Uniswap V3 Limit Orders Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

Hans

Assisting Auditors

Alex Roan

June 7, 2023

Contents

1 About Cyfrin

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

2 Disclaimer

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

3 Risk Classification

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

4 Protocol Summary

Uniswap V3 Limit Orders is a protocol that extends the functionality of Uniswap v3 by allowing users to place limit orders as liquidity positions in a given underlying Uniswap v3 pool. The protocol leverages Chainlink Automation to fulfill in the money orders is currently intended to be deployed to Ethereum mainnet, Polygon mainnet, Optimism and Arbitrum.

5 Executive Summary

Over the course of 11 days, the Cyfrin team conducted an audit on the Uniswap V3 Limit Orders smart contracts provided by GFX Labs. In this period, a total of 21 issues were found.

Summary

Project Name	Uniswap V3 Limit Orders
Repository	uniswap-v3-limit-orders
Commit	83f5db9cb909...
Audit Timeline	May 17th - May 31st
Methods	Manual Review, Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	6
Low Risk	5
Informational	7
Gas Optimizations	3
Total Issues	21

Summary of Findings

[M-1] Calls to `LimitOrderRegistry::newOrder` might revert due to overflow	Resolved
[M-2] New orders on a given pool in the opposite direction, separated by zero/one tick space, are not possible until previous `BatchOrder` is removed from the order book	Resolved
[M-3] Calls to `LimitOrderRegistry::cancelOrder` might revert due to overflow	Resolved
[M-4] A malicious user can cancel an ITM order at a given target tick by calling `LimitOrderRegistry::cancelOrder` with the opposite direction, separated by one tick space	Resolved
[M-5] Malicious validators can prevent orders from being created or cancelled	Resolved
[M-6] Gas griefing denial-of-service on `performUpkeep`	Resolved
[L-1] Perform additional validation on Chainlink fast gas feed	Acknowledged
[L-2] Withdrawing native assets may revert if wrapped native balance is zero	Resolved
[L-3] `call()` should be used instead of `transfer()` on address payable	Resolved
[L-4] Fee-on-transfer/deflationary tokens will not be supported	Acknowledged
[L-5] Fulfillable ITM orders may not always be fulfilled	Acknowledged
[I-1] Spelling errors and incorrect NatSpec	Resolved
[I-2] Avoid hardcoding addresses when contract is intended to be deployed to multiple chains	Resolved
[I-3] Validate inputs to `onlyOwner` functions	Acknowledged
[I-4] Limit orders can be frozen for one side of a Uniswap v3 pool if `minimumAssets` has not been set for one of the tokens	Acknowledged
[I-5] Improvements to use of ternary operator	Resolved
[I-6] Move shared logic to internal functions called within a modifier	Acknowledged
[I-7] Re-entrancy in `LimitOrderRegistry::newOrder` means tokens with transfer hooks could take over execution to manipulate price to be immediately ITM, bypassing validation	Resolved
[G-1] Use stack variables rather than making repeated external calls	Resolved

[G-2] Return result directly rather than assigning to a stack variable	Resolved
[G-3] Perform post-increment in a single line	Resolved

6 Findings

6.1 Medium Risk

6.1.1 Calls to `LimitOrderRegistry::newOrder` might revert due to overflow

Description: Reasonable input could cause an arithmetic overflow when opening new orders because large multiplications are performed on variables defined as `uint128` instead of `uint256`. Specifically, in `LimitOrderRegistry::_mintPosition` and `LimitOrderRegistry::_addToPosition` the following lines (which appear in both functions) are problematic:

```
uint128 amount0Min = amount0 == 0 ? 0 : (amount0 * 0.9999e18) / 1e18;
uint128 amount1Min = amount1 == 0 ? 0 : (amount1 * 0.9999e18) / 1e18;
```

Impact: It is not possible for users to create new orders with deposit amounts in excess of `341e18`, limiting the protocol to working with comparatively small orders.

Proof of Concept: Paste this test into `test/LimitOrderRegistry.t.sol`:

```
function test_OverflowingNewOrder() public {
    uint96 amount = 340_316_398_560_794_542_918;
    address msgSender = 0xE0b906ae06BfB1b54fad61E222b2E324D51e1da6;
    deal(address(USDC), msgSender, amount);
    vm.startPrank(msgSender);
    USDC.approve(address(registry), amount);

    registry.newOrder(USDC_WETH_05_POOL, 204900, amount, true, 0);
}
```

Recommended Mitigation: Cast the `uint128` value to `uint256` prior to performing the multiplication:

```
uint128 amount0Min = amount0 == 0 ? 0 : uint128((uint256(amount0) * 0.9999e18) / 1e18);
uint128 amount1Min = amount1 == 0 ? 0 : uint128((uint256(amount1) * 0.9999e18) / 1e18);
```

GFX Labs: Fixed by changing multipliers from 18 decimals to 4 decimals in commits f9934fe and c731cd4.

Cyfrin: Acknowledged.

6.1.2 New orders on a given pool in the opposite direction, separated by zero/one tick space, are not possible until previous `BatchOrder` is removed from the order book

Description: New order creation will revert with `LimitOrderRegistry__DirectionMisMatch()` if the direction opposes any existing orders at the current tick price. Due to the calculation of lower/upper ticks, this also applies to orders separated by one tick space. In the event price deviation causes existing orders to become ITM, it is not possible to place new orders in the opposing direction until upkeep has been performed for the existing orders, fully removing them from the order book.

Impact: This edge case is only an issue in the following circumstance:

- Any number of users place an order at a given tick price.
- Price deviates, causing this `BatchOrder` (and potentially others) to become ITM.
- If upkeep has not yet been performed, either through DoS, oracle downtime, or exceeding the maximum number of orders per upkeep (in the case of very large price deviations), the original `BatchOrder` remains on the order book (represented by a doubly linked list).
- So long as the original order remains in the list, new orders on a given pool in the opposite direction, separated by zero/one tick space, cannot be placed.

Proof of Concept:

```
function testOppositeOrders() external {
    uint256 amount = 1_000e6;
    deal(address(USDC), address(this), amount);

    // Current tick 204332
    // 204367
    // Current block 16371089
    USDC.approve(address(registry), amount);
    registry.newOrder(USDC_WETH_05_POOL, 204910, uint96(amount), true, 0);

    // Make a large swap to move the pool tick.
    address[] memory path = new address[](2);
    path[0] = address(WETH);
    path[1] = address(USDC);

    uint24[] memory poolFees = new uint24[](1);
    poolFees[0] = 500;

    (bool upkeepNeeded, bytes memory performData) = registry.checkUpkeep(abi.encode(USDC_WETH_05_POOL));

    uint256 swapAmount = 2_000e18;
    deal(address(WETH), address(this), swapAmount);
    _swap(path, poolFees, swapAmount);

    (upkeepNeeded, performData) = registry.checkUpkeep(abi.encode(USDC_WETH_05_POOL));

    assertTrue(upkeepNeeded, "Upkeep should be needed.");

    // registry.performUpkeep(performData);

    amount = 2_000e17;
    deal(address(WETH), address(this), amount);
    WETH.approve(address(registry), amount);
    int24 target = 204910 - USDC_WETH_05_POOL.tickSpacing();
    vm.expectRevert(
        abi.encodeWithSelector(LimitOrderRegistry.LimitOrderRegistry__DirectionMisMatch.selector)
    );
    registry.newOrder(USDC_WETH_05_POOL, target, uint96(amount), false, 0);
}
```

Recommended Mitigation: Implement a second order book for orders in the opposite direction, as discussed.

GFX Labs: Fixed by separating the order book into two lists, and having orders in opposite directions use completely different LP positions in commits 7b65915 and 9e9ceda.

Cyfrin: Acknowledged. Comment should be updated for `getPositionFromTicks` mapping to include 'direction' key.

GFX Labs: Fixed in commit f303a50.

Cyfrin: Acknowledged.

6.1.3 Calls to `LimitOrderRegistry::cancelOrder` might revert due to overflow

Description: Reasonable input could cause an arithmetic overflow when cancelling orders because large multiplications are performed on variables defined as `uint128` instead of `uint256`. Specifically, when calculating the liquidity percentage to take from a position in `LimitOrderRegistry::cancelOrder`, multiplication of `depositAmount` by `1e18` would cause an overflow when `depositAmount >= 341e18` as the result exceeds the range of `uint128`.

```
uint128 depositAmount = batchIdToUserDepositAmount[batchId][sender];
if (depositAmount == 0) revert LimitOrderRegistry__UserNotFound(sender, batchId);

// Remove one from the userCount.
order.userCount--;

// Zero out user balance.
delete batchIdToUserDepositAmount[batchId][sender];

uint128 orderAmount;
if (order.direction) {
    orderAmount = order.token0Amount;
    if (orderAmount == depositAmount) {
        liquidityPercentToTake = 1e18;
        // Update order tokenAmount.
        order.token0Amount = 0;
    } else {
        liquidityPercentToTake = (1e18 * depositAmount) / orderAmount; // @audit - overflow
        // Update order tokenAmount.
        order.token0Amount = orderAmount - depositAmount;
    }
} else {
    orderAmount = order.token1Amount;
    if (orderAmount == depositAmount) {
        liquidityPercentToTake = 1e18;
        // Update order tokenAmount.
        order.token1Amount = 0;
    } else {
        liquidityPercentToTake = (1e18 * depositAmount) / orderAmount; // @audit - overflow
        // Update order tokenAmount.
        order.token1Amount = orderAmount - depositAmount;
    }
}
```

Impact: It can become impossible for a user to cancel their order if their deposit amount equals or exceeds `341e18`, unless they are the final depositor to the pool.

Recommended Mitigation:

```
liquidityPercentToTake = (1e18 * uint256(depositAmount)) / orderAmount;
```

GFX Labs: Fixed by casting `depositAmount` as a `uint256` in commit d67b293.

Cyfrin: Acknowledged.

6.1.4 A malicious user can cancel an ITM order at a given target tick by calling `LimitOrderRegistry::cancelOrder` with the opposite direction, separated by one tick space

Description: Users are able to cancel their limit orders by calling `LimitOrderRegistry::cancelOrder`. By internally calling `_getOrderStatus` and validating the return value, it is intended that only OTM orders should be able to be cancelled.

```
    function cancelOrder(
        UniswapV3Pool pool,
        int24 targetTick,
        bool direction //@audit don't validate order.direction == direction
    ) external returns (uint128 amount0, uint128 amount1, uint128 batchId) {
        uint256 positionId;
        {
```

```
            // Make sure order is OTM.
            (, int24 tick, , , , , ) = pool.slot0();

            // Determine upper and lower ticks.
            int24 upper;
            int24 lower;
            {
                int24 tickSpacing = pool.tickSpacing();
                // Make sure targetTick is divisible by spacing.
                if (targetTick % tickSpacing != 0)
                    revert LimitOrderRegistry__InvalidTargetTick(targetTick, tickSpacing);
                if (direction) {
                    upper = targetTick;
                    lower = targetTick - tickSpacing;
                } else {
                    upper = targetTick + tickSpacing;
                    lower = targetTick;
                }
            }
            // Validate lower, upper,and direction. Make sure order is not ITM or MIXED
            {
                OrderStatus status = _getOrderStatus(tick, lower, upper, direction);
                if (status != OrderStatus.OTM) revert LimitOrderRegistry__OrderITM(tick, targetTick,
                ↪  direction);
            }

            // Get the position id.
            positionId = getPositionFromTicks[pool][lower][upper];

            if (positionId == 0) revert LimitOrderRegistry__InvalidPositionId();
        }
        ...
```

The main flaw is that users are able to provide a custom `direction` as a parameter but there is no validation that `direction == order.direction`.

A malicious user can therefore pass the opposite direction with target tick separated by one tick space from the real target tick to satisfy the OTM condition.

Once the batch order is retrieved from `positionId`, the function will continue to work correctly as it makes of use of `order.direction` from there ownard.

Impact: Malicious users can cancel orders at a given target tick by providing tick separated by one tick space and opposite direction. This could result in their placing of risk-free trades which can be cancelled at-will even when they are ITM and should not be able to be cancelled.

Proof of Concept:

```
function testCancellingITMOrderWrongly() external {
        uint96 usdcAmount = 1_000e6;
        int24 poolTick;
        int24 tickSpace = USDC_WETH_05_POOL.tickSpacing();
        {
            (, int24 tick, , , , , ) = USDC_WETH_05_POOL.slot0();
            poolTick = tick - (tick % tickSpace);
        }

        // Create orders.
        _createOrder(address(this), USDC_WETH_05_POOL, 2 * tickSpace, USDC, usdcAmount);

        // Make first order ITM.
        {
```

```
        address[] memory path = new address[](2);
        path[0] = address(WETH);
        path[1] = address(USDC);

        uint24[] memory poolFees = new uint24[](1);
        poolFees[0] = 500;

        uint256 swapAmount = 770e18;
        deal(address(WETH), address(this), swapAmount);
        _swap(path, poolFees, swapAmount);
    }

    (, int24 currentTick, , , , , ) = USDC_WETH_05_POOL.slot0();

    // Try to cancel it. But revert as it's ITM.
    vm.expectRevert(
        abi.encodeWithSelector(
            LimitOrderRegistry.LimitOrderRegistry__OrderITM.selector,
            currentTick,
            poolTick + 20,
            true
        )
    );
    registry.cancelOrder(USDC_WETH_05_POOL, poolTick + 2 * tickSpace, true);

    // Cancel with opposite direction, separated by one tick space
    registry.cancelOrder(USDC_WETH_05_POOL, poolTick + tickSpace, false);
}
```

Note: we were not able to fully validate this finding with a PoC in the allotted time - it currently outputs `LimitOrderRegistry__NoLiquidityInOrder()` as `amount0 == 0` when `order.direction = true`.

Temporarily removing the below validation section in `LimitOrderRegistry::cancelOrder` to demonstrate this finding, the ITM order can be canceled.

```
    if (order.direction) {
        if (amount0 > 0) poolToData[pool].token0.safeTransfer(sender, amount0);
        else revert LimitOrderRegistry__NoLiquidityInOrder();
        if (amount1 > 0) revert LimitOrderRegistry__AmountShouldBeZero();
    } else {
        if (amount1 > 0) poolToData[pool].token1.safeTransfer(sender, amount1);
        else revert LimitOrderRegistry__NoLiquidityInOrder();
        if (amount0 > 0) revert LimitOrderRegistry__AmountShouldBeZero();
    }
```

Recommended Mitigation: Validation that `direction == order.direction` should be added to `LimitOrderRegistry::cancelOrder`.

GFX Labs: Fixed by separating the order book into two lists, and having orders in opposite directions use completely different LP positions in commits 7b65915 and 9e9ceda.

Cyfrin: Acknowledged.

6.1.5 Malicious validators can prevent orders from being created or cancelled

Description: Use of `block.timestamp` as the deadline for `MintParams`, `IncreaseLiquidityParams` and `DecreaseLiquidityParams` means that a given transaction interfacing with Uniswap v3 will be valid whenever the validator decides to include it. This could result in orders prevented from being created or cancelled if a malicious

validator holds these transactions until after the tick price for the given pool has moved such that the deadline is valid but order status is no longer valid.

Impact: Whenever the validator decides to include the transaction in a block, it will be valid at that time, since `block.timestamp` will be the current timestamp. This could result in forcing an order to be fulfilled when it was the sender's intention to have it cancelled, by holding until price exceeds the target tick as ITM orders can't be cancelled, or never creating the order at all, by similar reasoning as it is not allowed to create orders that are immediately ITM.

Recommended Mitigation: Add deadline arguments to all functions that interact with Uniswap v3 via the `NonFungiblePositionManager`, and pass it along to the associated calls.

GFX Labs: Fixed by adding deadline arguments to all Position Manager calls in commit f05cdfc.

Cyfrin: Acknowledged.

6.1.6 Gas griefing denial-of-service on `performUpkeep`

Description: To mitigate against issues arising from a tangled list, `LimitOrderRegistry::performUpkeep` was modified to continue walking the list toward its head/tail until the next order that matches the walk direction is found. This is achieved using a while loop; however, this is susceptible to gas griefing attacks if there are a large number of orders with the opposite direction placed between the current and next order. Whilst order spamming is somewhat mitigated by the `minimumAssets` requirement, a reasonably well-funded/anarchic attacker can cause this loop to consume too much gas, greater than the maximum specified gas per upkeep, by placing a series of small orders which could prevent legitimate ITM orders from being fulfilled.

Impact: It is possible for a malicious user to perform a denial-of-service attack on upkeep, preventing ITM orders from being fulfilled.

Recommended Mitigation: Implement a second order book for orders in the opposite direction, as discussed, removing the need to walk the list until the next order that matches the walk direction is found.

GFX Labs: Fixed by separating the order book into two lists, and having orders in opposite directions use completely different LP positions in commits 7b65915 and 9e9ceda.

Cyfrin: Acknowledged.

6.2 Low Risk

6.2.1 Perform additional validation on Chainlink fast gas feed

Description: When consuming data feeds provided by Chainlink, it is important to validate a number of thresholds and return values. Without this, it is possible for a consuming contract to use stale or otherwise incorrect/invalid data. `LimitOrderRegistry` currently correctly validates the time since the last update against an owner-specified heartbeat value; however, it is possible to perform additional validation to ensure that the returned gas price is indeed valid, for example within upper/lower bounds and the result of a complete reporting round.

Impact: `LimitOrderRegistry::performUpkeep` is reliant on this functionality within `LimitOrderRegistry::getGasPrice` which could result in DoS on fulfilling orders if not functioning correctly; however, the escape-hatch is simply having the owner set the feed address to `address(0)` which causes the owner-specified fallback value to be used.

Recommended Mitigation: Consider calling `IChainlinkAggregator::latestRoundData` within a `try/catch` block and include the following additional validation:

```
require(signedGasPrice > 0, "Negative gas price");
require(signedGasPrice < maxGasPrice, "Upper gas price bound breached");
require(signedGasPrice > minGasPrice, "Lower gas price bound breached");
require(answeredInRound >= roundID, "Round incomplete");
```

GFX Labs: Acknowledged. Chainlink has an outstanding track record for their data feed accuracy and reliability, and if Chainlink nodes did want to start maliciously reporting incorrect values, there are significantly more profitable opportunities elsewhere.

Also, in the event of the fast gas feed becoming unreliable, the owner can either manually set the gas price, or it would be straightforward to make a custom Fast Gas Feed contract that is updated by a GFX Labs bot.

Cyfrin: Acknowledged.

6.2.2 Withdrawing native assets may revert if wrapped native balance is zero

Description: The function `LimitOrderRegistry::withdrawNative` allows the owner to withdraw native and wrapped native assets from the contract. If balances of both are zero, calls to this function revert as there is nothing to withdraw.

```
/**
 * @notice Allows owner to withdraw wrapped native and native assets from this contract.
 */
function withdrawNative() external onlyOwner {
    uint256 wrappedNativeBalance = WRAPPED_NATIVE.balanceOf(address(this));
    uint256 nativeBalance = address(this).balance;
    // Make sure there is something to withdraw.
    if (wrappedNativeBalance == 0 && nativeBalance == 0) revert LimitOrderRegistry__ZeroNativeBalance();
    WRAPPED_NATIVE.safeTransfer(owner, WRAPPED_NATIVE.balanceOf(address(this)));
    payable(owner).transfer(address(this).balance);
}
```

A zero value call with transfer non-zero wrapped native token balance will succeed just fine; however the call may revert in the opposite case. Given `safeTransfer` calls the wrapped native token's `transfer` function, the entire transaction for a non-zero value call with zero wrapped native token balance could revert if the wrapped native token reverts on zero-value transfers.

Impact: This would temporarily prevent the withdrawal of any native token balance until the wrapped native token balance is also non-zero, although it seems none of the wrapped native tokens revert on zero transfers on current intended target chains.

Recommended Mitigation: Separately validate the wrapped native token balance prior to attempting the transfer.

GFX Labs: Fixed by adding `if` statements to check the amount is non-zero before attempting to withdraw in commit d2dd99c.

Cyfrin: Acknowledged. It is not necessary to revert - this line can be removed:

```
if (wrappedNativeBalance == 0 && nativeBalance == 0) revert LimitOrderRegistry__ZeroNativeBalance();
```

GFX Labs: Acknowledged. Revert isn't strictly necessary, but also doesn't hurt.

Cyfrin: Acknowledged.

6.2.3 `call()` should be used instead of `transfer()` on address payable

Description: The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the `SLOAD` instruction. It is now therefore no longer recommended to use these functions on address payable.

There are currently a number of instances where use of `transfer()` should be resolved.

Impact: The use of the deprecated `transfer()` function will cause transactions to fail:

- If `owner` calling `LimitOrderRegistry::withdrawNative/sender` calling `LimitOrderRegistry::claimOrder` is a smart contract that does not implement a payable function.
- If `owner` calling `LimitOrderRegistry::withdrawNative/sender` calling `LimitOrderRegistry::claimOrder` is a smart contract that does implement a payable fallback which uses more than 2300 gas unit.
- If `sender` calling `LimitOrderRegistry::claimOrder` is a smart contract that implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.
- Additionally, using higher than 2300 gas might be mandatory for some multi-signature wallets.

Recommended Mitigation: Use `call()` or solmate's `safeTransferETH` instead of `transfer()`, but be sure to respect the Checks Effects Interactions (CEI) pattern.

GFX Labs: Fixed by using solmate `safeTransferETH` in commit 6c486c9.

Cyfrin: Acknowledged.

6.2.4 Fee-on-transfer/deflationary tokens will not be supported

Description: When creating a new order, `LimitOrderRegistry::newOrder` assumes that the amount of deposited tokens is equal to the function parameter plus the balance before the deposit. For tokens which take a fee on every transfer, this assumption does not hold and so the true amount transferred is less than the specified amount. As a result, tight slippage parameters in `_mintPosition` and `_addToPosition` which are calculated using this value will likely cause transactions to revert:

```
uint128 amount0Min = amount0 == 0 ? 0 : (amount0 * 0.9999e18) / 1e18;
```

Additionally, the amount stored in `batchIdToUserDepositAmount` will be larger than the true balance deposited by a given user, resulting in incorrect accounting.

Impact: It will not be possible to use fee-on-transfer/deflationary tokens within the protocol.

Recommended Mitigation: Avoid allowlisting such problematic tokens. If it is desired to support such tokens then it recommended to cache token balances before and after any transfers, using the difference between those two balances rather than the input amount as the amount received.

GFX Labs: Acknowledged. This contract will not use fee-on-transfer or deflationary tokens.

Cyfrin: Acknowledged.

6.2.5 Fulfillable ITM orders may not always be fulfilled

Description: Currently, orders are only fulfillable via `LimitOrderRegistry::performUpkeep` which processes them via the doubly linked list, with a limit of `maxFillsPerUpkeep` orders per call. Considering extreme and/or adversarial market conditions, there is no guarantee that all legitimate orders will eventually be processed. If the pool tick varies significantly and rapidly, as is the case for large price swings, and there exist large number of batch orders on the book, greater than `maxFillsPerUpkeep`, then only a subset of orders can be fulfilled in a given block. Assuming the price deviation lasts only a small number of blocks, it is possible that fulfillable ITM orders in the list are not cleared before they once again become OTM.

Impact: Valid and fulfillable ITM orders may not be fulfilled even though their liquidity within the position is technically being used for the duration the pool tick exceeds their target tick price.

Recommended Mitigation: To guarantee that any order can be fulfilled, consider adding a public `fulfillOrder` function that can be called by any user to fulfil a specific order independently from its position in the list. The order can be simply fulfilled and removed from the list, without affecting the upkeep procedure.

GFX Labs: Acknowledged. Even in traditional market settings, price can go past some limit order trigger, but that order can still be unfilled, if there is not enough volume.

Adding a custom function to allow users to fulfill a specific order is a possible solution, but its only a solution if the users create some bot which is better at TX management than Chainlink Automation, and that is a big ask for users. Because of this, the function was not added to reduce contract size.

Cyfrin: Acknowledged.

6.3 Informational

6.3.1 Spelling errors and incorrect NatSpec

The following spelling errors and incorrect NatSpec were identified:

- `cancelOrder` is documented to send all swap fees from a position to the last person that cancels the order; however, this is not the actual behaviour as fees are in fact stored in the `tokenToSwapFees` mapping within the internal call to `LimitOrderRegistry::_takeFromPosition`, withdrawable by the owner in `LimitOrderRegistry::withdrawSwapFees`. Therefore, this comment should be removed.
- References to 'Cellar' should be updated or removed.
- This comment should clearly state that the new order will revert if it is either ITM or in the MIXED state (i.e. not OTM).
- 'doubley linked list' should be spelled 'doubly linked list'.
- 'effectedOrder' should be spelled 'affectedOrder'.
- `LimitOrderRegistry::getGasPrice` is a view function and as such should be moved below to within the 'view logic' section header.

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.3.2 Avoid hardcoding addresses when contract is intended to be deployed to multiple chains

While the `LimitOrderRegistry::fastGasFeed` state variable is overwritten during construction, it is advised to avoid initializing this variable with a hardcoded address as this contract is intended to be deployed to multiple chains and there is no guarantee that the address will always be the same. Instead, simply declare the variable and add a comment indicating the mainnet address as is the case for other such variables.

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.3.3 Validate inputs to `onlyOwner` functions

Whilst there is a comment that states no input validation is performed on inputs to `LimitOrderRegistry::setRegistrar` and `LimitOrderRegistry::setFastGasFeed` because it is in the owner's best interest to choose valid addresses, it is recommended that some guardrails still be implemented.

For example, to validate the registrar, the contract could call `IKeeperRegistrar::typeAndVersion` and validate that the return value matches that expected in `LimitOrderRegistry::setupLimitOrder` prior to setting it in storage. The `IKeeperRegistrar::register` selector could also be loosely validated by performing a call to this function within a `try/catch` block, passing `address(0)` for the `adminAddress` and catching the `InvalidAdminAddress()` custom error which would indicate expected behaviour.

To validate the fast gas feed, the contract could first make a call to the data feed and check it is reporting correctly within an expected range prior to setting its value in storage.

GFX Labs: Acknowledged. This is a valid concern, but in an effort to reduce contract size, these checks will not be added. Also if the owner mistakenly puts in an illogical value, they can simply call the function again with the correct value.

Cyfrin: Acknowledged.

6.3.4 Limit orders can be frozen for one side of a Uniswap v3 pool if `minimumAssets` has not been set for one of the tokens

To prevent spamming low liquidity orders, `LimitOrderRegistry::minimumAssets` must be set for a given asset before limit orders are allowed to be placed. A Uniswap v3 pool contains two assets, but `LimitOrderReg-`

`istry::setMinimumAssets` is called for a specific asset across the entire contract and is not pool-specific. If this function is not called for both assets in a given pool then limit orders in a given direction will not be allowed, depending on which asset has no minimum set.

GFX Labs: Acknowledged. It is in the owner's best interest to make sure both assets in a pool are supported, so that there are more limit orders, and more swap fees generated.

Cyfrin: Acknowledged.

6.3.5 Improvements to use of ternary operator

There is currently one instance of ternary operator usage that can be simplified, with the boolean expression being evaluated directly:

```
bool direction = targetTick > node.tickUpper ? true : false;
```

should be changed to:

```
bool direction = targetTick > node.tickUpper;
```

Additionally, there is another conditional statement where use of the ternary operator would be recommended:

```
if (direction)
    assetIn = poolToData[pool].token0;
else assetIn = poolToData[pool].token1;
```

This conditional statement could be replaced with

```
assetIn = direction ? poolToData[pool].token0 : poolToData[pool].token1;
```

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.3.6 Move shared logic to internal functions called within a modifier

```
...
if (direction) data.token0.safeApprove(address(POSITION_MANAGER), amount0);
else data.token1.safeApprove(address(POSITION_MANAGER), amount1);

    // 0.9999e18 accounts for rounding errors in the Uniswap V3 protocol.
    uint128 amount0Min = amount0 == 0 ? 0 : (amount0 * 0.9999e18) / 1e18;
    uint128 amount1Min = amount1 == 0 ? 0 : (amount1 * 0.9999e18) / 1e18;
...
// If position manager still has allowance, zero it out.
    if (direction && data.token0.allowance(address(this), address(POSITION_MANAGER)) > 0)
        data.token0.safeApprove(address(POSITION_MANAGER), 0);
    if (!direction && data.token1.allowance(address(this), address(POSITION_MANAGER)) > 0)
        data.token1.safeApprove(address(POSITION_MANAGER), 0);
```

The above logic in `_mintPosition` and `_addToPosition` is shared and repeated across both functions. Consider moving this code to internal functions which are called within a modifier like so:

```
modifier approvePositionManager() {
    _approveBefore();
    _;
    _approveAfter();
}
```

GFX Labs: Acknowledged. This is a valid concern, but in an effort to reduce contract size and minimize code changes, this will not be implemented.

Cyfrin: Acknowledged.

6.3.7 Re-entrancy in `LimitOrderRegistry::newOrder` means tokens with transfer hooks could take over execution to manipulate price to be immediately ITM, bypassing validation

Whilst there does not appear to be any direct benefit to an attacker, it is possible for tokens with transfer hooks to take over execution from in-flight calls to `LimitOrderRegistry::newOrder`. This function validates that the new order status is OTM prior to performing the transfer of `assetIn` from the caller. An input token such as an ERC-777, which has such transfer hooks, could therefore be used to skew the pool tick after the new order has already passed validation such that it is actually MIXED or ITM. To mitigate against this, consider moving the asset transfer block to before the order is validated OTM, carefully select allowlisted tokens or make `newOrder` non-reentrant.

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.4 Gas Optimization

6.4.1 Use stack variables rather than making repeated external calls

Within `LimitOrderRegistry::withdrawNative`, calls to determine the contract balance of native/wrapped native assets are cached in stack variables but then never used. When performing subsequent transfers, the respective functions are called again to determine the amount to send, wasting gas on repeated external calls as the stack variables could simply be used instead.

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.4.2 Return result directly rather than assigning to a stack variable

In `LimitOrderRegistry::newOrder`, the return value is assigned to a stack variable, `batchId`, which is never used aside from the return statement. This assignment is not necessary as the return can be performed in a single line:

```
return orderBook[details.positionId].batchId;
```

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.

6.4.3 Perform post-increment in a single line

The post-increment of `batchCount` variable in `LimitOrderRegistry::_setupOrder` can be performed in a single line when it is assigned to `order.batchId`:

```
  function _setupOrder(bool direction, uint256 position) internal {
     BatchOrder storage order = orderBook[position];
-    order.batchId = batchCount;
+    order.batchId = batchCount++;
     order.direction = direction;
-    batchCount++;
  }
```

GFX Labs: Fixed in commit 4cff05a.

Cyfrin: Acknowledged.