low Severity Issues** **10**
- L-01 Sell rate imprecision for low-decimals tokens 10
- L-02 maxAdjustable0To1 and maxAdjustable1To0 might not be zero simultaneously 11
- **Informational Issues** **13**
- I-01 priceSq should be renamed to price 13
- I-02 Rebasing tokens are not handled correctly when claiming 13

Disclaimer **15**

About Certora **15**

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
v4-TWAMM-hook	akshatmittal/v4-twamm-hook	02950985541a210244d33a4d5c8e3efad8d4de2c	EVM

Project Overview

This document describes the findings of the manual review of **TWAMM hook**. The work was undertaken from Jan 19 to Jan 24, 2024

The following contract list is included in our scope:

- `src/*`

Protocol Overview

This protocol consists of a single Uniswap-v4 hook contract that establishes orders in arbitrary pools. The contract serves as a time-weighted-average market maker (TWAMM) for swapping orders on the Uniswap-v4 protocol. Traders submit swapping orders by depositing their funds to this hook which is coupled to a Uni-v4 pool, consisting of two tokens, but also specify the time frame in which the order is executed, effectively trading many very small token amounts over the specified trading period.

The hook makes sure to execute those orders and reward the original depositors with the output reserves the hook obtained.

certora

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	2	2	2
High	0	0	0
Medium	2	2	1
Low	2	2	2
Informational	2	2	1
Total	8	8	6

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
C-01	Uniswap protocol fees and LP fees are not implemented correctly	Critical	Fixed
C-02	Squaring of sqrtPriceX96 causes overflow in unchecked block	Critical	Fixed
M-01	Wrong calculation of tick bit position	Medium	Fixed
M-02	Matching order exhaustion doesn't average pool price history	Medium	Acknowledged
L-01	Sell rate imprecision for low-decimals tokens	Low	Fixed
L-02	maxAdjustable0To1 and maxAdjustable1To0 might not be zero simultaneously	Low	Fixed

Critical Severity Issues

C-01 Uniswap protocol fees and LP fees are not implemented correctly

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: TWAMM.sol	Status: Fixed	

Description:

The uniswap protocolFee is taken from the input, and then lp fees are taken from the remaining amount afterwards. This is explicitly documented on Slot0.sol:
“the protocolFee is taken from the input first, then the lpFee is taken from the remaining input”
TWAMM line 198 incorrectly sums both as the totalFee. The right way to express the swap fees can be referenced in the uniswap documentation in ProtocolFeeLibrary.sol::calculateSwapFee()
“protocolFee + lpFee(1_000_000 - protocolFee) / 1_000_000 (rounded up) “.

amountSelling is also calculated without using mulDiv, which could lead to significant precision loss and diversion from the Uniswap implementation as can be seen in SwapMath.computeSwapStep() function lines 64-65.

Recommendations:
We recommend calculating the swap total swap fee in the same way Uniswap does, so there is no mismatch between the TWAMM swap simulations and the actual swap execution.
In addition we recommend using mulDiv when calculating *amountSelling* to increase precision and mimic Uniswap precisely.

Client’s response: Protocol fee logic fixed at https://github.com/akshatmittal/v4-twamm-hook/commit/cde8b7cd80a93ddafcbc3298fac3cdfc4a7604d3

amountSelling precision fixed at:
https://github.com/akshatmittal/v4-twamm-hook/commit/97774ed231b93c5274bc83dc67e4f9eadd3c9333

Fix Review: Fixed appropriately.

C-02 Squaring of sqrtPriceX96 causes overflow in unchecked block

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: TWAMM.sol	Status: Fixed	

Description: the max sqrt price of a UniswapPool (TickMath.sol) is

```python
int160 internal constant MAX_SQRT_PRICE = 1461446703485210103287273052203988822378723970342;
```

which is roughly 2^160.
Squaring relatively large `sqrtPriceX96` values will cause overflow as the entire function call is within an "unchecked" block, so it would not revert, but actually give the wrong price and lead to loss of funds.

Recommendations:
Use `mulDiv` from Openzeppelin's math library and implement functionality to handle ERC20s with different decimals, for example USDC (6 decimals) or WETH (18 decimals). Revert if the token ERC20 interface does not implement `decimals` (it is optional).

Client's response: Fix commit:
https://github.com/akshatmittal/v4-twamm-hook/commit/cde8b7cd80a93ddafcbc3298fac3cdfc4a7604d3

Fix Review: Fixed appropriately.

Medium Severity Issues

M-01 Wrong calculation of tick bit position

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: PoolGetters.sol	Status: Fixed	

Description: In PoolGetters.sol, the function position(int24 tick) is supposed to return the bit and word positions of the tick, which mimics the original function from uniswap-v4.

```
function position(int24 tick) private pure returns (int16 wordPos, uint8 bitPos)
{
        unchecked {
            wordPos = int16(tick >> 8);
            bitPos = uint8(int8(tick % 256));
        }
    }
```

However, the intermediary conversion to int8 for the bit position could overflow because the bounds of the tick % 256 results are -2^8 and 2^8, but an int8 number is bounded by -2^7 and 2^7. Hence this conversion isn't safe.

Recommendations: Use a safe conversion with a larger uint (e.g. uint24). Otherwise, copy the original implementation of *position*() from the Uniswap-v4 codebase v4-core/src/libraries/TickBitmap.sol at main · Uniswap/v4-core.

Client's response: Fix commit:
Update protocolFee logic · akshatmittal/v4-twamm-hook@cde8b7c

Fix Review: Fixed appropriately.

certora

M-02 Matching order exhaustion doesn't average pool price history

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: TWAMM.sol	Status: Acknowledged	

Description:
_exhaustMatchedOrders() assumes the sqrtPriceX96 would remain static as the orders are matched. However, in reality, the price curve would shift in a specific direction, driving up the price of either token0 or token1 (unless they are exactly matched). This assumption leads to the market makers not receiving the expected returns.

Recommendations:
We recommend also simulating the swap for matched orders, but with the exclusion of the protocol and lp fees.

Client's response: The order of operations matters in a TWAMM. Depending on which side is executed first, it favors one of them.

So instead, this function calculates what part of the current order would be exhausted at the existing price and swaps it 1:1 leaving the remaining to be swapped in the pool.

The curve shift you're referring to would happen if the movement was asymmetrical, whereas this function removes assets from both sides symmetrically. You can think of it like swapping infinitely small amounts of assets in both directions at the same time.

Fix Review: After discussing with the client we came to the conclusion that our points of view in this matter differ. Client believes that making the swaps 1:1 in _exhaustMatchedOrders() is fairer than choosing the order of operations (by placing the biggest buyer first) and simulating the price curve shit that would occur in a direct swap with the pool. We agreed both methods have trade-offs and ultimately the client decided he would rather keep this method as is.

Low Severity Issues

L-01 Sell rate imprecision for low-decimals tokens		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: TWAMM.sol	Status: Fixed.	

Description: The sell rate of any order is determined by the ratio of the sell amount of that token and the time period in seconds. Due to rounding errors, the sell rate is effectively the nearest integer of the quotient *amount [tokens] / duration [seconds]*.
For low decimals tokens, the resulting rounding error could be significant if the duration is large. Taking USDC as an example, whose decimals is 6, the error is estimated as ~0.6 USDC per week. That is, if the user intended to sell X tokens per week, the resulting rate would be rounded down and the error of that rate would be 0.6USDC per week, at most.
That is rather a small amount, but still significant in terms of actual worth.

In general, for any duration D in seconds, any D-1 tokens, in the extreme rounding case, could be deducted from the total amount.

Recommendation: Consider using a higher precision calculation for the rate.

Client's response: Acknowledged and fixed at 031f3eb559d57cff8a093525ee8874283c046eed.

Fix Review: Fixed appropriately.

L-02 maxAdjustable0To1 and maxAdjustable1To0 might not be zero simultaneously

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: TWAMM.sol	Status: Fixed	

Description:

```
None
// If one is zero, the other must be zero too.
        if (maxAdjustable0To1 != 0) {
```

consider the calculation that precedes this line:
denote *sellRate0To1* and *sellRate1To0* as *sx* and *sy* respectively.
Then:

```
None
maxAdjustable0To1 = min(sx, floor(sy * 2^96 / price))
maxAdjustable1To0 = min(sy, floor(sx * price / 2^96))
```

The claim is of course true if either one of *sx* or *sy* is zero.
But if one is really small and the other one isn't, one max value can be zero but the other might not be.

Easiest (extreme) example:
sx = 1;
sy > 1;
P < 2^96

then maxAdjustable0To1 = 1
but maxAdjustable1To0 = 0 because price / 2^96 rounds down to zero

Recommendations:

Client's response:
Fix commit at:

https://github.com/akshatmittal/v4-twamm-hook/commit/97774ed231b93c5274bc83dc67e4f9eadd3c9333

Fix Review: Fixed appropriately.

Informational Issues

I-01 priceSq should be renamed to price

Severity: **Informational**	Impact:	Likelihood:
Files: TWAMM.sol	Status: Fixed	

Description: priceSq is actually the result of squaring the square root of the price. Consequently returning the price itself. Therefore priceSq should be renamed to price.

```
uint256 priceSq =
uint256(params.pool.sqrtPriceX96) ** 2 >> FixedPoint96.RESOLUTION;
```

Client's response: Acknowledged and fixed.

I-02 Rebasing tokens are not handled correctly when claiming

Severity: **Informational**	Impact:	Likelihood:
Files: TWAMM.sol	Status: Acknowledged	

Description: If the TWAMM hook is swapping inside a pool whose currency is a rebasing tokens, there could be an inconsistency between the number of tokens that the hook buys from the pool and the number of tokens a user is eligible for, upon claiming, as a result of the changing balance of the hook. If a user chooses to claim his tokens at some point, the value that he is owed is saved within the storage mapping *tokensOwed* which is updated upon a call to sync().

But, that value could change until the claiming operation, due to the rebasing nature of the token, which makes the balance dynamic.

The transferred amount to the user might be too small if the token had positive rebasing, because the value that is being claimed is the one stored in storage.

Recommendation:
Either introduce a rebasing tracker per the relevant tokens and fix the claimed amount accordingly, or disallow rebasing tokens all-together.

Client's response: Rebasing tokens are not supported.

Fix Review: As per client's response this finding has been downgraded to 'informational'. We advised the client to document the fact they do not support rebasing tokens.

certora

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.