Quantstamp Security Assessment Certificate

February 10th 2023 — Quantstamp Verified

Conveyor Finance

This audit report was prepared by Quantstamp, the leader in blockchain security.

Executive Summary

Type	Defi
Auditors	Marius Guggenmos, Senior Research Engineer Andy Lin, Senior Auditing Engineer Bohan Zhang, Auditing Engineer
Timeline	2022-09-19 through 2022-10-10
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Whitepaper
Documentation Quality	Medium
Test Quality	Low

Source Code

Repository	Commit
ConveyorLabs/Aggregator-v0	4182f2e initial audit

Total Issues	**29** (27 Resolved)
High Risk Issues	**7** (7 Resolved)
Medium Risk Issues	**6** (6 Resolved)
Low Risk Issues	**6** (6 Resolved)
Informational Risk Issues	**6** (6 Resolved)
Undetermined Risk Issues	**4** (2 Resolved)

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.
Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Fixed	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

Initial audit:
Conveyor finance is a platform for placing limit orders for decentralized exchanges. Overall, the code is documented well through the use of the NatSpec format. The tests proved to be insufficient since we found several logic errors that should be caught through them. We recommend improving the test suite by 1) increasing the overall coverage, 2) improving the quality of the existing tests by adding more verification checks of the expected results and side effects of the function calls under test, and 3) adding fuzzing tests for the math libraries.
In addition to logic bugs, the biggest issues we found are related to the almost complete lack of authorization, which leads to the contracts accepting forged orders and attackers able to steal user funds.

Fix review:
After the initial audit, the team significantly changed the design of the protocol. In our opinion, the changes greatly improved the protocol and made it more secure by reducing the complexity of the code. Please note however, that the fix review merely confirms whether the fixes for the findings of the initial report were implemented correctly. Therefore, the fix review does not constitute a full audit, since general restructuring and the addition of new code were not in scope.
The team addressed all of our findings. They also fixed most code documentation issues and implemented almost all of our best practice recommendations. Test coverage metrics remain relatively low and we recommend adding more tests to reach at least full line coverage.

Fix review 2:
The Conveyor Finance team reported an issue where the price simulation function for Uniswap V3 pools returned inaccurate results. We have investigated and confirmed the issue. Additionally, we reviewed the changes (code at commit hash `ffd27a1d`) that were provided by the team, most of which were related to fixing the reported issue. Apart from an unsafe cast that has already been fixed in commit `50c96ad8`, no other issues were identified.

ID	Description	Severity	Status
QSP-1	Stealing User and Contract Funds	High	Fixed
QSP-2	Missing Authorization for Execution Contracts	High	Fixed
QSP-3	Ignoring Return Value of ERC20 Transfer Functions	High	Fixed
QSP-4	Cancelling Order Provides Compensation Twice	High	Fixed
QSP-5	Updating an Existing Order Can Be Malicious	High	Fixed
QSP-6	Same Order Id Can Be Executed Multiple Times	High	Fixed
QSP-7	Incorrectly Computing the Best Price	High	Fixed
QSP-8	Reentrancy	Medium	Fixed
QSP-9	Not Cancelling Order as Expected	Medium	Fixed
QSP-10	Granting Insufficient Gas Credit to the Executor	Medium	Fixed
QSP-11	Integer Overflow / Underflow	Medium	Fixed
QSP-12	Updating Order Performs Wrong Total Order Quantity Accounting	Medium	Fixed
QSP-13	Not Always Taking Beacon Reward Into Account	Low	Fixed
QSP-14	Denial of Service Due to Unbound Iteration	Low	Mitigated
QSP-15	Missing Input Validation	Low	Fixed
QSP-16	Gas Oracle Reliability	Low	Fixed
QSP-17	Math Function Returns Wrong Type	Low	Fixed
QSP-18	Individual Order Fee Is Not Used in Batch Execution	Low	Fixed
QSP-19	Locking the Difference Between `beaconReward` and `maxBeaconReward` in the Contract	Informational	Fixed
QSP-20	Inaccurate Array Length	Informational	Fixed
QSP-21	`TaxedTokenLimitOrderExecution` Contains Code for Handling Non-Taxed Orders	Informational	Fixed
QSP-22	Unlocked Pragma	Informational	Fixed
QSP-23	Allowance Not Checked when Updating Orders	Informational	Fixed
QSP-24	Incorrect Restriction in `fromUInt256`	Informational	Fixed
QSP-25	Extremely Expensive Batch Execution for Uniswap V3	Undetermined	Fixed
QSP-26	Issues in Maximum Beacon Reward Calculation	Undetermined	Fixed
QSP-27	Verifier's Dilemma	Undetermined	Acknowledged
QSP-28	Taxed Token Swaps Using Uniswap V3 Might Fail	Undetermined	Acknowledged
QSP-29	Inaccurate Price Simulation	Medium	Fixed

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

DISCLAIMER:
If the final commit hash provided by the client contains features that are not in scope of the audit or a re-audit, those features are excluded from the consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.8.3

Steps taken to run the tools:

1. Declare `OrderBook.orderIdToOrder` as internal since the auto-generated code by the compiler leads to "stack too deep" errors otherwise.
2. Install the Slither tool: `pip3 install slither-analyzer`
3. Run Slither from the project directory: `slither .`

Findings

QSP-1 Stealing User and Contract Funds

Severity: *High Risk*

Status: Fixed

File(s) affected: `LimitOrderRouter.sol`, `SwapRouter.sol`, `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`, `TokenToWethLimitOrderExecution.sol`

Description: Some funds-transferring functions in the contracts are declared as `public` or `external` but without any authorization checks, allowing anyone to arbitrarily call the functions and transfer funds.

1. The visibility of the `safeTransferETH()` function in several contracts is `public`. The visibility allows anyone to call this function to transfer the ETH on the contract to any address directly. The following is the list of affected contracts: `LimitOrderRouter.sol`, `SwapRouter.sol`, `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`, `TokenToWethLimitOrderExecution.sol`.

2. In the `SwapRouter` contract, several `transferXXX()` functions allow anyone to call and direct transfer the funds away. The following is the list of functions: `transferTokensToContract()`, `transferTokensOutToOwner()`, and `transferBeaconReward()`.
3. The `SwapRouter.uniswapV3SwapCallback()` function does not verify that it is called from the Uniswap V3 contract, allowing anyone to steal funds by supplying fake inputs.

Exploit Scenario:

1. Alice wants to place an order selling some of token A. She therefore approves the `SwapRouter` for the amount she wants to sell, since this is required for the `placeOrder()` function to execute successfully.
2. From this point on, an attacker can call `transferTokensToContract()` with the order owner set to the address of Alice and the amount to the approved amount, transferring the tokens to the `SwapRouter`.
3. After that, the attacker calls `transferTokensOutToOwner()` with the order owner set to their address.
4. This can be performed by a smart contract in a single transaction or alternatively in a single call by using the `uniswapV3SwapCallback()` function.

Recommendation:

1. Consider using the `sendValue()` function from OpenZeppelin instead (see: doc) to remove the need of duplicating the code. Otherwise, change the visibility of the `safeTransferETH()` function to `private` or `internal` for all of the listed contracts.
2. Limit that only the execution contracts can call the 'transferXXX()` functions.
3. Limit that only the Uniswap V3 contract can call the function.

Update:

1. The `safeTransferETH()` function visibility was changed to internal for all contracts affected.
2. All `transferXXX()` functions were updated to only be callable by the execution contract.
3. The Uniswapv3 swap callback now verifies that the caller is a Uniswapv3 pool.

QSP-2 Missing Authorization for Execution Contracts

Severity: *High Risk*

Status: Fixed

File(s) affected: `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`, `TokenToWethLimitOrderExecution.sol`

Description: Several functions are missing authorization validation and allow anyone to call the function instead of the specific callers. Specifically, the "execution" contracts are designed to be triggered by the `LimitOrderRouter` contract. However, those functions do not verify the caller. If anyone calls those functions on the "execution" contract, it will trigger the order execution without updating the order status as fulfilled.

Exploit Scenario: The attacker calls the `TokenToTokenLimitOrderExecution.executeTokenToTokenOrders()` directly. The order will be executed but not flagged as fulfilled. Consequently, the attacker can continue to call the function to swap the token of the order's owner.

Recommendation: Add validation that only the `LimitOrderRouter` contract can trigger the following functions:

1. `TaxedTokenLimitOrderExecution.executeTokenToWethTaxedOrders()`
2. `TaxedTokenLimitOrderExecution.executeTokenToTokenTaxedOrders()`
3. `TokenToTokenLimitOrderExecution.executeTokenToTokenOrders()`
4. `TokenToTokenLimitOrderExecution.executeTokenToTokenOrderSingle()`
5. `TokenToWethLimitOrderExecution.executeTokenToWethOrders()`
6. `TokenToWethLimitOrderExecution.executeTokenToWethOrderSingle()`

Update: Execution functions were merged into a single execution contract called `ConveyorExecutor`. Validation was added to each execution function via a modifier called `onlyLimitOrderRouter`.

QSP-3 Ignoring Return Value of ERC20 Transfer Functions

Severity: *High Risk*

Status: Fixed

File(s) affected: `SwapRouter.sol`

Description: Several functions use ERC20's `transfer()` and `transferFrom()` without checking their return values. Since per the specification, these functions merely `SHOULD` throw, some implementations return `false` on error. This is very dangerous, as transfers might not have been executed while the contract code assumes they did.
Affected functions:

- `transferTokensToContract()#L487`
- `transferTokensToOwner()#L504`
- `_swapV2()#L693,696`
- `uniswapV3SwapCallback()#L930,932`

Recommendation: Consider using a library such as OpenZeppelin's SafeERC20 instead of calling the ERC20 functions directly. These will check for return values and also handle tokens that incorrectly implement the standard and do not return a value at all.

Update: The contracts now use the transfer functions provided by `SafeERC20`.

QSP-4 Cancelling Order Provides Compensation Twice

Severity: *High Risk*

Status: Fixed

File(s) affected: `LimitOrderRouter.sol`

Description: After validating that a user does not have sufficient gas credits, the function `validateAndCancelOrder()` first calls `_cancelOrder()`, which removes the order from the system, transfers compensation to the message sender and emits an `OrderCancelled` event. After the call, the function itself sends compensation to the message sender again and emits an `OrderCancelled` event for the second time.

Recommendation: Remove the compensation logic and the emission of events from the `validateAndCancelOrder()` function. Additionally, add tests with at least full line coverage and expected balance checks after the calls to catch obvious errors like this.

Update: Duplicate logic to cancel order compensation has been removed.

QSP-5 Updating an Existing Order Can Be Malicious

Severity: *High Risk*

Status: Fixed

File(s) affected: `OrderBook.sol`

Description: The function `updateOrder()` allows the order owner to change the old order's parameters. From the code, the owner is allowed to change anything except the member `orderID`. We identified the following fields as being problematic:

1. `owner`, the owner can transfer the ownership of the order to any address, including 0x0.
2. `tokenIn`, if `oldOrder.tokenIn != newOrder.tokenIn`, then the code between L246 and L262 makes no sense and will make the system save a wrong `totalOrdersValue`.
3. `lastRefreshTimestamp`, anyone can call `LimitOrderRouter::refreshOrder()` to update `lastRefreshTimestamp` once the block timestamp has exceeded 30 days from original `lastRefreshTimestamp`. If this succeeded, 0.02 ether will be transferred from the order owner's gas balance to the function caller. However, by updating the order, the timestamp can be set to a value where it no longer needs to be refreshed.

Recommendation: Add checks to ensure that only fields that are intended to be updated can be updated. For instance, the `tokenIn` field should not be changeable with the current implementation. If the majority of fields should not be updated, consider modifying the the function's signature to accept an order ID and the new values, e.g. `updateOrder(uint256 orderId, uint256 newQuantity, ...)`.

Update: The signature of the `updateOrder()` function has been changed to only accept a new price and quantity.

QSP-6 Same Order Id Can Be Executed Multiple Times

Severity: *High Risk*

Status: Fixed

File(s) affected: `LimitOrderRouter.sol`

Description: In the current implementation, if the input `orderIds` in the function `executeOrders()` contains duplicate orderIDs, the function will execute the same order more than once.

Recommendation: In `OrderBook._resolveCompletedOrder()`, check if an order exists before deleting it. If it did exist, revert the transaction.

Update: Logic was added within the `_resolveCompletedOrder()` function to check if the order exists in the `orderIdToLimitOrder` mapping. Since the order ID gets removed from this mapping after successful execution, if there is a duplicate order ID in the array of order IDs being executed, the `orderToLimitOrder` mapping will return 0 for the duplicated order ID, causing the transaction to revert.

QSP-7 Incorrectly Computing the Best Price

Severity: *High Risk*

Status: Fixed

File(s) affected: `LimitOrderBatcher.sol`

Description: The function `_findBestTokenToWethExecutionPrice()` initializes the `bestPrice` as `0` for buy orders and `type(uint256).max` for sell orders. For buy orders, the code checks for each execution price whether that price is less than the current `bestPrice` and updates the `bestPrice` and `bestPriceIndex` accordingly. Since there is no "better" price than 0, the function will always return the default value of `bestPriceIndex`, which is 0. Similarly for sell orders, the `bestPrice` is already the best it can be and will always return 0.

Recommendation: Change the initial value of `bestPrice` to `type(uint256).max` for buy orders and `0` for sell orders. This is already done correctly in `_findBestTokenToTokenExecutionPrice()`. Also update the code comments inside of both functions that contain the wrong values (`LimitOrderBatcher.sol#L350,369,469,486`).

Update: The `bestPrice` variable is now initialized correctly, which means the best price is now computed correctly.
Note: The comments on L29 and L48 are outdated now and should be fixed or removed.

QSP-8 Reentrancy

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `LimitOrderRouter.sol`, `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`, `TokenToWethLimitOrderExecution.sol`

Description: A reentrancy vulnerability is a scenario where an attacker can repeatedly call a function from itself, unexpectedly leading to potentially disastrous results. The following are places that are at risk of reentrancy:

1. `LimitOrderRouter.executeOrders()` function: within the function, it calls the "execution" contracts to execute the order(s). The "execution" contracts will swap and

transfer tokens in and out. During the token transfer, it can potentially cause reentrancy if non-standard ERC-20 is integrated (e.g., ERC777). The attack can cause an order to be executed multiple times without the users' consent.

2. `withdrawConveyorFees()` (implemented on multiple contracts): Despite being of lower risk as the function is guarded by `onlyOwner`, a malicious owner can reenter the function and continue to transfer the ETH before `conveyorBalance` is reset to 0. The same function is implemented in multiple contracts: `LimitOrderRouter.sol`, `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution`, and `TokenToWethLimitOrderExecution.sol`. The vulnerability allows the owner(s) of the contracts to retrieve more than the `conveyorBalance` from the contracts.

Exploit Scenario: Here is a sample scenario of attacking one of the listed functions above:

1. Alice placed an order to swap from token A to token B. Token A allows contract hooking on receive, e.g. ERC777's `tokensReceived` hook.
2. Bob also places an order using a malicious contract that can trigger reentrancy for token A.
3. Bob calls `LimitOrderRouter.executeOrders()` with Alice's and Bob's orders.
4. The malicious contract gets triggered after the swap and receives the token. The contract calls `LimitOrderRouter.executeOrders()` again to re-execute the orders.
5. Alice is forced to execute two swaps instead of just the one she placed.

Recommendation: The following are the recommendations matching the order of the listed vulnerable places in the description:

1. Add the `nonReentrant` modifier to the `LimitOrderRouter.executeOrders()` function.
2. For all of the `withdrawConveyorFees()` functions, set the `conveyorBalance` to zero before doing the ETH transfer to follow the checks-effects-interactions pattern or guard the function with the `nonReentrant` modifier.

Update: `nonReentrant` modifiers have been added to the affected functions.

QSP-9 Not Cancelling Order as Expected

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `LimitOrderBatcher.sol`, `SwapRouter.sol`

Description: A few code comments state that orders should be cancelled in certain cases while the implementation does not cancel them:

1. Both the `LimitOrderBatcher.batchTokenToTokenOrders()` and the `LimitOrderBatcher._batchTokenToWethOrders()` functions state that "If the transfer fails, cancel the order" (L124 and L278). However, the implementation does not handle the failed transfer for the call of `IOrderRouter(orderRouter).transferTokensToContract()`. Moreover, since the implementation of the `SwapRouter.transferTokensToContract()` function ignores the return value of `IERC20(order.tokenIn).transferFrom()`, it is possible that the transfer failed silently without reverting the transaction, causing the execution to proceed without a token transfer.
2. `LimitOrderRouter.refreshOrder()` states, "Check that the account has enough gas credits to refresh the order; otherwise, cancel the order and continue the loop". However, on L234-240, the block for the condition `if (gasCreditBalance[order.owner] < REFRESH_FEE)` does not cancel the order.

Recommendation: If the documentation is outdated and the orders should not be cancelled in these scenarios, remove or update the comments in the code. Otherwise, add code to cancel the orders.

Update:

1. The affected code has been removed.
2. The orders are now cancelled as expected.

QSP-10 Granting Insufficient Gas Credit to the Executor

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `LimitOrderRouter.sol`

Description: The `calculateExecutionGasConsumed()` function returns the gas difference of the `initialTxGas` and the current gas retrieved by the `gas()` call. The returned value is the gas without multiplying it with the gas price. The `calculateExecutionGasCompensation()` function uses the returned gas value directly to calculate the `gasDecrementValue`. The `gasDecrementValue` does not take the gas price into account either. Consequently, the executor will not get enough gas compensation with the current implementation.

Recommendation: Consider multiplying the `executionGasConsumed` with the `tx.gasprice` when calculating the `gasDecrementValue` to get the actual gas cost. However, we also recommend checking that the `tx.gasprice` is not significantly more than the `GasOracle.getGasPrice()` value plus a reasonable threshold so that the executor cannot attack the user's gas credit by setting an unrealistically high gas price when executing the order.

Update: The compensation model has been changed to no longer rely on gas prices. An off-chain executor will now receive a flat amount `minExecutionCredit` that can be adjusted by the owner.

QSP-11 Integer Overflow / Underflow

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `OrderBook.sol`, `ConveyorMath.sol`, `SwapRouter.sol`

Related Issue(s): SWC-101

Description: Integer overflow/underflow occurs when an integer hits its bit-size limit. Every integer has a set range; the value loops back around when that range is passed. A clock is a good analogy: at 11:59, the minute hand goes to 0, not 60, because 59 is the most significant possible minute.
We noticed that the `ConveyorMath` library implements changes from the ABDK library and introduced several issues because the overflow protection on the original library would work only on

the signed integers or with 128 bits. The overflow can lead to a miscalculation of the fees and rewards in the `SwapRouter` contract.
The following is the list of places that has the risk of overflow or underflow:

1. `OrderBook._calculateMinGasCredits()#L480-488`: The code uses the `unchecked` block when calculating the `minGasCredits`. However, there is no boundary for the input variables, and the calculation (especially the multiplication on L482-485) can overflow.
2. `ConveyorMath.sub()#L80`: The cast `int128(MAX_64x64)` will always be `-1`. `MAX_64x64` is the constant for `uint128` with 128 bits of 1. After casting to `int128`, the code will treat it as `-1`. Judging from the logic, it does not make sense to check `require(... && result <= -1)`.
3. `ConveyorMath.sub64UI()#L86`: The line `(y << 64)` can be over-shifted and lose some bits for `y`. Also, the calculation `result = x - (y << 64)` can underflow as `y <<64` can be larger than `x`, especially since `x` is of the type `uint128` and `y` is of the type `uint256`. The line `require(result >= 0x0 && uint128(result) <= uint128(MAX_64x64))` would not help preventing the overflow.
4. `ConveyorMath.add128x128()#L100`: The line `answer = x + y` can overflow. The check `require(answer <= MAX_128x128)` would not help as the `answer` will already have overflowed and always passes the validation.
5. `ConveyorMath.add128x64()#L112`: The line `answer = x + (uint256(y) << 64)` can overflow. The check `require(answer <= MAX_128x128)` would not help as the `answer` will already have overflowed and always passes the validation.
6. `ConveyorMath.mul128x64()#L139`: The line `(uint256(y) * x)` can overflow since `x` is of type `uint256`. The check `require(answer <= MAX_128x128)` would not help as the `answer` will already have overflowed and always passes the validation.
7. `ConveyorMath.mul128I()#:177-178`: The line `(uint256(x) * (y & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))` can overflow since `x` is of type `uint256`.
8. `ConveyorMath.div128x128()#L224-225`: The line `hi + lo` can overflow.
9. `ConveyorMath.divUU128x128()#L369-385`: The `require(answer <= MAX_128x128...)` check is useless as `MAX_128x128` is `type(uint256).max` so the `answer` of the `uint256` type can never exceed this value. The following lines `answer * (y >> 128)` and `answer * (y & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)` can potentially overflow since `answer` is not capped. The code is tweaked from the `divUU()` function. However, it cannot simply do so unless accepting phantom overflow, as the original function is designed to work with 128 bits calculations with 256 bits buffer. To handle 256 bits calculation requires a full re-implementation. Also, note that final return value `answer << 128` is incorrect. Since most of the implementation is identical with the `divUU()`, the `answer` is a `64.64` fixed-point number. The implementation should shift the `answer` 64 bits instead of 128 bits to convert a `64.64` value to `128.128` format.
10. `SwapRouter._calculateAlphaX()#L640`: The line `QuadruplePrecision.fromInt(int256(_k))` casts the `_k` from `uint256` to `int256`. If `_k` is larger than or equal to `1<<255`, the cast from `uint256` to `int256` would accidentally treat `_k` as a negative number.

Recommendation:

1. Remove the `unchecked` block to ensure the overflow check is in place for `OrderBook.sol#L480-488`.
2. In `ConveyorMath.sub()`, change `int128(MAX_64x64)` to `type(int128).max` instead.
3. Remove the `unchecked` block as it would not be trivial to add validation to prevent overflow for unsigned integers, in this case for `ConveyorMath.sub64UI()`.
4. Remove the `unchecked` block in `ConveyorMath.add128x128()` and remove the unnecessary validation `require(answer <= MAX_128x128)`.
5. Remove the `unchecked` block in `ConveyorMath.add128x64()` and remove the unnecessary validation `require(answer <= MAX_128x128)`.
6. Remove the `unchecked` block in `ConveyorMath.mul128x64()` and remove the unnecessary validation `require(answer <= MAX_128x128)`.
7. Remove the `unchecked` block in `ConveyorMath.mul128I()` and simply do `(x * y) >> 128` instead of the complex `hi + lo` logic.
8. In the `ConveyorMath.div128x128()` function, change the validation on L224 from `hi + lo <= MAX_128x128` to `hi <= MAX_128x128 - lo` instead.
9. Consider removing this function and use `divUU()` instead of the caller. Alternatively, accept the phantom overflow and simply return `divUU() << 64` for the `divUU128x128()` function.
10. In the `SwapRouter._calculateAlphaX()` function, use `QuadruplePrecision.fromUInt(_k)` instead on L640. `k` should always be a positive number so the code should call `fromUint()` instead of `fromInt()`.

Update: For all issues, the function has either been removed or the recommendation has been implemented.

QSP-12 Updating Order Performs Wrong Total Order Quantity Accounting

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `OrderBook.sol`

Description: When updating an order the total amount of tokens for a particular token are tracked per address in the `totalOrdersQuantity` mapping. The following code performs the adjustments:

```
uint256 totalOrdersValue = _getTotalOrdersValue(oldOrder.tokenIn);

///@notice Update the total orders value
if (newOrder.quantity > oldOrder.quantity) {
    totalOrdersValue += newOrder.quantity - oldOrder.quantity;
} else {
    totalOrdersValue += oldOrder.quantity - newOrder.quantity;
}

///@notice If the wallet does not have a sufficient balance for the updated total orders value, revert.
if (IERC20(newOrder.tokenIn).balanceOf(msg.sender) < totalOrdersValue) {
    revert InsufficientWalletBalance();
}

///@notice Update the total orders quantity
updateTotalOrdersQuantity(
    newOrder.tokenIn,
    msg.sender,
    totalOrdersValue
);
```

The `updateOrders()` function first retrieves the value for the `oldOrder.tokenIn`, then performs adjustments depending on whether the new order has a higher or lower order quantity. The first issue with the function is that the `else` branch of the adjustment code is incorrect. If we assume the old value was 100 and the new value is 50, the total will be updated by `+= 100 - 50`, i.e. an increase by 50 instead of a decrease. While this code path is exercised in a test, the total order value is never checked to be equal to the expected value.
Additionally, the function updates the total order quantity with a call to `updateTotalOrdersQuantity(newOrder.tokenIn, ...)`. Since it is never checked that the `tokenIn` member of the old and new order are the same, this could update some completely unrelated token order quantity.

Recommendation: To fix the first issue, update the `totalOrdersValue` without any conditionals as follows:

```
totalOrdersValue += newOrder.quantity;
totalOrdersValue -= oldOrder.quantity;
```

This will work regardless of the quantity values for the new and old order.
Second, check that the `newOrder.tokenIn` is the same as `oldOrder.tokenIn` at the start of the function.
Finally, make sure that tests check for expected conditions after calling the tested function to avoid these types of bugs.

Update: The order is now updated as suggested in the recommendation.

QSP-13 Not Always Taking Beacon Reward Into Account

Severity: *Low Risk*

Status: Fixed

File(s) affected: `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`

Description: In the `TaxedTokenLimitOrderExecution` and `TokenToTokenLimitOrderExecution` contracts, the `_executeTokenToTokenOrder()` functions will always return a zero amount `beaconReward` when `order.tokenIn != WETH`. Note that the function `_executeSwapTokenToWethOrder()` has the logic for computing `beaconReward` in it but the `beaconReward` value is not returned as part of the function. The `_executeTokenToTokenOrder()` function will need to get the `beaconReward` value from the `_executeSwapTokenToWethOrder()` function.

Recommendation: In both of the `TaxedTokenLimitOrderExecution` and `TokenToTokenLimitOrderExecution` contracts, the `_executeSwapTokenToWethOrder()` should return a `beaconReward` and then the `_executeTokenToTokenOrder()` should use that value as the return value.

Update: The function `_executeSwapTokenToWethOrder()` now returns the `amountOutWeth - (beaconReward + conveyorReward)` after the beacon reward is adjusted in the event that it is capped.

QSP-14 Denial of Service Due to Unbound Iteration

Severity: *Low Risk*

Status: Mitigated

File(s) affected: `OrderBook.sol`

Related Issue(s): SWC-SwapRouter.sol

Description: There is a limit on how much gas a block can execute on the network. It can consume more gas than the network limit when iterating over an unbounded list. In that case, the transaction will never work and block the service. The following is the list of places that are at risk:

1. `OrderBook.getAllOrderIds()` (L515-556): The function loops over the value of `addressToAllOrderIds[owner]`. However, the `addressToAllOrderIds` will keep appending without removal or a size limit. Thus, the loop in the L533-549 can fail. The impact of this function failing is unclear as it is not used within the contract. It seems like the off-chain component will fail when calling this.
2. `SwapRouter.getAllPrices()` (L1200-1265): The function loops through all of the `dexes` set in the `constructor()`. This will risk failing the several "execute" functions in the `TaxedTokenLimitOrderExecution`, the `TokenToTokenLimitOrderExecution`, and the `TokenToWethLimitOrderExecution` contract.

Recommendation:

1. For the function `OrderBook.getAllOrderIds()`, consider recording this off-chain instead and remove the function. The off-chain component can monitor the events and update its storage instead of tracking all elements in an array on the chain.
2. Consider validating that the size of the `dexes` is less than a practical cap in the `SwapRouter.constructor()`.

Update:

1. The function `OrderBook.getAllOrderIds()` has been renamed to `getOrderIds()`. This function now accepts an offset and a length to enable pagination.
2. The team assured us that the number of entries in the `dexes` array will be small enough to not be an issue.

QSP-15 Missing Input Validation

Severity: *Low Risk*

Status: Fixed

File(s) affected: `OrderBook.sol`

Description: It is essential to validate inputs to avoid human error, even if they only come from trusted addresses. The following is the list of places that can benefit from adding extra validations:

1. `OrderBook._resolveCompletedOrder()` (L396-410): Check if the order exists or not before removing from storage. This function is used in `LimitOrderRouter.executeOrders()` (L432-532). However, this function does not check the order existence either.
2. `LimitOrderRouter.transferOwnership()` (L566-571): The validation on L567 `owner == address(0)` seems to be mistaken. Should validate that the `newOwner` is a non-zero address instead of validating the `owner`.
3. `LimitOrderRouter.executeOrders()` (L432-532): Check if the order exists for the order ID in the `orderIds` array. The validation can be added in the loop at L440-446 that sets `orders[i] = getOrderById(orderIds[i])`. Otherwise, an empty order has a chance to be executed unexpectedly. Additionally, the function should check that `orderIds.length` is non-zero.
4. `SwapRouter.constructor()` (L190-208): The first element of the `_isUniV2` input must be `true` as the 0th index DEX needs to be Uniswap V2 compatible due to the implementation of the `SwapRouter._calculateFee()` function.
5. `GasOracle.constructor()` does not check if `_gasOracleAddress` is the zero address.
6. `OrderBook.constructor()` does not check if `_orderRouter` is the zero address.
7. `LimitOrderRouter.constructor()` does not check if any of `_weth`, `_usdc`, `_tokenToTokenExecutionAddress`, `_taxedExecutionAddress`, `_tokenToWethExecutionAddress`, `_orderRouter` are the zero address, and `_executionCost` is not 0.

8. `LimitOrderRouter.depositGasCredits()` does not check if `msg.value` is 0. If `msg.value == 0`, then the function does nothing.
9. `SwapRouter.constructor()` does not check if any element in `_deploymentByteCodes` or `_dexFactories` is 0.
10. `SwapRouter.calculateFee()` does not check if `amountIn` is not 0.
11. `LimitOrderBatcher.constructor()` does not check that `weth`, `quoterAddress`, `orderRouterAddress` are not the zero address.
12. `TokenToTokenLimitOrderExecution.constructor()`, `TaxedTokenLimitOrderExecution.constructor()`, `TokenToWethLimitOrderExecution.constructor()` do not check that `_usdc` is not 0.

Recommendation: Add validation checks as laid out in the description section.

Update: Either input validation has been added as recommended or the affected code has been removed. Note that the fix for entry number seven relies on the contract being deployed by the `ConveyorExecutor` contract that performs input validation on the relevant arguments in its constructor.

QSP-16 Gas Oracle Reliability

Severity: *Low Risk*

Status: Fixed

File(s) affected: `GasOracle.sol`, `LimitOrderRouter.sol`

Description: There are two potential concerns about the `GasOracle` contract implementation:

1. Chainlink updates the feeds periodically (heartbeat idle time). The application should check that the timestamp of the latest answer is updated within the latest heartbeat or within the time limits acceptable for the application (see: Chainlink docs). The `GasOracle.getGasPrice()` function does not check the timestamp of the `answer` from the `IAggregatorV3(gasOracleAddress).latestRoundData()` call.
2. The "Fast Gas Data Feed" prices from Chainlink can be manipulated according to their docs. The application should be designed to detect gas price volatility or malicious activity.

Exploit Scenario: The Chainlink gas price feed is either out of sync or manipulated. The Conveyor contracts will use the wrong gas price to validate user orders. For instance, if the gas price is manipulated to be larger than usual, anyone can call the `LimitOrderRouter.refreshOrder()` function to cancel user orders, as it will not pass the check of `_hasMinGasCredits()` on L259.

Recommendation:

1. Check that the returned timestamp from the `IAggregatorV3(gasOracleAddress).latestRoundData()` call is recent enough.
2. Consider adding gas price volatility detection on the contract. For instance, compare the new gas price with a time-weighted average gas price from the previous monitored price and check that the difference is within an acceptable threshold.

Update: The design has been changed and the code no longer uses a gas oracle.

QSP-17 Math Function Returns Wrong Type

Severity: *Low Risk*

Status: Fixed

File(s) affected: `ConveyorMath.sol`

Description: Under the assumption that the function `divUU128x128()` should return a `128.128` fixed point number, this function does not return the correct value.

Exploit Scenario: The call `divUU128x128(3,2)` will return `0x18000000000000000.00000000000000000000000000000000` (dot after 128 bits), which is incorrect. The result is a `64.192` fixed point number.

Recommendation: Fix the implementation. Since the function does not seem to be used anywhere, we classified the issue as `low`.

Update: The affected function has been removed.

QSP-18 Individual Order Fee Is Not Used in Batch Execution

Severity: *Low Risk*

Status: Fixed

File(s) affected: `TokenToWethLimitOrderExecution.sol`

Description: In `TokenToWethLimitOrderExecution.sol#L365`, `getAllPrices()` is using the first order's `order.feeIn` to compute uniswap prices for all of the orders in the batch. However, different orders are not guaranteed to have the same `feeIn`. Thus, the computed result may not apply to all orders in the batch.

Recommendation: Change the logic to use each individual order's `feeIn` or check that they all have the same `feeIn` value, e.g. in `LimitOrderRouter._validateOrderSequencing()`.

Update: The `LimitOrderRouter._validateOrderSequencing()` function now verifies that all orders are using the same `feeIn`.

QSP-19 Locking the Difference Between `beaconReward` and `maxBeaconReward` in the Contract

Severity: *Informational*

Status: Fixed

File(s) affected: `TaxedTokenLimitOrderExecution.sol`, `TokenToTokenLimitOrderExecution.sol`, `TokenToWethLimitOrderExecution.sol`

Description: The code has the concept of a `maxBeaconReward` to cap the max beacon reward sent to the executor. So whenever the raw `beaconReward` is greater than the `maxBeaconReward`, the executor will get the `maxBeaconReward`. However, the implementation will lock the difference between the two in the contract.

1. `TaxedTokenLimitOrderExecution._executeTokenToWethTaxedSingle()`: The function calls the `_executeTokenToWethOrder()` function on L133 and the `_executeTokenToWethOrder()` function will return `uint256(amountOutWeth - (beaconReward + conveyorReward))` (L192) as the `amountOut`. The `amountOut` is the final amount transferred to the order's owner. Later on L148-150, the raw `beaconReward` is capped to the `maxBeaconReward`. The difference will be left and locked in the contract.
2. `TaxedTokenLimitOrderExecution._executeTokenToTokenTaxedSingle()`: The function calls `_executeTokenToTokenOrder()`. The raw `beaconReward` will also be part of the deduction in `_executeTokenToTokenOrder()` (L357): `amountInWethToB = amountIn - (beaconReward + conveyorReward)`. The `maxBeaconReward` cap is applied later, and the difference will be left locked in the contract.
3. `TokenToTokenLimitOrderExecution._executeTokenToTokenSingle()`: The function calls the `_executeTokenToTokenOrder()`. The raw `beaconReward` will be deducted when calculating the `amountInWethToB`. Later in `_executeTokenToTokenSingle()`, the `maxBeaconReward` is applied to the `beaconReward`. The difference between the `beaconReward` and `maxBeaconReward` will be left in the contract.
4. `TokenToTokenLimitOrderExecution._executeTokenToTokenBatchOrders()`: The function calls `_executeTokenToTokenBatch()`. In the `_executeTokenToTokenBatch()` function, the raw `beaconReward` is deducted when calculating the `amountInWethToB`. Later in the `_executeTokenToTokenBatchOrders()` function, the `maxBeaconReward` is applied to the `totalBeaconReward`. The difference between the `totalBeaconReward` and `maxBeaconReward` will be left in the contract.
5. `TokenToWethLimitOrderExecution._executeTokenToWethSingle()`: The function calls `_executeTokenToWethOrder()`. The `_executeTokenToWethOrder()` will deduct the raw `beaconReward` when returning the `amountOut` value. Later, the `_executeTokenToWethSingle()` function caps the `beaconReward` to `maxBeaconReward`. The difference between the `beaconReward` and `maxBeaconReward` will be left in the contract.
6. `TokenToWethLimitOrderExecution._executeTokenToWethBatchOrders()`: The function calls the `_executeTokenToWethBatch()` function. The `_executeTokenToWethBatch()` will deduct the raw `beaconReward` when returning the `amountOut` value. Later, the `_executeTokenToWethBatchOrders()` function caps the `totalBeaconReward` to `maxBeaconReward`. The difference between the `totalBeaconReward` and `maxBeaconReward` will be left in the contract.

Recommendation: We recommend capping the raw `beaconReward` to the `maxBeaconReward` before deducting it from the swap output amount. In other words, the part exceeding the `maxBeaconReward` will be transferred to the users as part of the swap output instead of having it locked in the contract.

Update: The reward cap is now computed before subtracting it from the output amount.

QSP-20 Inaccurate Array Length

Severity: *Informational*

Status: Fixed

File(s) affected: `LimitOrderBatcher.sol`, `OrderBook.sol`

Description: Some functions return arrays that are padded with empty elements. The caller of those functions will need to be aware of this fact to not accidentally treat the padding as real data. The following is a list of functions that have this issue:

1. `OrderBook.getAllOrderIds()`: The impact is unclear, as the function is only used in the test contracts.
2. `LimitOrderBatcher.batchTokenToTokenOrders()`: The function is called by `TokenToTokenLimitOrderExecution.executeTokenToTokenOrders()`. Fortunately, the implementation of `executeTokenToTokenOrders()` seems to be aware of the fact that batches can be empty.

Recommendation: Either get an exact array length and allocate the array with the correct size or try to override the array length before returning the array. Otherwise, consider adding a warning to the above functions to ensure callers are aware of the returned array potentially containing empty elements.
While newer solidity versions no longer allow assigning the array length directly, it is still possible to do so using assembly:

```
assembly {
    mstore(<:your_array_var>, <:reset_size>)
}
```

Update:

1. The array no longer contains empty entries since it is now truncated using assembly.
2. The affected code has been removed.

QSP-21 `TaxedTokenLimitOrderExecution` Contains Code for Handling Non-Taxed Orders

Severity: *Informational*

Status: Fixed

File(s) affected: `TaxedTokenLimitOrderExecution.sol`

Description: The function `_executeTokenToTokenOrder()` checks whether the order to execute is *taxed*. In case it is not, the tokens for the order are transferred to the `SwapRouter` contract. When actually executing the swap in `_executeSwapTokenToWethOrder()`, the swap is performed using the `order.owner` as sender, i.e. the tokens will be sent again from that address.
Since the function is typically only called from code paths where orders are taxed, the best case scenario is that `TaxedTokenLimitOrderExecution.sol#L323-326` is dead code. In case somebody calls this function manually with a non-taxed order, it might lead to tokens being sent erroneously to the `SwapRouter`.

Recommendation: Remove the code at `TaxedTokenLimitOrderExecution.sol#L323-326`.

Update: Taxed token limit order execution has been removed. Instead of a specific function for taxed token execution, taxed tokens are now executed through the functions `executeTokenToTokenOrders()` or `executeTokenToWethOrders()`.

QSP-22 Unlocked Pragma

Severity: *Informational*

Status: Fixed

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (^) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend removing the caret to lock the files to a specific Solidity version. Consult the slither documentation for recommended versions of solidity.

Update: All core contracts now use `pragma solidity 0.8.16`.

QSP-23 Allowance Not Checked when Updating Orders

Severity: *Informational*

Status: Fixed

File(s) affected: `OrderBook.sol`

Description: When placing an order, the contract will check if users set a high enough allowance to the `ORDER_ROUTER` contract (L216). However, this is not checked when updating an order in the function `updateOrder()`.

Recommendation: Add an allowance check in `updateOrder()` for the new order quantity.

Update: The allowance is now also checked when updating an order.

QSP-24 Incorrect Restriction in `fromUInt256`

Severity: *Informational*

Status: Fixed

Description: In the function `fromUInt256()`, if the input `x` is an unsigned integer and `x <= 0xFFFFFFFFFFFFFFFF`, then after `x << 64`, it will be less than or equal to `MAX64.64`. However the restriction for `x` is set to `<= 0x7FFFFFFFFFFFFFFF` in the current implementation.

Recommendation: Change the restriction to `x <= 0xFFFFFFFFFFFFFFFF` in `ConveyorMath.sol#L20`.

Update: The require statement has been updated to check for the correct value.

QSP-25 Extremely Expensive Batch Execution for Uniswap V3

Severity: *Undetermined*

Status: Fixed

File(s) affected: `LimitOrderBatcher.sol`, `SwapRouter.sol`

Description: According to the discussion with the team, the motivation for executing orders in batches is to save gas by reducing the number of swaps. However, when it comes to Uniswap V3, the code heavily relies on the `QUOTER.quoteExactInputSingle()` call to get the output amount of the swap. Unfortunately, the `QUOTER.quoteExactInputSingle()` is as costly as a real swap because it is performing the swap and then force reverting to undo it (see: code and doc). `QUOTER.quoteExactInputSingle()` is called in `SwapRouter.getNextSqrtPriceV3()`, `LimitOrderBatcher.calculateNextSqrtPriceX96()`, and `LimitOrderBatcher.calculateAmountOutMinAToWeth()`.

Exploit Scenario: In the worst case for batch executions, it can be four times the swap gas cost.
First, `LimitOrderBatcher._simulateAToBPriceChange()` calls `QUOTER.quoteExactInputSingle()` through `calculateNextSqrtPriceX96()`. Later, the function `TokenToTokenLimitOrderExecution._executeSwapTokenToWethOrder()` will call it again. Then, `SwapRouter._swapV3()` will call `getNextSqrtPriceV3()` and call it once more. Finally, the real swap of `IUniswapV3Pool(_lp).swap()` will perform the fourth and final swap.

Recommendation: We recommend performing some analysis to understand the gas cost and performance of the batching code. If it turns out that the savings are relatively minimal or it is even more expensive to perform batching, consider launching without batching for the first version. This might also lead to a safer overall implementation, since the batching code is fairly complex and has the potential to contain hard to find bugs.

Update: As per the recommendation, the batching functionality has been removed. This greatly simplified the execution logic and lowered the overall gas consumption. Additionally, price calculation is now done by simulating a swap instead of using the Uniswap V3 Quoter contract.

QSP-26 Issues in Maximum Beacon Reward Calculation

Severity: *Undetermined*

Status: Fixed

File(s) affected: `SwapRouter.sol`

Description: From the discussion with the team, the team informed us that the idea of the max beacon reward is to prevent flash loan attacks. The max beacon reward is designed so that when the price diverges too much, the beacon executor cannot earn more beacon rewards than the cost to manipulate the price on the Uniswap pools. The main logic for this is in the `SwapRouter.calculateMaxBeaconReward()` function. However, we found some potential issues and some points are unclear to us:

1. On L414-417, the `if (buy && v2Outlier > v3Spot) {return MAX_UINT_128;}` block does not check if `v3PairExists`. If the V3 pair does not exist, `v3Spot` will be zero, and the `v2Outlier` will be greater than the `v3Spot`. Thus, if the v3 pair does not exist, the buy order will not be capped by the max beacon reward.
2. The implementation assumes that there can be at most one Uniswap V3 address in the `dexes` array. On L382-390, the `if (!dexes[i].isUniV2)` block overrides the `v3Spot` and `v3PairExists` variables without checking if those were set already or not. If more than one Uniswap V3 contract is integrated, the implementation will always take the data from the last V3 contract element.
3. L442 hardcodes the `UNI_V2_FEE` as the input when calling `_calculateMaxBeaconReward()` for V3. The `v3PairExists` block is supposed to find the `priceDivergence` between V3 and V2. However, Uniswap V3 allows fees of `0.05%`. If the fee rate is lower on V3, the cost to manipulate the pool price is also lower. In such a case, the `maxBeaconReward` should also be lower.
4. It is unclear why the comparison points for V3 and V2 are different when calculating the `priceDivergence`. For V3, it compares with the V2 (outlier) price, while for V2, it compares with the prices in the orders. It is unclear to us why V2 and V3 are handled differently.

Recommendation:

1. Change the condition to `buy && v3PairExists && v2Outlier > v3Spot`.

2. Clarify whether it is intended that the code can support only one V3 pool for a token pair. If that is the case, add validation in to `SwapRouter.constructor()`, to ensure that at most one V3 contract is added.
3. Try to get the real fee rate for V3 instead of the hardcoded ones. Alternatively, change the hardcoded fee to `0.05` instead of `0.3` for the `UNI_V2_FEE`.
4. Clarify the intention and make the changes accordingly.

Update: The team decided to remove the `alphaX` and `maxBeaconReward` functions from the contract because of the gas consumed from the computation. They also decided that capping the reward was not necessary for any limit order other than a stop loss, which has been implemented using a constant.

QSP-27 Verifier's Dilemma

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `LimitOrderRouter.sol`

Description: The beacon network allows any executor to submit the execution of the orders. As an incentive for monitoring the protocol and executing orders once they become valid, the protocol pays the executors a reward. However, an attacker can steal the work of another executor by simply monitoring the mempool and front-run their transaction. Despite the attacker never monitoring and matching the orders, the attacker can earn the beacon reward by stealing another executor's work.
Furthermore, since only the fastest executor receives a reward, this might discourage participants from becoming an executor themselves.

Recommendation: We do not have a specific recommendation as it requires complex design changes to mitigate or solve the issue. However, we recommend the team keep this in mind and monitor this issue. Also, the team should research potential mitigations to solve this issue in the future. For instance, the team might apply some techniques to mitigate the risk and prevent front-running, such as committing and receiving.

Update: The team acknowledge the issue and argue that the impact is likely to be relatively low in practice with the following explanation:

> The default behavior for the off-chain logic will be to route the execution transaction through private relays, encrypted mempools and other means of transaction obfuscation when available. For chains that do not have an equivalent option, the default Beacon will have built in priority gas auction (PGA) logic to compete with other bots that try to front run their transaction. While this does not alleviate the Verifier's dilemma on these chains, it gives the beacon a defense mechanism against front runners.

QSP-28 Taxed Token Swaps Using Uniswap V3 Might Fail

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `SwapRouter.sol`

Description: The `SwapRouter` does not seem to handle taxed tokens, i.e. ones with a "fee on transfer", correctly when routed through Uniswap V3. In the `uniswapV3SwapCallback()` function, one of the function parameters dictates the amount of tokens that have to be transferred to the pool for the swap to succeed. The `SwapRouter` attempts to send exactly that many tokens. However, if the token transfers are "taxed", the pool will receive less than intended and the swap will fail.

Recommendation:

1. Verify that the taxed swap behavior is the same as described above by adding tests for that scenario.
2. Either modify the code dealing with Uniswap V3 swaps to be able to handle taxed tokens or blacklist taxed tokens from being swapped through Uniswap V3.

Update: The team have added tests that ensure taxed tokens can be swapped as expected.

QSP-29 Inaccurate Price Simulation

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `ConveyorTickMath.sol`

Description: The Conveyor Finance team found an issue that was not discovered in the first fix review. The function `simulateAmountOutOnSqrtPriceX96()`, used for calculating the change in price after a token swap on Uniswap V3, sometimes returned inaccurate results. The issue stemmed from passing local mappings to uniswap's internal library functions, instead of using the information from the pool that swap is executed on. More specifically, the functions `nextInitializedTickWithinOneWord()`and `cross()`operate on local mappings and are not querying the pool's storage for this information.

Update: Since it is impossible to pass entire mappings of an external contract to a function, the team has changed the API of the internal uniswap libraries to accept a pool address instead. Furthermore, the functions have been adjusted to query the pool's storage for the tick and tick bitmap information.

Automated Analyses

Slither

Slither reported 680 results, many of which were either false positives or classified as not being an issue. The relevant results have been integrated with the findings of this report.

Adherence to Specification

1. In the section `Fee Structure` of `internal.conveyorlabs.whitepaper`, "The base free shrinks" should be "The base fee shrinks".

Code Documentation

1. **Fixed:** Consider providing instructions on how to build and test the contracts in the README.
2. **Fixed:** Consider providing a link in the code comment for the `SwapRouter._getV2PairAddress()` function (L1025-1045) on how the address is determined: Uniswap V2 Pair Address doc.
3. **Fixed:** The comment in `LimitOrderRouter.sol#L416` (within the `_validateOrderSequencing()` function) does not match the implementation. Change it from "Check if the token tax status is the same..." to "Check if the buy/sell status is the same..." instead.
4. **Fixed:** The code documentation/comment in `LimitOrderBatcher.sol#L22` and `LimitOrderBatcher.sol#L35` for the `batchTokenToTokenOrders()` function seems inconsistent with the implementation. The comment states "Function to batch multiple token to weth orders together" and "Create a new token to weth batch order", but the function is token to "token" and not token to "weth".
5. **Partially fixed:** `LimitOrderBatcher.sol#L469` states, "If the order is a buy order, set the initial best price at 0". However, the implementation set the initial best price to the max of `uint256`. Similarly, L486 states, "If the order is a sell order, set the initial best price at max uint256". In contrast, the implementation sets the initial price to zero. The implementation seems correct, and the comment is misleading.
6. **Fixed:** The code documentation for the following contracts seems misplaced: `TaxedTokenLimitOrderExecution`, `TokenToTokenLimitOrderExecution`, and `TokenToWethLimitOrderExecution`. They all have `@title SwapRouter` instead of each contract's documentation.
7. **Fixed:** Fix the code documentation for the `ConveyorMath.add64x64()` function. L65 states that "helper to add two unsigened 128.128 fixed point numbers" while the functions add two 64.64 fixed point numbers instead. Also, there is a typo on the word "unsigened", which should be "unsigned".
8. **Fixed:** Consider adding NatSpec documentation for the following functions in `ConveyorMath.sol`: `sub()`, `sub64UI()`, `abs()`, `sqrt128()`, `sqrt()`, and `sqrtBig()`. It is unclear which types they operate on (e.g., whether they should be fixed-point numbers).
9. **Fixed:** Fix the code documentation for the `ConveyorMath.mul128x64()` function. L130 states that "helper function to multiply two unsigned `64.64` fixed point numbers" while multiplying a `128.128` fixed point number with another `64.64` fixed-point number.
10. **Fixed:** Add `@param` comment for the field `taxIn` of the struct `Order` (L44-73) in `OrderBook.sol`.
11. **Fixed:** Consider adding a warning for the `SwapRouter.calculateFee()` function that the `amountIn` can only be the amount WETH (or 18 decimal tokens).
12. **Fixed:** The `onlyOwner` modifier implemented in the `*LimitOrderExecution.sol` contracts has documentation that states that the modifier should be applied to the function `transferOwnership()`. As there is no `transferOwnership()` function in those contracts, either add one or remove it from the modifier documentation.
13. **Fixed:** `ConveyorMath.mul128I()#L167`, "multiply unsigned 64.64" should be "128.128".
14. **Fixed:** `ConveyorMath.div128x128()#L213`, "@return unsigned uint128 64.64" should be "128.128".
15. **Fixed:** `ConveyorMath.divUI()#L229`, "helper function to divide two 64.64 fixed point numbers" should be "... two integers".
16. **Fixed:** `ConveyorMath.divUI128x128()#L310`, "helper function to divide two unsigned 64.64 fixed point" should be "... two integers".
17. `ConveyorMath.divUI128x128()#L313`, "@return unsigned uint128 64.64 unsigned integer" should be "... uint256 128.128 fixed point number".
18. **Fixed:** `ConveyorMath.divUU128x128()#L330`, "@return unsigned 64.64" should be "... 128.128".
19. **Fixed:** `TokenToWethLimitOrderExecution.sol#L349`, the documentation is wrong, since the function only handles tokenA -> Weth.
20. **Fixed:** `TaxedTokenLimitOrderExecution.sol#L197`, the documentation is wrong, since the function only handles tokenA -> Weth.
21. **Fixed:** The following functions do not have any documentation:
    1. `ConveyorTickMath.fromX96()`
    2. `ConveyorMath.sub()`
    3. `ConveyorMath.sub64UI()`
    4. `ConveyorMath.sqrt128()`
    5. `ConveyorMath.sqrt()`
    6. `ConveyorMath.sqrtBig()`
    7. `QuadruplePrecision.to128x128()`
    8. `QuadruplePrecision.fromInt()`
    9. `QuadruplePrecision.toUInt()`
    10. `QuadruplePrecision.from64x64()`
    11. `QuadruplePrecision.to64x64()`
    12. `QuadruplePrecision.fromUInt()`
    13. `QuadruplePrecision.from128x128()`
22. The `@return` documentation for the following functions is unclear:
    1. `ConveyorMath.mul64x64()` (expecting unsigned 64.64).
    2. `ConveyorMath.mul128x64()` (expecting unsigned 128.128).
    3. `ConveyorMath.mul64I()` (expecting unsigned integer).
    4. `ConveyorMath.mul128I()` (expecting unsigned integer).

Adherence to Best Practices

1. **Fixed:** Remove the unused function `OrderBook._resolveCompletedOrderAndEmitOrderFufilled()` (L371-392).
2. **Fixed:** Remove the unused function `OrderBook.incrementTotalOrdersQuantity()` (L441-448).
3. **Fixed:** `OrderBook.sol#L487`, replace the magic number `100` in the `_calculateMinGasCredits()` function with a named constant.
4. **Fixed:** `OrderBook.sol#L505`, replace the magic number `150` in the `_hasMinGasCredits()` function with a named constant.
5. **Fixed:** Consider setting the `tempOwner` to zero in the `LimitOrderRouter.confirmTransferOwnership()` function once the `owner` is set. By cleaning up the storage, the EVM will refund some gas.
6. **Fixed:** Consider replacing the assembly block with simply `initalTxGas = gasleft()` in `LimitOrderRouter.sol#434-436` (within the `executeOrders()` function). The gas saved with the assembly is negligible (around 10).
7. **Fixed:** Consider removing the `LimitOrderBatcher._buyOrSell()` function. The code using this function can replace it simply with `firstOrder.buy` on L44 and L207.
8. **Fixed:** Consider renaming the `ConveyorMath.mul64I()` (L149) and the `ConveyorMath.mul128I()` (L171) functions to `mul64U()` and `mul128U()` instead. The functions

handle unsigned integers instead of signed integers.

9. **Fixed:** `GasOracle.getGasPrice()` tends to get called multiple times per execution. Consider whether it's possible to cache it to avoid multiple external calls.
10. **Fixed:** `OrderBook.addressToOrderIds` seems unnecessary. It is used to check whether orders exist via: `bool orderExists = addressToOrderIds[msg.sender][newOrder.orderId];`. This can also be done through `bool orderExists = orderIdToOrder[newOrder.orderId].owner == msg.sender`.
11. **Fixed:** `OrderBook.orderIdToOrder` should be declared as `internal` since the generated getter function leads to "stack too deep" errors when compiled without optimizations, which is required for collecting code coverage.
12. Consider using `orderNonce` as the `orderId` directly instead of hashing it with `block.timestamp`, since the `orderNonce` will already be unique.
13. **Fixed:** `OrderBook.sol#L177` and `LimitOrderRouter.sol#L285` perform a modulo operation on the `block.timestamp` and cast the result to `uint32`. A cast to `uint32` will truncate the value the same way the modulo operation does, which is therefore redundant and can be removed.
14. **Fixed:** In `OrderBook.placeOrder()`, the local variables `orderIdIndex` and `i` will always have the same value. `orderIdIndex` can be removed and uses replaced by `i`.
15. **Fixed:** Consider removing the `OrderBook.cancelOrder()` function, since the `OrderBook.cancelOrders()` contains nearly identical code. Additionally, to place an order, only the `OrderBook.placeOrders()` function exists, which makes the API inconsistent.
16. **Fixed:** `LimitOrderRouter.refreshOrder()#L254` calls `getGasPrice()` in each loop iteration. Since the gas price does not change within a transaction, move this call out of the loop to save gas.
17. **Fixed:** `LimitOrderRouter.refreshOrder()#L277` sends the fees to the message sender on each loop. Consider accumulating the amount and use a single `safeTransferETH()` call at the end of the function.
18. `SwapRouter.sol` should implement the `IOrderRouter` interface explicitly to ensure the function signatures match.
19. **Fixed:** `SwapRouter._calculateV2SpotPrice()#L961` computes the Uniswap V2 token pair address manually and enforces that it is equal to the `IUniswapV2Factory.getPair()` immediately after. Since the addresses must match, consider using just the output of the call to `getPair()` and remove the manual address computation. The `getPair()` function returns the zero address in case the pair has not been created.
20. **Fixed:** `SwapRouter._swapV3()` makes a call to `getNextSqrtPriceV3()` to receive the `_sqrtPriceLimitX96` parameter that is passed to the pool's `swap()` function. Since the `getNextSqrtPriceV3()` function potentially also performs the expensive swap through the use of a `Quoter.quoteExactInputSingle()` call and the output amount of the swap will be checked by `uniswapV3SwapCallback()` anyway, consider using the approach of Uniswap V3's router and supply `(_zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.MAX_SQRT_RATIO - 1)` as the `_sqrtPriceLimitX96` parameter.
    This would also allow removing the dependency on the `Quoter` contract for the `SwapRouter` contract.
21. **Fixed:** Save the `uniV3AmountOut` amount in memory and set the contract storage back to 0 before returning the amount in `SwapRouter._swapV3()#L829` to save gas (see EIP-1283).
22. **Fixed:** The `iQuoter` member should be removed from the `*LimitOrderExecution.sol` contracts, since they are not used by the contracts and already inherited through `LimitOrderBatcher`.
23. **Fixed:** `ConveyorMath.mul64x64#L125` uses an unclear require message that looks like a leftover from debugging.
24. **Fixed:** `SwapRouter.calculateReward()#L320`, change `(0.005-fee)/2+0.001*10**2` to `((0.005-fee)/2+0.001)*10**2` to avoid confusion about operator precedence.
25. **Fixed:** `ConveyorMath.divUI()`and `ConveyorMath.divUU()` perform the same computation. Remove `divUI()`.
26. **Fixed:** `ConveyorMath.divUI128x128()` and `ConveyorMath.divUU128x128()` perform the same computation. Remove `divUI128x128()`.
27. **Fixed:** The function `mostSignificantBit()` exists in both `ConveyorBitMath.sol` and `QuadruplePrecision.sol`. Remove one of them.
28. **Partially fixed:** Typos in variables:
    1. Several variables contain the typo `fufilled` instead of `fulfilled`. Some of these are externally visible.
    2. parameter `_reciever` in `SwapRouter._swapV2()` should be renamed to `_receiver`, the return variable `amountRecieved` should be `amountReceived`
    3. parameter `_reciever` in `SwapRouter._swapV3()` should be renamed to `_receiver`, the return variable `amountRecieved` should be `amountReceived`
    4. parameter `_reciever` in `SwapRouter._swap()` should be renamed to `_receiver`, the return variable `amountRecieved` should be `amountReceived`.
29. `OrderBook.sol#L240` could use `storage` instead of `memory` to save gas.
30. **Fixed:** The internal function `_executeSwapTokenToWethOrder()` in `TokenToWethLimitOrderExecution.sol` is never used and can be removed.

Test Results

Test Suite Results

The data has been collected using the command `forge test --fork-url $ETH_RPC_URL --ffi --fork-block-number 16225780`.

```
Running 21 tests for src/test/LimitOrderExecutor.t.sol:LimitOrderExecutorTest
[PASS] testExecuteTaxedTokenToTaxedTokenBatch() (gas: 696229)
[PASS] testExecuteTaxedTokenToTaxedTokenSingle() (gas: 723096)
[PASS] testExecuteTaxedTokenToTokenBatch() (gas: 962656)
[PASS] testExecuteTaxedTokenToTokenSingle() (gas: 719790)
[PASS] testExecuteTaxedTokenToWethBatch() (gas: 1385417)
[PASS] testExecuteTaxedTokenToWethSingle() (gas: 641386)
[PASS] testExecuteTokenToTaxedTokenSingle() (gas: 742160)
[PASS] testExecuteTokenToTokenBatch() (gas: 1120104)
[PASS] testExecuteTokenToTokenSingle(uint80) (runs: 256, μ: 365, ~: 367)
[PASS] testExecuteTokenToWethOrderBatch() (gas: 1553647)
[PASS] testExecuteTokenToWethSingle(uint112) (runs: 256, μ: 474, ~: 476)
[PASS] testExecuteWethToTaxedTokenBatch() (gas: 1044644)
[PASS] testExecuteWethToTaxedTokenSingle(uint112) (runs: 256, μ: 385, ~: 387)
[PASS] testExecuteWethToTokenOrderBatch() (gas: 1740132)
[PASS] testExecuteWethToTokenSingle() (gas: 538539)
[PASS] testFailExecuteOrders_InvalidCalldata() (gas: 33158)
[PASS] testFailExecuteOrders_OrderDoesNotExist() (gas: 49937)
[PASS] testFailExecuteTokenToTokenBatch_DuplicateOrdersInExecution() (gas: 1568478)
[PASS] testFailExecuteTokenToTokenBatch_InvalidNonEOAStoplossExecution() (gas: 740805)
[PASS] testFailExecuteTokenToWethOrderBatch_DuplicateOrdersInExecution() (gas: 1171867)
[PASS] testFailExecuteTokenToWethOrderBatch_InvalidNonEOAStoplossExecution() (gas: 357312)
Test result: ok. 21 passed; 0 failed; finished in 384.34ms

Running 19 tests for src/test/OrderBook.t.sol:OrderBookTest
[PASS] testCancelOrder() (gas: 399803)
[PASS] testCancelOrders() (gas: 589840)
[PASS] testDecreaseExecutionCredit(uint128,uint128,uint128) (runs: 256, μ: 417302, ~: 486271)
[PASS] testFailCancelOrder_OrderDoesNotExist(bytes32) (runs: 256, μ: 20994, ~: 20994)
[PASS] testFailCancelOrders_OrderDoesNotExist(bytes32,bytes32) (runs: 256, μ: 53874, ~: 53874)
[PASS] testFailGetLimitOrderById_OrderDoesNotExist() (gas: 21262)
[PASS] testFailPlaceOrder_IncongruentTokenInOrderGroup(uint256,uint256,uint256,uint256) (runs: 256, μ: 377797, ~: 591718)
```

```
[PASS] testFailPlaceOrder_InsufficientAllowanceForOrderPlacement(uint256,uint256) (runs: 256, μ: 314435, ~: 450294)
[PASS] testFailPlaceOrder_InsufficientExecutionCredit() (gas: 44409)
[PASS] testFailPlaceOrder_InsufficientWalletBalance() (gas: 64105)
[PASS] testFailUpdateOrder_InsufficientAllowanceForOrderUpdate(uint128,uint64,uint128,uint128) (runs: 256, μ: 467983, ~: 483594)
[PASS] testFailUpdateOrder_MsgSenderIsNotOrderOwner(uint128,uint64,uint128,uint128,uint64) (runs: 256, μ: 468786, ~: 480890)
[PASS] testFailUpdateOrder_OrderDoesNotExist(uint256,uint256,bytes32) (runs: 256, μ: 263927, ~: 200498)
[PASS] testGetLimitOrderById() (gas: 460784)
[PASS] testGetOrderIds() (gas: 586292)
[PASS] testGetTotalOrdersValue() (gas: 456618)
[PASS] testIncreaseExecutionCredit(uint256,uint256,uint64) (runs: 256, μ: 372294, ~: 478880)
[PASS] testPlaceOrder(uint256,uint256) (runs: 256, μ: 325185, ~: 459615)
[PASS] testUpdateOrder(uint128,uint64,uint128,uint128,uint64) (runs: 256, μ: 375016, ~: 470387)
Test result: ok. 19 passed; 0 failed; finished in 384.78ms

Running 4 tests for src/test/ConveyorTickMath.t.sol:ConveyorTickMathTest
[PASS] testFromSqrtX96() (gas: 47179)
[PASS] testSimulateAmountOutOnSqrtPriceX96CrossTick(uint112) (runs: 256, μ: 489, ~: 503)
[PASS] testSimulateAmountOutOnSqrtPriceX96__ZeroForOneFalse(uint64) (runs: 256, μ: 29813, ~: 341)
[PASS] testSimulateAmountOutOnSqrtPriceX96__ZeroForOneTrue(uint64) (runs: 256, μ: 363, ~: 363)
Test result: ok. 4 passed; 0 failed; finished in 1.14s

Running 16 tests for src/test/LimitOrderRouter.t.sol:LimitOrderRouterTest
[PASS] testFailOnlyEOA() (gas: 5513)
[PASS] testFailRefreshOrder_OrderNotEligibleForRefresh() (gas: 509437)
[PASS] testFailValidateAndCancelOrder() (gas: 323124)
[PASS] testFailValidateOrderSequence_IncongruentBuySellStatusInBatch() (gas: 166215)
[PASS] testFailValidateOrderSequence_IncongruentFeeIn() (gas: 164123)
[PASS] testFailValidateOrderSequence_IncongruentFeeOut() (gas: 164167)
[PASS] testFailValidateOrderSequence_IncongruentInputTokenInBatch() (gas: 315126)
[PASS] testFailValidateOrderSequence_IncongruentStoplossStatus() (gas: 315205)
[PASS] testFailValidateOrderSequence_IncongruentTaxedTokenInBatch() (gas: 166269)
[PASS] testFailValidateOrderSequence_IncongruentTokenOut() (gas: 168298)
[PASS] testFailValidateOrderSequence_InvalidBatchOrder() (gas: 166737)
[PASS] testOnlyEOA() (gas: 10438)
[PASS] testRefreshOrder() (gas: 486514)
[PASS] testRefreshOrder_CancelOrderOrderExpired() (gas: 440432)
[PASS] testValidateAndCancelOrder() (gas: 344652)
[PASS] testValidateOrderSequence() (gas: 144532)
Test result: ok. 16 passed; 0 failed; finished in 27.87s

Running 7 tests for src/test/LimitOrderBatcher.t.sol:LimitOrderBatcherTest
[PASS] testFindBestTokenToTokenExecutionPrice() (gas: 10764)
[PASS] testSimulateAToBPriceChangeV2ReserveOutputs(uint112) (runs: 256, μ: 62896, ~: 85686)
[PASS] testSimulateAToBPriceChangeV2SpotPrice(uint64) (runs: 256, μ: 84094, ~: 85996)
[PASS] testSimulateAToWethPriceChangeV2() (gas: 12663)
[PASS] testSimulateAmountOutV3_Fuzz1_ZeroForOneFalse(uint64) (runs: 256, μ: 100039, ~: 105401)
[PASS] testSimulateAmountOutV3_Fuzz1_ZeroForOneTrue(uint64) (runs: 256, μ: 92465, ~: 107077)
[PASS] testSimulateWethToBPriceChangeV2() (gas: 12487)
Test result: ok. 7 passed; 0 failed; finished in 27.87s

Running 22 tests for src/test/SwapRouter.t.sol:SwapRouterTest
[PASS] testCalculateFee(uint112) (runs: 256, μ: 95338, ~: 97024)
[PASS] testCalculateOrderRewardBeacon(uint64) (runs: 256, μ: 16095, ~: 520)
[PASS] testCalculateOrderRewardConveyor(uint64) (runs: 256, μ: 19524, ~: 453)
[PASS] testCalculateV2SpotSushiTest1() (gas: 44374)
[PASS] testCalculateV2SpotSushiTest2() (gas: 50922)
[PASS] testCalculateV2SpotSushiTest3() (gas: 31312)
[PASS] testCalculateV2SpotUni1() (gas: 40873)
[PASS] testCalculateV2SpotUni2() (gas: 38653)
[PASS] testCalculateV2SpotUni3() (gas: 40976)
[PASS] testCalculateV3SpotPrice1() (gas: 35797)
[PASS] testCalculateV3SpotPrice2() (gas: 42977)
[PASS] testFailSwapV2_InsufficientOutputAmount() (gas: 215602)
[PASS] testFailSwap_InsufficientOutputAmount() (gas: 264632)
[PASS] testFailUniswapV3Callback_UnauthorizedUniswapV3CallbackCaller() (gas: 6551)
[PASS] testGetAllPrices2() (gas: 122979)
[PASS] testLPIsNotUniv3() (gas: 12551)
[PASS] testSwap() (gas: 240410)
[PASS] testSwapV2_1() (gas: 188807)
[PASS] testSwapV2_2() (gas: 214649)
[PASS] testSwapV2_3() (gas: 173020)
[PASS] testSwapV3_1() (gas: 159498)
[PASS] testSwapV3_2() (gas: 332643)
Test result: ok. 22 passed; 0 failed; finished in 28.46s
```

Code Coverage

The code coverage was collected using the command `forge coverage --fork-url $ETH_RPC_URL --ffi --fork-block-number 16225780`.

File	% Lines	% Statements	% Branches	% Funcs
src/ConveyorExecutor.sol	73.03% (65/89)	74.23% (72/97)	15.00% (3/20)	60.00% (6/10)
src/LimitOrderBatcher.sol	87.93% (153/174)	87.91% (189/215)	55.26% (21/38)	100.00% (16/16)
src/LimitOrderRouter.sol	70.71% (70/99)	72.90% (78/107)	43.48% (20/46)	66.67% (6/9)
src/OrderBook.sol	89.73% (131/146)	90.30% (149/165)	71.15% (37/52)	100.00% (16/16)
src/SwapRouter.sol	91.41% (117/128)	92.95% (145/156)	65.91% (29/44)	100.00% (16/16)
src/lib/ConveyorMath.sol	0.00% (0/261)	0.00% (0/289)	0.00% (0/210)	0.00% (0/22)
src/lib/ConveyorTickMath.sol	76.67% (23/30)	81.08% (30/37)	35.71% (5/14)	100.00% (2/2)
src/test/ConveyorTickMath.t.sol	66.67% (2/3)	66.67% (2/3)	100.00% (0/0)	66.67% (2/3)
src/test/LimitOrderBatcher.t.sol	100.00% (3/3)	100.00% (3/3)	100.00% (0/0)	100.00% (3/3)
src/test/LimitOrderExecutor.t.sol	12.50% (1/8)	12.50% (1/8)	100.00% (0/0)	22.22% (2/9)
src/test/LimitOrderRouter.t.sol	0.00% (0/1)	0.00% (0/1)	100.00% (0/0)	0.00% (0/2)
src/test/SwapRouter.t.sol	85.71% (6/7)	85.71% (6/7)	100.00% (0/0)	85.71% (6/7)
src/test/utils/Console.sol	0.00% (0/381)	0.00% (0/381)	100.00% (0/0)	0.00% (0/380)
src/test/utils/ScriptRunner.sol	46.67% (14/30)	45.00% (18/40)	100.00% (0/0)	40.00% (2/5)
src/test/utils/Swap.sol	55.56% (5/9)	54.55% (6/11)	100.00% (0/0)	50.00% (1/2)
src/test/utils/Utils.sol	0.00% (0/29)	0.00% (0/34)	0.00% (0/18)	0.00% (0/2)
src/test/utils/test.sol	0.00% (0/207)	0.00% (0/209)	0.00% (0/104)	0.00% (0/54)
Total	36.76% (590/1605)	39.65% (699/1763)	21.06% (115/546)	13.98% (78/558)

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

```
4495363cae70103f2d8cd8b3c98af4d6f8c0c06733f829dd14ea6bc1adbd7e1c ./src/GasOracle.sol
9b3d75bcffd310cf8ecb1b6fee9bb9b3c59af06844c2eb67c444dd2159a0fb6d ./src/LimitOrderBatcher.sol
f072b75049a3865432eb4fccb9e9ce1869a10ece155364d2410a20539a178471 ./src/TokenToTokenLimitOrderExecution.sol
8b56b5ac7a4c80b4429f2ba73195ca7667b0b38f1372446cb498e680638a1047 ./src/TokenToWethLimitOrderExecution.sol
4646258867ee3c3223cacc4aff741606049b12e6598132050f5ae23890c38907 ./src/SwapRouter.sol
0654759b59538cd72f7d1fe82cd57f302cf07ff58bd8ccc3420c9d9c7fd9a0b0 ./src/ConveyorErrors.sol
325188dfc2be4a4bd5ff0da96dae68cd3034db83eb77002c5cd535f459cc810a ./src/TaxedTokenLimitOrderExecution.sol
1e9e9ebf0425d83b654bf12581de631de48c8eadae8a5acbb123e47de4ec34ed ./src/LimitOrderRouter.sol
0c09bdc4d938d55447c9a9fffc8459e199be79b8e686d96a1df94946ed13152e ./src/OrderBook.sol
```

Tests

```
bde86c58502fddd6a618aad4210b669b9500c0a17b5f4d66b4e68999f61892f2 ./test/GasOracle.t.sol
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 ./test/TaxedTokenToTokenExecution.t.sol
7775677a2cde54d059c69d8265e2067a2154f4c1879778651e05c979e52d341f ./test/LimitOrderBatcher.t.sol
7a96279dbdcc7f5e69a41c62b47d34b310959cf5c3e58bb301a8a6fd32adb515 ./test/ConveyorLimitOrders.t.sol
8b17fd3861b011b467960eba084ee6617028c89c895b69d45ba1873d965efcc2 ./test/OrderRouter.t.sol
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 ./test/TokenToWethLimitOrderExectuion.t.sol
c3de1da244718dede0f7b8c0416f6ec96c20499e5ead2ba5fa51b74049bce4b3 ./test/LimitOrderRouter.t.sol
01ccf09f3003a1ae2d458f9507072911d4facc36e4cd37b960265bb189c5f570 ./test/OrderBook.t.sol
8cfdadfe82a616a1cfdb937daab5278a47625c72771c9568d8ae986326adc6d1 ./test/utils/Swap.sol
c8c9e46ad7a6b504436d75d132cf16ccbefc90652d5b90968210f2fb6d6a5f62 ./test/utils/Console.sol
35fb2580dbedcb348ec4e5d5f25e09caab9692c3f7e46979fb7b70431845c53e ./test/utils/Utils.sol
9473ec69fa6aa5688ee0a3addf35740ab5aae404b133cd61e5259063cfbe94fe ./test/utils/ScriptRunner.sol
b5c6c405587dad532393f30af1b04ea7e9c2957abdec54a8de7b07f6714c0a6b ./test/utils/test.sol
```

Changelog

- 2022-10-10 - Initial report
- 2022-12-21 - Fix review (`d1b566d7`)
- 2023-02-09 - Fix review (`50c96ad8`)

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Aave, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites&aspo; owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that your access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.