

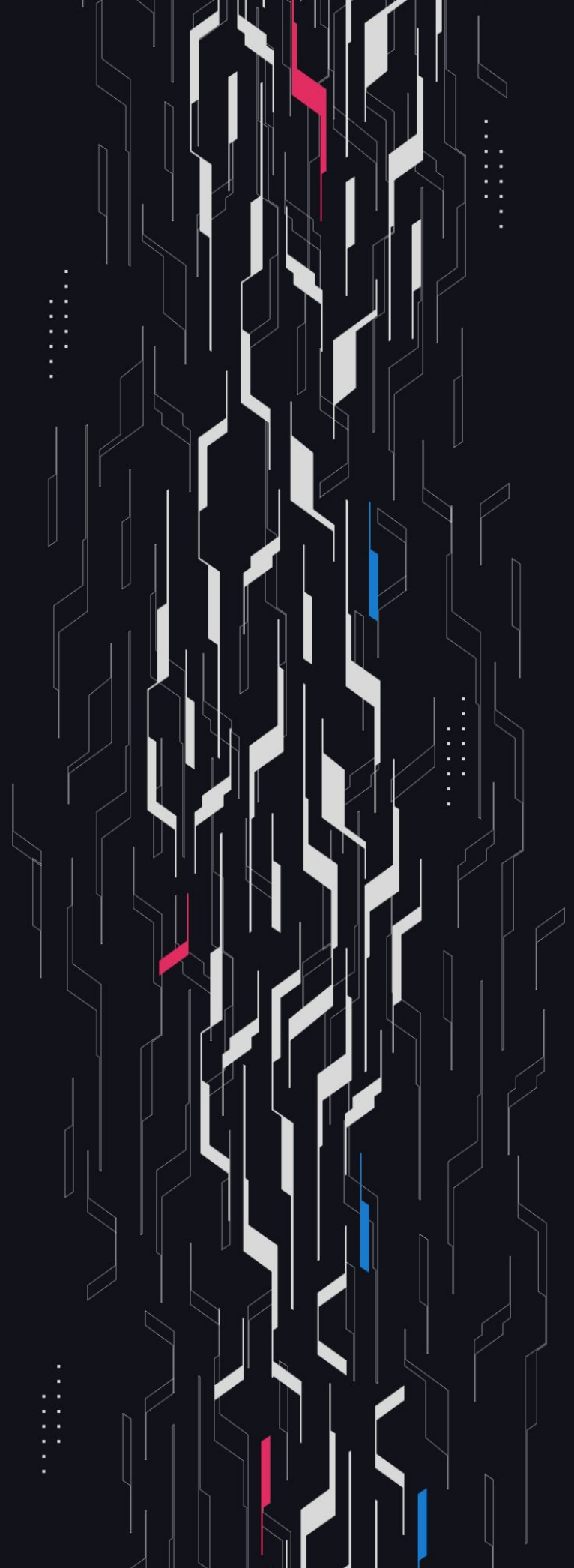
GA GUARDIAN

Trueo

Uni V4 Review

Security Assessment

November 3rd, 2025



Summary

Audit Firm Guardian

Prepared By Nicholas Chew, Ciphky, Parker Stevens, Michael Lett

Client Firm Trueo


Final Report Date November 3, 2025

Audit Summary

Trueo engaged Guardian to review the security of their Trueo Uni V4. From the 22nd of August to the 16th of October a team of 4 auditors reviewed the source code in scope. **Note:** Fixes to the Remediations V3 Findings have not been reviewed by Guardian in this report.

Confidence Ranking

Given the number of High and Critical issues detected as well as additional code changes made after the main review, and to address findings from the remediation review, Guardian recommends that an independent security review of the finalized deployed version is conducted.

 Blockchain network: **Base**

 Verify the authenticity of this report on Guardian's GitHub: <https://github.com/guardianaudits>

 PoC test suite: <https://github.com/GuardianOrg/truth-contracts-team1-1756065615985>,
<https://github.com/GuardianOrg/truth-contracts-team2-1756065620509>,
<https://github.com/GuardianOrg/truth-contracts-team3-1756065625022>,
<https://github.com/GuardianOrg/truth-contracts-fuzz-1756065631250>

Table of Contents

Project Information

Project Overview 5

Audit Scope & Methodology 6

Smart Contract Risk Assessment

Invariants Assessed 8

Findings & Resolutions 10

Addendum

Disclaimer 52

About Guardian 53

Project Overview

Project Summary

Project Name	Trueo
Language	Solidity
Codebase	https://github.com/truth-market/truth-contracts
Commit(s)	Initial commit: f8ad921851f264c3443b09067c0837b05d6dbb0c Final commit: 7f34b9d52d76aea875dc786f0c4d38f9cc6a6f5e

Audit Summary

Delivery Date	October 16, 2025
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
● Critical	0	0	0	0	0	0
● High	8	0	0	0	0	8
● Medium	15	0	0	1	0	14
● Low	10	0	0	2	0	8
● Info	4	0	0	0	0	4

Audit Scope & Methodology

Scope and details:

Contract,source,total,comment:

```
truth-contracts/src/FeeCollector.sol,143,254,67
truth-contracts/src/OrderManager.sol,440,711,143
truth-contracts/src/TruthMarketHook.sol,120,167,15
truth-contracts/src/validators/TruthMarketSwapValidator.sol,37,53,5
truth-contracts/src/base/DeltaResolver.sol,80,145,34
truth-contracts/src/base/ExecutionDeferer.sol,42,84,23
truth-contracts/src/base/OrderManagerState.sol,8,11,0
truth-contracts/src/base/PaymentDeferer.sol,29,60,17
truth-contracts/src/libraries/OrderBook.sol,138,249,67
truth-contracts/src/libraries/OrderValidation.sol,39,49,0
truth-contracts/src/libraries/PackedOrderId.sol,67,81,2
truth-contracts/src/libraries/TransientSlot.sol,73,183,89
truth-contracts/src/interfaces/IOrderManager.sol,52,78,2
truth-contracts/src/interfaces/ISwapValidator.sol,6,15,7
source count: {
  total: 2140,
  source: 1274,
  comment: 471,
  single: 364,
  block: 107,
  mixed: 10,
  empty: 405,
  todo: 0,
  blockEmpty: 0,
  commentToSourceRatio: 0.36970172684458397
}
```

Additionally, the following diffs in 4 files from commit b44c9834d6cf3ff93e949586dcff54190db94b23 to f8ad921851f264c3443b09067c0837b05d6dbb0c:

Oracle bonds diff: <https://www.diffchecker.com/t9qb9gZA/>

Oracle Council diff: <https://www.diffchecker.com/gGv2Znip/>

TruthMarketManager diff: <https://www.diffchecker.com/MYhL8Zqi/>

TruthMarket diff: <https://www.diffchecker.com/dzfBy5Dn/>

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: <i>High</i>	Impact: <i>Medium</i>	Impact: <i>Low</i>
Likelihood: <i>High</i>	● Critical	● High	● Medium
Likelihood: <i>Medium</i>	● High	● Medium	● Low
Likelihood: <i>Low</i>	● Medium	● Low	● Low

Impact

- High** Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.
- Medium** A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.
- Low** Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

- High** The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.
- Medium** An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.
- Low** Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.
Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.



















Invariants Assessed

During Guardian’s review of Trueo, fuzz-testing was performed on the protocol’s main functionalities. Given the dynamic interactions and the potential for unforeseen edge cases in the protocol, fuzz-testing was imperative to verify the integrity of several system invariants.

Throughout the engagement the following invariants were assessed for a total of 1,000,000+ runs with a prepared fuzzing suite.

ID	Description	Tested	Passed	Remediation	Run Count
CREATE-01	After creating an order, the pending orders count should increase by 1	✓	✓	✓	1m+
CREATE-02	After creating an order, the order should have positive liquidity	✓	✓	✓	1m+
CREATE-03	After creating an order, the user's balance should decrease	✓	✓	✓	1m+
ORDER-01	Expected order count should match total active orders	✓	✓	✓	1m+
ORDER-02	A limit order zeroForOne should never be set at a tick below or equal to current tick and vice versa	✓	✗	✗	1m+
ORDER-03	pendingOrder liquidity should always match with Uniswap position	✓	✓	✓	1m+
ORDER-04	Order owner's balance of token0/1 (or ERC6906 claims) should always increase after an order is deleted	✓	✓	✓	1m+
ORDER-05	All pending orders should have liquidity > 0	✓	✓	✓	1m+
ORDER-06	Tick range should be valid for all orders	✓	✓	✓	1m+
ORDER-07	OrderBook active orders should account for partial fills and deferred orders	✓	✗	✗	1m+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
ORDER-08	Cancelled orders should be completely removed from pending tracking				1m+
ORDER-09	Deferred execution orders should still exist in pending orders				1m+
ORDER-10	Highest and lowest active ticks should always be within valid tick range				1m+
DEFERRED-01	Sum of deferredPayments.amount should be less than or equal to OrderManager's ERC6909 claims balance				1m+
DEFERRED-02	If deferred nonce increased, then OrderManager's balance should increase				1m+
FEE-01	After a swap, if fees are enabled accumulated fees should increase in FeeCollector				1m+
FEE-02	After withdrawPoolFees, fee recipients should have increased balances				1m+

Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Deferred Orders Can Be Executed Out Of Bounds	DoS	● High	Resolved
H-02	Down-Move Pushback Missing In _executeOrder	Logical Error	● High	Resolved
H-03	Native ETH And Blacklist Tokens Can Cause DOS	DoS	● High	Resolved
H-04	Off-By-One Tick Can Prevent Order Execution	Logical Error	● High	Resolved
M-01	Inconsistent Minimum Amount Validation	Validation	● Medium	Resolved
M-02	Deferred Payment Hash Collision Risk	Logical Error	● Medium	Resolved
M-03	Griefing Via Forced Range Shift In Partial Order	DoS	● Medium	Acknowledged
M-04	cancelOrder Lacks Minimum Output Protection	Logical Error	● Medium	Resolved
M-05	Off-By-One Tick Can Block Valid Orders	Logical Error	● Medium	Resolved
M-06	unlockCallback Missing Deferred Payment Handling	Logical Error	● Medium	Resolved
L-01	Rescue Function Clears All Deferred Payments	Warning	● Low	Resolved
L-02	Incompatible Blacklist Checks Can Block Transfer	DoS	● Low	Acknowledged
L-03	Unhandled InsufficientDeposits In cancelOrder	Logical Error	● Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-04	Incorrect Handling Of Non-PartialFill Orders	Logical Error	<div><div></div>Low</div>	Resolved
L-05	Inefficient Max Execution Handling	DoS	<div><div></div>Low</div>	Resolved

H-01 | Deferred Orders Can Be Executed Out Of Bounds

Category	Severity	Location	Status
DoS	● High	ExecutionDeferer.sol: 30	Resolved

Description

If there are enough limit orders so that the number of orders exceeds `maximumExecutionCount`, the execution of the orders is deferred until the admin calls the `resolveDeferredExecution()` function.

However, when the admin calls this function, there is no check that the current tick of the pool falls within the deferred orders' range, meaning they will be executed out of range and the order owners will receive a different distribution of `token0/token1` than expected.

This can occur naturally or intentionally via griefing attack.

Recommendation

In the `resolveDeferredExecution()` function, add a check that the current pool tick falls within the specified orders' range.

Resolution

Trueo Team: Resolved.

H-02 | Down-Move Pushback Missing In _executeOrder

Category	Severity	Location	Status
Logical Error	● High	OrderManager.sol: 440	Resolved

Description [PoC](#)

The `_executeOrder` function should push an order back into the order book when a swap crosses its boundary tick but nothing fills. This works on upward moves, but the downward path is not handled.

For a down move (`fromTick > toTick`), `moveTick` scans `[toTick, fromTick + 1)` and clears boundary entries in that range. `_executeOrder` then only checks the opposite interval `[fromTick, toTick)` to decide whether to push the order back, which never matches on a down move.

As a result, the order is removed from the `orderBook` without being executed or deferred. It still exists in `pendingOrders` (so it can be cancelled) but it will not execute.

In practice a malicious swapper can “cancel” out a `zeroForOne` order by moving price down across its upper boundary, stripping it from the order book. Example: order `[60, 120]`. Start at tick 0; swap up `0→120` so `moveTick` uses `[0, 120)` and leaves `orders[120]`.

Then swap down `120→60`; `moveTick` uses `[60, 121)` and clears `orders[120]`. `_executeOrder` checks the other interval for the down move, skips `pushOrder`, and the order is no longer in the `orderBook` while still listed in `pendingOrders`.

Recommendation

Update `_executeOrder` to handle the down-move case correctly by ensuring that orders crossing their boundary without execution are consistently pushed back into the `orderBook`.

Resolution

Trueo Team: Resolved.

H-03 | Native ETH And Blacklist Tokens Can Cause DOS

Category	Severity	Location	Status
DoS	● High	DeltaResolver.sol: 72	Resolved

Description

If the Truemarket pool involves the use of either native ETH or a blacklistable token such as USDC, a single user can cause the execution of all orders to fail.

When a swap moves the tick past the orders' tick range, the `OrderManager` will attempt to execute them.

However, if a single user is on the token blacklist or places the order from a contract that reverts when it receives ETH, the execution of the entire order stack will revert.

The revert takes place in the `poolManager.take()` function which attempts to send the native or blacklistable token to the user.

This may occur naturally if contracts attempt to place orders but have not implemented a receive or fallback function.

Recommendation

It is recommended to wrap the call to the `take()` function in a low-level call. If a revert is detected, the tokens can be stored in the contract for later retrieval.

Resolution

Trueo Team: Resolved.

H-04 | Off-By-One Tick Can Prevent Order Execution

Category	Severity	Location	Status
Logical Error	● High	TruthMarketHook.sol: 109	Resolved

Description [PoC](#)

Uniswap’s swap logic has a known quirk: when `result.sqrtPriceX96 = step.sqrtPriceNextX96` at the end of a swap step and the direction is `zeroForOne`, the pool sets the current tick to `tickNext - 1`. This means the pool’s current tick may differ from the tick derived from `sqrtPriceX96`.

In `TruthMarketHook`, the `toTick` value is obtained using `TickMath.getTickAtSqrtPrice(sqrtPriceX96)` from:

```
(uint160 sqrtPriceX96,,,) = poolManager.getSlot0(key.toId());
int24 toTick = TickMath.getTickAtSqrtPrice(sqrtPriceX96);
```

If the above Uniswap quirk is triggered, `toTick` may be off by one compared to the pool’s actual tick. This value is later passed into `OrderBook.moveTick`, which can cause limit orders at the affected tick to remain uncleared and unexecuted.

As a result, users may miss their intended execution at the desired price, losing an opportunity to receive the tokens they expected.

Recommendation

In `TruthMarketHook`, obtain the tick directly from `slot0` rather than re-calculating it from `sqrtPriceX96`, and account for this Uniswap tick adjustment behavior.

Resolution

Trueo Team: Resolved.

M-01 | Inconsistent Minimum Amount Validation

Category	Severity	Location	Status
Validation	● Medium	OrderValidation.sol: 42	Resolved

Description

The `validateMinimumAmount` function, used in `createOrder`, enforces that the order amount exceeds the contract's `minimumLiteralAmount`. This is done by dividing `amountIn` by the `tokenIn`'s decimals and comparing it to the threshold.

The issue is that a single `minimumLiteralAmount` value (currently set to 1) is applied to all tokens, regardless of their market value.

As a result, the effective minimum differs drastically between tokens. For example, if `tokenIn` is USDC, the minimum is only slightly above one USDC, while if `tokenIn` is ETH, the minimum becomes one whole ETH, which is significantly higher in value.

This inconsistent enforcement can unintentionally restrict orders with certain tokens while being too permissive with others.

Recommendation

Consider implementing token-specific minimums, or scale the minimum by relative value, to ensure consistent and fair minimum order requirements across different tokens.

Resolution

Trueo Team: Resolved.

M-02 | Deferred Payment Hash Collision Risk

Category	Severity	Location	Status
Logical Error	● Medium	PaymentDeferer.sol: 68	Resolved

Description

When payment is deferred, a `DeferredPayment` struct is hashed to generate a `hashId`. However, the `hashId` only depends on (currency, amount, to, timestamp).

This can lead to collisions — for example, if Pool A and Pool B both defer a payment of 100 USDC to the same user in the same block, the resulting `hashId` will be identical.

As a result, one payment silently overwrites the other, causing loss of funds to the affected order owner.

Recommendation

Consider incorporating `poolId` and a unique increasing nonce into the `hashId`. For example:

```
bytes32 hashId = keccak256(
  abi.encode(payment, poolId, deferredPaymentNonce++)
);
```

Resolution

Trueo Team: Resolved.

M-03 | Griefing Via Forced Range Shift In Partial Order

Category	Severity	Location	Status
DoS	● Medium	OrderManager.sol: 617	Acknowledged

Description [PoC](#)

In `_partialFillOrder`, when the recalculated tick range collapses (i.e., the new lower and upper ticks are equal), the order's limit range is shifted upward by one tick spacing:

```
if (newTickLower > newTickUpper) {
    newTickUpper = newTickLower + key.tickSpacing;
}
```

This introduces a griefing vector. An attacker can repeatedly swap to exactly one tick above the order's lower tick (for a `zeroForOne` order).

Each forced adjustment shifts the order's range upward, effectively altering the execution level over time.

As a result, the order may eventually be fully filled at an unintended, less favorable price. The order owner could therefore receive an unexpected amount of tokens once the order completes.

Recommendation

One possible approach is to settle the order immediately when the range collapses, rather than automatically recreating it at a shifted range.

However, this introduces a different risk where orders may be settled prematurely. At minimum, this behavior and its implications should be clearly documented so that users understand the possibility of griefing when enabling partial fills.

Resolution

Trueo Team: Acknowledged.

M-04 | cancelOrder Lacks Minimum Output Protection

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 355	Resolved

Description

Users can call the `cancelOrder` function to cancel their order, which withdraws their liquidity position from the pool. However, no user-defined minimum output is specified.

As a result, if the price moves against the user before the call, the withdrawn amounts may consist of a different token mix than intended or result in a strictly lower overall value. This exposes users to unexpected execution outcomes when canceling an order.

Recommendation

Add user-specified minimum output parameters to `cancelOrder`, similar to how the Uniswap v4 periphery enforces slippage protections when modifying liquidity.

Resolution

Trueo Team: Resolved.

M-05 | Off-By-One Tick Can Block Valid Orders

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 288	Resolved

Description

During order creation, the tick is derived from `sqrtPriceX96` using `TickMath.getTickAtSqrtPrice` before being passed into `validateTickThreshold`:

```
int24 tick = TickMath.getTickAtSqrtPrice(sqrtPriceX96);
OrderValidation.validateTickThreshold(
    tick, params.tickLower, params.tickUpper, tickThreshold, params.zeroForOne
);
```

Because of the Uniswap quirk where, after a `zeroForOne` swap, the pool sets the tick to `tickNext - 1` when `sqrtPriceX96 = step.sqrtPriceNextX96`, the derived tick may be off by one relative to the actual pool tick.

This can cause valid `zeroForOne` orders to be incorrectly rejected. For example:

- A prior `zeroForOne` swap moved the price exactly to the tick = -120 boundary. Due to the quirk, the pool’s current tick becomes -121.
- A user creates a limit order to buy `token0` between ticks -120 and -60. This should be valid since the order’s `fromTick` (-120) is greater than the pool tick (-121).
- However, since tick was derived from `sqrtPriceX96` (returning -120), the validation incorrectly blocks the order.

As a result, users may be prevented from submitting valid `zeroForOne` orders in these circumstances.

Recommendation

When validating ticks, rely on the pool’s current tick from `slot0` rather than recalculating it from `sqrtPriceX96`.

Resolution

Trueo Team: Resolved.

M-06 | unlockCallback Missing Deferred Payment Handling

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 91	Resolved

Description

The `_takeOrSettle` function includes an edge case where, if the `poolManager` does not have sufficient deposits, it returns `ResolveResult.InsufficientDeposits` without sending owed tokens to the user.

In most places where `_takeOrSettle` is called, this case is handled by passing the owed amount to `_deferPayment`, which creates a deferred claim for later resolution.

In the `unlockCallback` function, however, this defer step is missing. If `_takeOrSettle` returns `InsufficientDeposits`, no deferred claim is created and the user’s owed amount goes unaccounted.

Furthermore, `unlockCallback` calls `_clearDelta` at the end, which clears the user’s expected amount as “dust delta,” making it permanently unrecoverable.

Recommendation

Modify the `unlockCallback` function to defer payments via `_deferPayment` whenever `_takeOrSettle` returns `InsufficientDeposits`, ensuring users’ owed amounts are accounted for.

Resolution

Trueo Team: Resolved.

L-01 | Rescue Function Clears All Deferred Payments

Category	Severity	Location	Status
Warning	● Low	OrderManager.sol: 184	Resolved

Description

The `rescueToken` function in the `OrderManager` contract withdraws all minted tokens from Uniswap. As a result, any outstanding deferred payments, including those from other pools, will fail when `resolveDeferredPayment` is later called.

These payments would then need to be handled manually through the `rescueToken` function instead. Additionally, unlike the normal resolution flow, the `hashId` entries in the `deferredPayments` mapping are not deleted, leaving stale records that no longer correspond to valid claims.

Recommendation

If this behaviour is intended, ensure it is clearly documented and that operators are aware deferred payments will need to be managed manually after calling `rescueToken`.

Resolution

Trueo Team: Resolved.

L-02 | Incompatible Blacklist Checks Can Block Transfer

Category	Severity	Location	Status
DoS	● Low	TruthMarketV2.sol: 572	Acknowledged

Description

In the `_transferBondFromMarket` and `_transferReward` functions, the receiver is validated against a blacklist using:

```
try IBlacklistable(address(token)).isBlacklisted(_account) returns (bool result)
```

This works only if the token implements the `isBlacklisted` interface. However, if the token uses a different mechanism for blacklist functionality—such as WLF1’s `Usd1` token, which uses `frozen` instead—the transfer would still fail.

In such a case, critical settlement operations could be blocked, even though the blacklist check was not applicable.

Recommendation

Replace the current `try/catch` implementation to ensure that failures in the blacklist check or transfer do not block the system.

```
try token.safeTransfer(_account, _amount) {}  
catch { token.safeTransfer(marketManager.safeBoxAddress(), _amount); }
```

Resolution

Trueo Team: Acknowledged.

L-03 | Unhandled InsufficientDeposits In cancelOrder

Category	Severity	Location	Status
Logical Error	● Low	OrderManager.sol: 104	Resolved

Description

When `cancelOrder` is called, the user’s liquidity is removed from the pool. During the callback, `_takeOrSettle` is responsible for transferring the owed amounts back to the user.

However, unlike order execution, the cancel flow does not properly handle the case where the owed amount exceeds the balance available in the pool manager.

In execution paths, this condition is handled with `InsufficientDeposits`, and a deferred payment is recorded.

In the cancel flow, if this condition were to occur, the `unlockCallback` proceeds to `_clearDelta`, which clears all deltas and results in the user potentially losing all tokens they were owed.

Although no concrete scenario was identified where the pool manager would have insufficient funds during cancellation, the current design leaves the risk of permanent loss in such a situation.

Recommendation

Update the cancel flow to handle insufficient deposits. For example:

```
if (_takeOrSettle(key.currency0, amount0, owner, address(this)) =
ResolveResult.InsufficientDeposits) {
_deferPayment(key.currency0, uint256(uint128(amount0)), owner);
}
```

Resolution

Trueo Team: Resolved.

L-04 | Incorrect Handling Of Non-PartialFill Orders

Category	Severity	Location	Status
Logical Error	● Low	OrderManager.sol: 491	Resolved

Description

At the end of `_executeOrder`, orders that are not executed are pushed back into the order book. These orders are currently pushed at both `tickLower` and `tickUpper`.

However, for orders with `enablePartialFill = false`, only a single order should be reinserted at the `tickThreshold`.

This behavior is correctly implemented in `createOrder` but not mirrored in `_executeOrder`, leading to inconsistent handling of non-partial-fill orders.

Recommendation

Update `_executeOrder` to push only a single order at `tickThreshold` when `enablePartialFill = false`, ensuring consistent logic with `createOrder`.

Resolution

Trueo Team: Resolved.

L-05 | Inefficient Max Execution Handling

Category	Severity	Location	Status
DoS	● Low	OrderManager.sol: 208	Resolved

Description

The `movePoolTick` function defers execution whenever the number of in-range orders exceeds the contract's configured `maximumExecutionCount`. Once deferred, the orders are later resolved in smaller batches by a backend service.

However, the current implementation either defers all orders or executes all orders, rather than partially executing up to the maximum allowed. This introduces inefficiency since orders that could have been executed immediately are instead unnecessarily deferred.

Additionally, orders in the opposite direction are still included in the in-range order count even though they are not eligible for execution and will ultimately just be pushed back. This inflates the count and increases the chance of triggering deferral.

With a low `minAmount` requirement when creating orders, this mechanism can be abused by malicious users to spam the system with many minimal orders, forcing deferrals on order execution.

Recommendation

Update the `movePoolTick` logic to execute orders up to the `maximumExecutionCount` and consider stricter minimum order requirements to prevent spam.

Resolution

Trueo Team: Resolved.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	User Can Arbitrarily Move Order Ticks To DoS	DoS	● High	Resolved
M-01	Deferred Payments Still Target Blacklisted User	Logical Error	● Medium	Resolved
M-02	Off-By-One Tick Issue In validateAfterSwap	Logical Error	● Medium	Resolved
M-03	Deferred Orders Stranded If Tick Range Invalid	Logical Error	● Medium	Resolved
M-04	_executeOrder Pushback Includes Settled Orders	Logical Error	● Medium	Resolved
M-05	Deferred Orders Misplaced Due To Stale Tick	Logical Error	● Medium	Resolved
L-01	Reentrancy Best Practice For Deferred Actions	Best Practices	● Low	Resolved
I-01	Unused Error In FeeCollector	Error	● Info	Resolved
I-02	Redundant Timestamp In DeferredPayment Hash	Gas Optimization	● Info	Resolved
I-03	Invalid Comment On _validateAndAdjustTickRange	Documentation	● Info	Resolved
I-04	Line Numbers In Comments Don't Line Up	Best Practices	● Info	Resolved

H-01 | User Can Arbitrarily Move Order Ticks To DoS

Category	Severity	Location	Status
DoS	● High	OrderManager.sol: 683-685	Resolved

Description

There is a small error when handling partial fill orders that allows an arbitrary user to cause orders with `partialFillEnabled` to be unfillable. Essentially, the attacker is able to move an order's ticks up or down to extreme values, ensuring that the orders are never executed.

Take the following example:

- Current tick is 0
- Alice creates a limit order with partial fills enabled for [180, 300]
- Bob swaps the tick up to 181. This initiates the partial fill logic which will create a new limit order for Alice at [240, 300].
- Bob then swaps the tick up to 241. This creates a new limit order for Alice at [300, 360].

Bob can continuously push all limit orders with partial fills enabled to extreme tick levels.

The issue occurs because the partial fill logic simply creates a new limit order at the next tick spacing above the `currentTick`. If the new `lowerTick` is greater than or equal to the previous `upperTick`, then the `upperTick` is increased by one tick spacing.

```
if (newTickLower > newTickUpper) {
    newTickUpper = newTickLower + key.tickSpacing;
}
```

Therefore, the order can be pushed an unlimited amount of ticks.

It's also important to note that the attacker would swap to 1 tick larger than the lower boundary to ensure that minimal token amounts are settled for the user.

Recommendation

When checking if the order is partially fillable, simple return false if `tickUpper - tickLower < TICK_SPACING`. This will ensure that partial orders that only span 1 `TICK_SPACING` can only be fulfilled fully.

Resolution

Trueo Team: Resolved.

M-01 | Deferred Payments Still Target Blacklisted User

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 804	Resolved

Description

In `_handleDeltaResolveResult`, if the resolve result is `TakeFailed` (likely caused by a blacklisted address), the payment is deferred:

```
} else if (result = ResolveResult.TakeFailed) {  
    // Handles blacklist scenarios to prevent DoS attacks  
    _deferPayment(currency, amount, taker, PaymentDeferredReason.UnableToTransfer);  
}
```

However, the deferred payment is still assigned to the same taker who could not receive funds due to being blacklisted. This does not resolve the underlying issue, as the funds remain stuck in the contract and may be permanently unclaimable.

Recommendation

If take fails due to blacklisting, consider deferring the payment to an admin-controlled safe address instead of the blacklisted taker. This allows funds to be recovered or manually remediated, avoiding permanent lockup.

Resolution

Trueo Team: Resolved.

M-02 | Off-By-One Tick Issue In validateAfterSwap

Category	Severity	Location	Status
Logical Error	● Medium	TruthMarketSwapValidator.sol: 43	Resolved

Description

The Uniswap off-by-one tick issue persists in `validateAfterSwap` where `currentTick` may be one tick higher than actual tick:

```
(uint160 sqrtPriceX96,,, ) = poolManager.getSlot0(poolKey.toId());  
  
int24 currentTick = TickMath.getTickAtSqrtPrice(sqrtPriceX96);
```

As a result, in edge scenarios, `currentTick` could have moved below `tickLowerBoundary` but the validation logic fails to catch it.

Recommendation

Use the tick reported directly from `slot0` rather than recomputing it from `sqrtPriceX96`.

Resolution

Trueo Team: Resolved.

M-03 | Deferred Orders Stranded If Tick Range Invalid

Category	Severity	Location	Status
Logical Error	● Medium	ExecutionDeferer.sol: 49	Resolved

Description

The `_resolveDeferredExecution` function was modified to check tick range validity before executing orders. If the tick range is not valid, the order execution is skipped entirely:

```
if (isValid) {
    _executeOrders(key, deferred.orderIds, adjustedFromTick, adjustedToTick);
    emit IOrderManager.DeferredExecutionResolved(poolId, hashId, adjustedFromTick, adjustedToTick);
} else {
    emit IOrderManager.DeferredExecutionSkipped(poolId, hashId, deferred.fromTick, deferred.toTick);
}
```

However, this is incorrect because the order has already been removed from the order book at this stage. While it is true that liquidity should not be pulled out in this case, the order must be reinserted back into the order book.

Failing to do so results in the affected orders becoming stranded, no longer present in the order book with the only recourse to cancel the order.

Recommendation

Initially we considered setting all conditions in `_validateAndAdjustTickRange` to be valid such that `_executeOrders` will always be called to handle order re-insertion to order book.

However, it may be quite tricky to correctly adjust tick range such that `_isTickInRange` will pass and ensure the order is re-inserted.

Instead, we could consider in `_resolveDeferredExecution`, if order is invalid, to do the re-insertion to order book there directly:

```
if (isValid) {
    _executeOrders(key, deferred.orderIds, adjustedFromTick, adjustedToTick);
    emit IOrderManager.DeferredExecutionResolved(poolId, hashId, adjustedFromTick, adjustedToTick);
} else {
    << re-insert order to orderBook >>
    emit IOrderManager.DeferredExecutionReinserted(poolId, hashId, deferred.fromTick, deferred.toTick);
}
```

Resolution

Trueo Team: Resolved.

M-04 | _executeOrder Pushback Includes Settled Orders

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 486	Resolved

Description

The `_partialFillOrder` function calls `_settleOrder`, which deletes the order from the `pendingOrders` mapping and returns `hasNewOrder` as false when `newLiquidity` is zero.

In `_executeOrder`, because `hasNewOrder` is false, the early return is skipped and the function proceeds to the pushback logic. This can result in pushing back an order that no longer exists.

Although the order is skipped when eventually processed, it still contributes to `totalOrders`. This can artificially inflate the order count, potentially exceeding `maximumExecutionCount` and causing deferred batches that would not otherwise be necessary.

Recommendation

Ensure `_executeOrder` does not include nonexistent orders in pushback logic to prevent unnecessary batch deferrals.

Resolution

Trueo Team: Resolved.

M-05 | Deferred Orders Misplaced Due To Stale Tick

Category	Severity	Location	Status
Logical Error	● Medium	ExecutionDeferer.sol: 35	Resolved

Description

The `_resolveDeferredExecution` function resolves a batch of deferred order executions. The batch's `fromTick` and `toTick` are validated against the current pool state, and if still in range, `_executeOrders` uses the original batch range.

The issue arises because `_executeOrders` passes `toTick` to `_partialFillOrder`, which then uses it as `currentTick` to calculate `newTickLower` and `newTickUpper` for the remaining order. For regular swaps, `toTick` matches `currentTick`, so this works correctly.

However, for deferred batches, the batch's `toTick` may differ from the pool's actual current tick even when in range. As a result, the `newTickLower` and `newTickUpper` calculation is based on the batch's `toTick` rather than the true `currentTick`, potentially misplacing the remaining order in the orderbook.

Recommendation

Adjust `_partialFillOrder` to ensure `newTickLower` and `newTickUpper` are always calculated using the actual current tick of the pool rather than the batch's `toTick`.

Resolution

Trueo Team: Resolved.

L-01 | Reentrancy Best Practice For Deferred Actions

Category	Severity	Location	Status
Best Practices	● Low	PaymentDeferer.sol: 49, ExecutionDeferer.sol: 52	Resolved

Description

In `_resolveDeferredPayment`, external calls (`poolManager.burn`, `poolManager.take`) are called before the state updates of deleting `deferredPayments[hashId]` and decreasing `_deferredAmounts`.

While it is unlikely for the external call to re-enter the contract, it is generally best practice to update state variables before the external calls.

This issue also applies to `_resolveDeferredExecution` in `ExecutionDeferrer`.

Recommendation

Consider reordering the flow in `_resolveDeferredPayment` to:

```
uint256 amt = p.amount;

address curId = p.currency.toId();

delete deferredPayments[hashId];

_deferredAmounts[curId] -= amt;

poolManager.burn(...);

poolManager.take(...);
```

and for `_resolveDeferredExecution` to:

```
delete deferredExecutions[poolId][hashId];

_validateAndAdjustTickRange(...);

_executeOrders();

...
```

Resolution

Trueo Team: Resolved.

I-01 | Unused Error In FeeCollector

Category	Severity	Location	Status
Error	● Info	FeeCollector.sol: 49	Resolved

Description

The `InsufficientPoolBalance` error is never actually used anywhere in the current code—it’s only declared at line 49 of `FeeCollector.sol` and never thrown or referenced after that.

Recommendation

Consider removing the error.

Resolution

Trueo Team: Resolved.

I-02 | Redundant Timestamp In DeferredPayment Hash

Category	Severity	Location	Status
Gas optimization	● Info	PaymentDeferer.sol: 74	Resolved

Description

In the DeferredPayment struct, a 160-bit timestamp is stored and included in the hash:

```
DeferredPayment memory payment = DeferredPayment(currency, amount, to, uint160(block.timestamp), currentNonce);
```

However, it is not necessary to include timestamp in the hash, now that we have introduced a nonce for hash uniqueness.

Recommendation

Remove the timestamp field from the DeferredPayment struct if it is not required for business logic or accounting. Relying on (currency, amount, to, nonce) is sufficient for uniqueness, while reducing per-entry storage costs.

Resolution

Trueo Team: Resolved.

I-03 | Invalid Comment On _validateAndAdjustTickRange

Category	Severity	Location	Status
Documentation	● Info	ExecutionDeferer.sol: 73, ExecutionDeferer.sol: 87	Resolved

Description

In the `_validateAndAdjustTickRange` function, this comment which is mentioned twice is not accurate:

```
// If current tick has already passed the entire range, execution is invalid
```

If the current has passed an entire range, the order is properly filled and execution should be valid instead of invalid.

Recommendation

Consider removing the comment altogether as each case has its own own specific comment.

Resolution

Trueo Team: Resolved.

I-04 | Line Numbers In Comments Don't Line Up

Category	Severity	Location	Status
Best Practices	● Info	OrderManager.sol: 598-601	Resolved

Description

In OrderManager.sol, there are some comments that reference functionality by line number, however these references seem to be outdated.

Recommendation

Fix the line references.

Resolution

Trueo Team: Resolved.

Remediations V2 Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Incorrect price used to compute newLiquidity	Error	● High	Resolved
M-01	Cancelled Orders Removed From Wrong Tick	Logical Error	● Medium	Resolved
M-02	Inconsistent Pushback Handling	Logical Error	● Medium	Resolved
M-03	Unaccounted Zero-Case in Tick Adjustment	Logical Error	● Medium	Resolved
L-01	Unclear Behavior & Comment in _partialFillOrder	Logical Error	● Low	Resolved
L-02	Potential Order ID Exhaustion in getNextOrderId	Warning	● Low	Acknowledged
L-03	_takeOrSettleAll Omits settle after Payment	Error	● Low	Resolved
L-04	Unsafe Casting of Negative Amounts	Warning	● Low	Resolved

H-01 | Incorrect Price Used To Compute newLiquidity

Category	Severity	Location	Status
Error	● High	OrderManager.sol: 801	Resolved

Description [PoC](#)

In `_partialFillOrder`, when calculating new liquidity to be added, the current implementation uses price obtained from current tick:

```
//@audit wrong way to obtain current price
uint160 sqrtPriceX96 = TickMath.getSqrtPriceAtTick(currentTick);
..
newLiquidity = LiquidityAmounts.getLiquidityForAmounts(
sqrtPriceX96,
TickMath.getSqrtPriceAtTick(newTickLower),
TickMath.getSqrtPriceAtTick(newTickUpper),
amount0,
amount1
);
```

However, this is incorrect as actual current price can lie between ticks. As a result, `getLiquidityForAmounts` may return a `newLiquidity` that requires more `amount0` / `amount1` than is available.

This leads to an underflow during settlement, causing a revert and disrupting the entire execution process.

Recommendation

Use price obtained from `slot0` instead:

```
- uint160 sqrtPriceX96 = TickMath.getSqrtPriceAtTick(currentTick);
+ (uint160 sqrtPriceX96,,, ) = poolManager.getSlot0(key.toId());
```

Resolution

Trueo Team: Resolved.

M-01 | Cancelled Orders Removed From Wrong Tick

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 461-462	Resolved

Description

A recent update introduced an adjustment where partial fill orders are pushed one tick spacing away from their `tickLower` / `tickUpper` bounds:

```
if (params.enablePartialFill) {
  orderBook.pushOrder(
    params.zeroForOne ? params.tickLower + params.poolKey.tickSpacing : params.tickLower, orderId
  );
  orderBook.pushOrder(
    params.zeroForOne ? params.tickUpper : params.tickUpper - params.poolKey.tickSpacing, orderId
  );
}
```

However, `cancelOrder` still attempts to remove orders using the original `tickLower` / `tickUpper`, without accounting for the tick spacing offset.

This mismatch leaves stale references in the order book. Cancelled orders are not properly removed, causing the order book to accumulate empty entries.

Over time, this creates unnecessary bloat and larger order ID batches that swaps must iterate through, increasing gas costs and reducing efficiency.

Recommendation

Update `cancelOrder` to mirror the same tick placement logic used in `createOrder` for partial fills.

Resolution

Trueo Team: Resolved.

M-02 | Inconsistent Pushback Handling

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 545	Resolved

Description

The `_executeOrder` function pushes back only a single order into the orderbook when handling partial fills, that end up with the same tick thresholds.

However, this specific case is not accounted for in the `_executeOrders` function’s pushback logic. As a result, the system can push back two identical orders into the orderbook for the same partial order, creating unintended duplication.

Recommendation

Consider aligning the pushback logic in `_executeOrders` with the behaviour in `_executeOrder` to ensure partial orders are handled consistently and to prevent duplicate entries in the orderbook.

Resolution

Trueo Team: Resolved.

M-03 | Unaccounted Zero-Case In Tick Adjustment

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 725	Resolved

Description

The `_partialFillOrder` function adjusts tick ranges after a partial fill.

Specifically, it deducts `tickSpacing` from `newTickLower` in the `zeroForOne` case when `newTickLower` is less than zero, and increments `newTickUpper` by `tickSpacing` in the `oneForZero` case when `newTickUpper` is greater than zero.

However, the scenario where `newTickLower` or `newTickUpper` equals exactly zero is not handled in either branch. As a result, the recalculated tick range may exclude the current tick.

Recommendation

Consider handling the condition where `newTickLower` or `newTickUpper` is zero to ensure that the adjusted tick range always encompasses the current tick.

Resolution

Trueo Team: Resolved.

L-01 | Unclear Behavior & Comment In _partialFillOrder

Category	Severity	Location	Status
Logical Error	● Low	OrderManager.sol: 599-600	Resolved

Description

In `_partialFillOrder`, there are two early return cases:
Case 1 – Tick movement too small:

```
if (newTickLower = oldTickLower) {
  return (false, 0, 0);
}
```

Case 2 – Ticks hitting the max boundary:

```
if (newTickLower > maxUsableTick || newTickUpper > maxUsableTick) {
  return (false, 0, 0);
}
```

Both cases return `(false, 0, 0)`. As a result, `_executeOrder` also returns false. However, the comment above is misleading:

```
// if both ticks are 0, it means order will move out of range
// so no new order is created
if (hasNewOrder & newTickLower = 0 newTickUpper = 0) {
  return false;
}
```

In practice, when false is returned, `executeOrders` sets `executed = false`, which then triggers the pushback logic to create a new order.

This creates ambiguity:

- Case 1 (movement too small): unclear whether a new order should be pushed back.
- Case 2 (ticks at min/max): seems clear that no new order should be created.

Recommendation

Re-examine the logic in `_executeOrder` to ensure that Cases 1 and 2 are handled correctly. Update the comments to correctly describe the behavior—particularly whether false should lead to a new order being created or not in each scenario.

Resolution

Trueo Team: Resolved.

L-02 | Potential Order ID Exhaustion In getNextOrderId

Category	Severity	Location	Status
Warning	● Low	OrderBook.sol	Acknowledged

Description

The getNextOrderId function increments nextOrderId as a uint32 with unchecked arithmetic and skips over ID 0. When nextOrderId reaches type(uint32).max (4,294,967,295), it wraps around to 0, skips it, and reuses ID 1.

If an existing order with ID 1 is still active, the subsequent createOrder call will revert due to the pendingOrders[poolId][orderId].owner = address(0) check, blocking all future order creation.

This behavior can cause collisions once the ID space is exhausted, especially in high-throughput pools where large numbers of orders are created and cancelled.

Recommendation

Consider expanding the ID space (e.g., to uint64 or uint128) to make exhaustion practically unreachable. implement a recycling mechanism to safely reuse old order IDs once cancelled, ensuring no collisions with active orders.

Resolution

Trueo Team: Acknowledged.

L-03 | `_takeOrSettleAll` Omits `Settle` After Payment

Category	Severity	Location	Status
Error	● Low	DeltaResolver.sol: 127	Resolved

Description

`_takeOrSettleAll` handles positive and negative delta paths. In the negative-delta branch (where the contract owes tokens to the pool) the code calls `_pay(...)` but does not subsequently call `poolManager.settle(...)`.

Omitting this `settle` leaves the pool’s accounting unresolved and subsequently causes reverts (for example, in Uniswap V4’s unlock callback) because the pool expects a settle step after payments.

Currently this bug is not triggered by `OrderManager:_executeOrders` because that flow only produces positive deltas for fee transfers.

However, if `_takeOrSettleAll` is ever reused for contexts that produce negative deltas, this omission can leave pool state inconsistent and halt operations that rely on a proper settle (order execution, payment resolution, callbacks).

Recommendation

Call `poolManager.settle` after `_pay` to resolve outstanding delta.

Resolution

Trueo Team: Resolved.

L-04 | Unsafe Casting Of Negative Amounts

Category	Severity	Location	Status
Warning	● Low	OrderManager.sol: 867	Resolved

Description

In both `_safeTakeOrSettle` and `_safeTakeOrSettleAll`, when `amount` is negative (`int256 < 0`), it is cast directly to `uint256` before being passed to `_handleDeltaResolveResult`. This causes underflow wrapping – e.g., `uint256(-1)` becomes `2*256 - 1`.

While this wrapped value is currently harmless (since `_handleDeltaResolveResult` does not defer or act on negative deltas), it introduces latent risk:

- Future logic changes (e.g. deferred settlements, batching, or accounting extensions) might treat the wrapped value as a legitimate positive amount.
- This could lead to excessive token transfers, accounting corruption, or DoS if the system attempts to handle `uint256.maxsize` values.

Recommendation

Add an explicit check before casting to `uint256` to prevent unintended wrapping:

```
if (amount < 0) {
  return;
} else {
  _handleDeltaResolveResult(result, uint256(amount), currency, taker);
}
```

Resolution

Trueo Team: Resolved.

Remediations V3 Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Tick Misordering in _partialFillOrder	DoS	● High	Resolved
H-02	Missing Tick Boundary Checks in partialFillOrder	Logical Error	● Medium	Resolved
M-01	Duplicate Pushback on Deferred Partial Orders	Logical Error	● Medium	Pending

H-01 | Tick Misordering In _partialFillOrder

Category	Severity	Location	Status
DoS	● High	OrderManager.sol: 598	Resolved

Description [PoC](#)

The `_executeOrder` function determines whether a partial order can be fulfilled by calling `_isTickInRange`, which checks if the tick movement crossed the `fulfillThreshold`.

When resolving a deferred execution, however, the function uses the batch's `toTick` even if the current tick has already moved beyond it.

As a result, a partial order that should be fully filled is incorrectly treated as still partial, and execution proceeds to the partial fill logic.

In this path, the function recalculates the new tick range adjusting `newTickUpper` based on the current tick while keeping the previous `newTickLower` for the `oneForZero` case.

If the current tick has already passed the threshold, this results in misordered tick bounds (`tickLower > tickUpper`), causing the system to incorrectly attempt re-adding liquidity for an already-filled order which will also ultimately fail on the Uniswap side due to invalid tick ordering.

Recommendation

Modify `_executeOrder` to compare the live `currentTick` against the order's `fulfillThreshold`. If the current tick is already beyond the threshold, treat the order as fully filled rather than performing a partial re-add.

Resolution

Trueo Team: Resolved.

H-02 | Missing Tick Boundary Checks In partialFillOrder

Category	Severity	Location	Status
Logical Error	● High	OrderManager.sol	Resolved

Description [PoC](#)

In previous versions, `_partialFillOrder` included boundary checks to prevent re-adding liquidity when `newTickLower = newTickUpper` or when the new tick range exceeded usable bounds. These checks were recently removed:

```
if (zeroForOne) {
  if (newTickLower > newTickUpper) {
    newTickUpper = newTickLower + key.tickSpacing;
  }
  if (newTickLower > maxUsableTick || newTickUpper > maxUsableTick) {
    return (false, 0, 0);
  }
} else {
  if (newTickLower > newTickUpper) {
    newTickLower = newTickUpper - key.tickSpacing;
  }
  if (newTickLower < minUsableTick || newTickUpper < minUsableTick) {
    return (false, 0, 0);
  }
}
```

These checks ensured that new liquidity ranges were valid and non-overlapping. Without them, if the `currentTick` is close to an order's tick boundaries, `newTickLower` can equal `newTickUpper`, causing `getLiquidityForAmounts` to revert when attempting to compute liquidity for a zero-width range.

Recommendation

Reinstate the previous validation logic to ensure safety at tick boundaries.

Resolution

Trueo Team: Resolved.

M-01 | Duplicate Pushback On Deferred Partial Orders

Category	Severity	Location	Status
Logical Error	● Medium	OrderManager.sol: 560	Resolved

Description

The `resolveDeferredExecution` function processes deferred orders in smaller batches when they could not be executed in the `_afterSwap` hook due to exceeding `maximumExecutionCount`.

If an order is no longer in range, it is pushed back to the orderbook. For partial fill orders, both ticks are pushed back if they are in range and `thresholdLower = thresholdUpper`.

An edge case occurs when a partial order has its `tickLower` and `tickUpper` deferred within the same range.

If, at resolution time, the current tick is out of range, both sides push back the `tickLower` and the `tickUpper` resulting in the same order being pushed back twice.

Recommendation

Consider adding a condition to prevent duplicate pushbacks when both deferred ticks originate from the same order.

Resolution

Trueo Team: Resolved.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit <https://guardianaudits.com>

To view our audit portfolio, visit <https://github.com/guardianaudits>

To book an audit, message <https://t.me/guardianaudits>