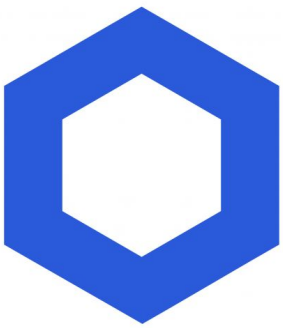
Uniswap Limit Orders Audit

Smart Contract Security Assessment

May 04, 2023



ABSTRACT

Dedaub was commissioned to perform a security audit of the Uniswap Limit Orders functionality for the Chainlink protocol.

SETTING & CAVEATS

This audit report refers to the Chainlink repository located at https://github.com/crispymangoes/uniswap-v3-limit-orders, commit number `249257bd105bd37df1fa7b1adfe7aad9625b8a2d`.

The full list of audited contracts can be found bellow:

```
├── src
│   ├── interfaces
│   │   ├── chainlink
│   │   │   ├── IChainlinkAggregator.sol
│   │   │   └── IKeeperRegistrar.sol
│   │   └── uniswapV3
│   │       ├── IUniswapV3Router.sol
│   │       ├── NonFungiblePositionManager.sol
│   │       └── UniswapV3Pool.sol
│   ├── LimitOrderRegistryLens.sol
│   ├── LimitOrderRegistry.sol
│   ├── MockERC20.sol
│   ├── TradeManagerFactory.sol
│   └── TradeManager.sol
```

Two auditors worked on the codebase for 14 days.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	Loss of funds made possible by the use of “stale” orders in the opposite direction.	**OPEN**

The protocol uses a uniswap LP position for each created order. The position, however, depends only on the tick range of the order and not on its direction. A position could be used in the upward direction if the current tick is lower than the position’s range. Then, at a later moment the tick could be higher than the position’s range, and the same position will be used in the downward direction.

When a new order is created, `LimitOrderRegistry::newOrder` checks whether a position already exists for that range, and if so, whether that position is already used in the list. If this is the case, the existing order will be reused by adding more liquidity to it and updating its amount.

```
details.positionId =
    getPositionFromTicks[pool][details.lower][details.upper];

if (details.positionId == 0) {
    ...
} else {
    // Check if the position id is already being used in List.
    BatchOrder memory order = orderBook[details.positionId];
    if (order.token0Amount > 0 || order.token1Amount > 0) {
        // Dedaub: this order might have the wrong direction!
        // Need to add liquidity.
        PoolData memory data = poolToData[pool];
        _addToPosition(data, details.positionId, details.amount0, details.amount1,
direction);

        // Update token0Amount, token1Amount, batchIdToUserDepositAmount mapping.
        details.userTotal = _updateOrder(details.positionId, sender, amount);
    } else {
```

```
        ...
    }
}
```

The above code assumes that the existing order will have the same direction as the new one. This, however, is not guaranteed! It could be the case that this order has the opposite direction, so it is now ITM, but has simply not been fulfilled yet.

For example, imagine that the current price is $1.5k/ETH and an order is created to swap 100 ETH at $2k/ETH. This will create a position in the $1999-$2k range (numbers are simplified). Later ETH goes above $2k, so the position now contains $200k, but the order is not fulfilled yet.

Now a malicious user creates a new **downward** order to swap $5k at $1999/ETH. This order also uses a position in the $1999-$2000 range, and since such a position already exists, it will be reused, despite the fact that the previous order is in the wrong direction.

`LimitOrderRegistry::newOrder` will then call `_addToPosition`, adding the new $5k to the position, which now contains $205k. Note that `_addToPosition` takes the `direction` as an argument, so it will use the **new downward** direction.

Then, `newOrder` calls `_updateOrder` to update the amounts:

```
function _updateOrder(uint256 positionId, address user, uint128 amount) internal
returns (uint128 userTotal) {
    BatchOrder storage order = orderBook[positionId];
    // Dedaub: this is the original direction!
    if (order.direction) {
        // token1
        order.token0Amount += amount;
    } else {
        // token0
        order.token1Amount += amount;
```

```
    }

    // Check if user is already in the order.
    uint128 batchId = order.batchId;
    uint128 originalDepositAmount = batchIdToUserDepositAmount[batchId][user];
    // If this is a new user in the order, add 1 to userCount.
    if (originalDepositAmount == 0) order.userCount++;
    batchIdToUserDepositAmount[batchId][user] = originalDepositAmount + amount;
    return (originalDepositAmount + amount);
}
```

Note, however, that `_updateOrder` reads the direction from `orderBook[positionId]`, which is the **upward** direction of the **original order**! As a consequence,instead of registering a deposit of $5k (else branch), it registers a deposit of 5000 ETH (if branch)! So now we have registered an order of 5100 deposited ETH, 98% of which seem to be deposited by the malicious user. The adversary can now call `performUpkeep` (or wait until it is automatically called) to fulfill the order, and finally call `claimOrder`, to collect 98% of the $205k.

Some further remarks:

- This issue can happen either accidentally or by a malicious user.
- To ensure that the old order is not fulfilled, the adversary can simply wait until the order is almost ITM (the price is close to $2k/ETH). Then, the adversary can perform a final swap to raise the price (making the order ITM) and execute the attack in the same transaction, before `performUpkeep` has any chance of being called.
- If the order contains sufficient funds, the adversary does not even have to wait. He could in fact tilt the pool (using a flash loan) to make the order ITM, provided that the stolen funds are sufficient to cover the cost of tilting the pool (swap fee and loan fee).

To prevent this issue the contract should correctly handle the case when a new order has the opposite direction than the previous one. For instance, it could fulfill the old order (which has to be ITM since the new order is OTM) before proceeding with the new one. We also recommend adding tests to cover the case when a new order has the same tick range as a previous one in the opposite direction.

HIGH SEVERITY:

[NO HIGH SEVERITY ISSUES]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	No staleness check for `LimitOrderRegistry::getGasPrice` oracle.	**OPEN**
The `LimitOrderRegistry::getGasPrice` function uses an oracle of type `IChainlinkAggregator` to obtain the current gas price. However no staleness check is made to check whether the oracle is returning the most recent values. It is recommended that the timestamp of the last value is checked if this is available and that a sensible default is returned if this exceeds a certain tolerance threshold. This is important since the result of `getGasPrice()` influences the fees charged to the users for checking whether the Uniswap V3 positions 'are in the money'. If this value is incorrect the bot calling `LimitOrderRegistry::getGasPrice` could end up paying for the upkeep out of pocket.		
M2	No guarantee that orders will eventually be fulfilled	**OPEN**

The only way to fulfill orders is via the `performUpkeep` function, which processes them via the linked list, with a limit of `maxFillsPerUpkeep` orders per call. Since a malicious adversary can add a large number of orders to that list, there is no guarantee that legitimate orders will eventually be processed.

A "minimum liquidity" requirement is used to protect against such attacks, however it is far from optimal for the following reasons:

- It is only based on financial incentives which are hard to analyze.
- An adversary needs a large capital to create many orders, by only the fees are actually spent, his funds are not at risk.
- There is a tradeoff between security and usability, since a large limit prevents users from creating small orders.

There is, however, a simple solution to guarantee that any order can be fulfilled: adding a public `fulfillOrder` function that can be called by any user to fulfill 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. This would offer a simple backup solution to any griefing attack on the list.

LOW SEVERITY:

ID	Description	STATUS
L1	Hard coded index in `TradeManager::PerformUpkeep`	**OPEN**
The `TradeManager::PerformUpkeep` function allocates MAX_CLAIMS claimInfo structs in memory, but then decodes a fixed amount of 10 claimInfo objects. The hardcoded value 10 should be replaced by MAX_CLAIMS to avoid the code breaking		

due to a future change in the MAX_CLAIMS value. The loop index of the loop following the allocation of these structs is also hardcoded and needs to be changed.

```
function performUpkeep(bytes calldata performData) external {
        // Accept claim array and claim all orders
        ClaimInfo[MAX_CLAIMS] memory claimInfo = abi.decode(performData,
(ClaimInfo[10]));
        for (uint256 i; i < 10; ++i) {
            if (limitOrderRegistry.isOrderReadyForClaim(claimInfo[i].batchId)) {
                (ERC20 asset, uint256 assets) = limitOrderRegistry.claimOrder{
value: claimInfo[i].fee }(
                    claimInfo[i].batchId,
                    address(this)
                );
                ownerOrders.remove(claimInfo[i].batchId);
                if (claimToOwner) asset.safeTransfer(owner, assets);
            }
        }
    }
```

L2	`LimitOrderRegistryLens::walkOrders` returns structs even when there are no orders to return	**OPEN**

The `walkOrders()` function of the `LimitOrderRegistryLens` contract will always return at least one struct, even when it should return no values.

This is because there is no check on whether the order at startingNode, centerHead or centerTail is actually in the linked list of orders.

The resulting BatchOrderViewData struct can have a zero or non-zero id, and a batchOrder with zero or non-zero data.

For instance, if the centerHead or centerTail is zero, the BatchOrderViewData struct will have a zero id and a zero batchOrder struct.

On the other hand, if startingNode is set, the BatchOrderViewData struct will have the startingNode as id, and whatever data is still held in the corresponding batchOrder (note that not all data is deleted from fulfilled BatchOrders).

This may or may not be a problem depending on the consumer of the result of the `walkOrders()` function, but this edge case is being flagged here just in case

```
function walkOrders(
        UniswapV3Pool pool,
        uint256 startingNode,
        uint256 returnCount,
        bool direction
    ) external view returns (BatchOrderViewData[] memory orders) {
        orders = new BatchOrderViewData[](returnCount);
        (uint256 centerHead, uint256 centerTail, , , ) =    registry.poolToData(pool);
        if (direction) {
            // Walk toward head.
            uint256 targetId = startingNode == 0 ? centerHead : startingNode;

            LimitOrderRegistry.BatchOrder memory target =
            registry.getOrderBook(targetId);

            // Dedaub - No check here on whether target actually exists in linked
                        list.

            for (uint256 i; i < returnCount; ++i) {
                orders[i] = BatchOrderViewData({ id: targetId, batchOrder: target });
                targetId = target.head;
                if (targetId != 0) target = registry.getOrderBook(targetId);
                else break;
            }
        } else {
            // Walk toward tail.
            uint256 targetId = startingNode == 0 ? centerTail : startingNode;

            LimitOrderRegistry.BatchOrder memory target =
            registry.getOrderBook(targetId);

            for (uint256 i; i < returnCount; ++i) {
                orders[i] = BatchOrderViewData({ id: targetId, batchOrder: target });
                targetId = target.tail;
                if (targetId != 0) target = registry.getOrderBook(targetId);
```

```
                else break;
            }
 }
    }
```

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

[NO CENTRALISATION ISSUES]

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Possible Optimisation in LimitOrderRegistry::cancelOrder	**OPEN**

In the cancelOrder function of the LimitOrderRegistry contract, whenever the cancellation of the order results in all liquidity being withdrawn, order.token0Amount and order.token1Amount are zeroed multiple times.

There is the possibility of a slight optimisation by zeroing both amounts when first checking whether orderAmount == depositAmount, and then not effecting any assignment after calling _removeOrderFromList.

```
function cancelOrder(
        UniswapV3Pool pool,
        int24 targetTick,
        bool direction
    ) external returns (uint128 amount0, uint128 amount1, uint128 batchId) {
        uint256 positionId;
        {

                            ...

            uint128 orderAmount;
            if (order.direction) {
                orderAmount = order.token0Amount;
                if (orderAmount == depositAmount) {
                    liquidityPercentToTake = 1e18;
                    // Update order tokenAmount.
                    order.token0Amount = 0;
     //Dedaub - add order.tokenAmount1 = 0; here

                } else {
                    liquidityPercentToTake = (1e18 * depositAmount) / orderAmount;
```

```
                    // Update order tokenAmount.
                    order.token0Amount = orderAmount - depositAmount;
                }
            } else {
                orderAmount = order.token1Amount;
                if (orderAmount == depositAmount) {
                    liquidityPercentToTake = 1e18;
                    // Update order tokenAmount.
                    order.token1Amount = 0;
      //Dedaub - add order.tokenAmount0 = 0; here

                } else {
                    liquidityPercentToTake = (1e18 * depositAmount) / orderAmount;
                    // Update order tokenAmount.
                    order.token1Amount = orderAmount - depositAmount;
                }
            }

            (amount0, amount1) = _takeFromPosition(positionId, pool,
liquidityPercentToTake);
            if (liquidityPercentToTake == 1e18) {
                _removeOrderFromList(positionId, pool, order);
                // Zero out balances for cancelled order.

                //Dedaub - the following two lines can now be deleted

                order.token0Amount = 0;
                order.token1Amount = 0;

                order.batchId = 0;
            }
        }
                        ...
    }
```

A2	Redundant use of the upper tick in LimitOrderRegistry::getPositionFromTicks	**OPEN**

In the `LimitOrderRegistry` contract, getPositionFromTicks is a mapping of the form:

```
mapping(UniswapV3Pool => mapping(int24 => mapping(int24 => uint256))) public
getPositionFromTicks;
// maps pool -> lower -> upper -> positionId
```

Note that:

- The upper tick is always equal to lower + spacing (and spacing is constant for each pool)
- The position does not depend on the direction.

As a consequence, it seems impossible to have two positions in the mapping with the same lower and different upper ticks, hence using the upper tick is redundant. Removing it would make the code simpler and save some gas.

A3	Inconsistent version pragmas	**OPEN**

The contracts under audit use three different compiler version pragmas: >=0.8.0, ^0.8.10 and ^0.8.16. It is recommended that a uniform compiler version pragma is used for the contracts to avoid code inconsistencies which can lead to security issues.

A4	Floating version pragma in contracts	**OPEN**

Use of floating version pragmas such as >= or ^ allow contracts to be compiled with different versions of the Solidity compiler. Even though versions might not differ drastically, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment so as to avoid unexpected behaviour.

A5	Compiler version and possible bugs	**OPEN**

The code can be compiled with different versions of the Solidity compiler. According to the foundry.toml file of the codebase, version 0.8.16 is currently used which has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.