UniswapX Audit

OPENZEPPELIN SECURITY | OCTOBER 10, 2023

Security Audits

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2023-07-31
: To 2023-08-11

Languages
: Solidity

Total Issues
: 16 (10 resolved, 1 partially resolved)

Critical Severity Issues
: 0 (0 resolved)

High Severity Issues
: 0 (0 resolved)

Medium Severity Issues
: 2 (1 resolved)

Low Severity Issues
: 8 (6 resolved)

Notes & Additional Information
: 6 (3 resolved, 1 partially resolved)

Client Reported Issues
: 0 (0 resolved)

Scope

We audited the `Uniswap/UniswapX` repository at the 7c5e359 commit.

```
|   ├─ ProtocolFees.sol
|   ├─ ReactorEvents.sol
|   └─ ReactorStructs.sol
├─ external/
|   └─ ISwapRouter02.sol
├─ interfaces/
|   ├─ IProtocolFeeController.sol
|   ├─ IReactor.sol
|   ├─ IReactorCallback.sol
|   └─ IValidationCallback.sol
├─ lens/
|   └─ OrderQuoter.sol
├─ lib/
|   ├─ CurrencyLibrary.sol
|   ├─ DutchDecayLib.sol
|   ├─ DutchOrderLib.sol
|   ├─ ExclusiveDutchOrderLib.sol
|   ├─ ExclusivityOverrideLib.sol
|   ├─ LimitOrderLib.sol
|   ├─ OrderInfoLib.sol
|   ├─ Permit2Lib.sol
|   └─ ResolvedOrderLib.sol
├─ reactors/
|   ├─ BaseReactor.sol
|   ├─ DutchOrderReactor.sol
|   ├─ ExclusiveDutchOrderReactor.sol
|   └─ LimitOrderReactor.sol
├─ sample-executors/
|   └─ SwapRouter02Executor.sol
└─ sample-validation-contracts/
    └─ ExclusiveFillerValidation.sol
```

System Overview

orders detailing their swap requirements, while participants known as fillers employ arbitrary strategies to fulfill these orders by competing with each other.

Orders

UniswapX leverages `Permit2`, a token approval contract introducing signature-based approvals and transfers for ERC-20 tokens without the need for them to be compliant with EIP-2612. Swappers must approve the `Permit2` contract first for each token, but this only needs to be done once. After that, instead of self-submitting transactions, they sign orders which are shared via API to a network of fillers (MEV searchers, market makers, or other on-chain agents). Fillers send them to a specific reactor contract, which settles orders ensuring that trade execution aligns with user expectations and reverts non-compliant trades. The gas-less experience for the swapper is achieved because the fillers are the ones submitting orders on-chain, covering the gas fees on their behalf. This cost is recouped, in addition to a small profit on the trade itself, by factoring it into the execution price.

UniswapX incorporates the logic to manage three distinct order types by providing three different reactor contracts, each of which presents its own unique fulfilment rules.

Reactors

Reactor contracts share the main logic, but they differ in the type of orders they are managing and the validations they perform. All of them have four entry points, so that fillers can have enough flexibility to execute single orders or batches of orders in a sequential fashion. In both cases, fillers can also decide whether they want to get a callback in order to perform arbitrary strategies to get the funds on-chain. Alternatively, fillers have the option of filling orders directly with no callbacks. Either way, fillers need to grant prior approval to the reactor contract, as it will transfer the input tokens to them and pull the output tokens directly from their account to the swapper's address.

Prior to the input token transfer, a validation takes place ensuring the order has not expired and addresses the correct reactor. Further custom validation can be enabled by the swapper through an additional callback.

Dutch orders are designed to optimize swapper execution price versus simply choosing an AMM, by closely emulating the mechanics of a Dutch auction. The intrinsic feature of Dutch orders lies in the execution price decaying over time, fostering a competitive environment among fillers to swiftly secure the most favorable price for swappers while maintaining a modest profit margin. Fillers are incentivized to fill these orders as soon as it is profitable for them to do so. If a filler decides to wait for the decay to grow larger in order to make a larger profit, another filler may jump in and fill the order first for a more modest profit.

Exclusive Dutch Order Reactor

An exclusive Dutch order introduces a temporary exclusivity period before the linear decay of the execution price begins. This innovative mechanism provides a specific filler with a timeframe to engage with the order without any decay nor any other competitor being able to fill the order at that price. However, in order to always respect the swappers' best interest, there are settings in place to enable any other filler to jump in and provide a better price, provided that this price improvement is large enough.

Once the exclusivity period concludes, the order seamlessly transitions into a standard linear-decay phase as if it was a standard Dutch order.

Limit Order Reactor

In contrast to the dynamic nature of Dutch orders, a limit order represents a straightforward approach to trading. This type of order requires a specific amount of output tokens to be transferred, ensuring the order is fulfilled successfully.

Fillers are incentivized to fill limit orders as soon as possible, as long as the requested execution price makes sense for them since the price is not going to get better over time.

Order Parametrization

Determining the order price and decay function of a Dutch order presents a difficult challenge for swappers. Setting the price too high might dissuade fillers from executing their order due to surpassing the prevailing market price. Conversely, setting the price too low could attract a filler but

To guide swappers in this order parametrization process, UniswapX offers an off-chain system called Request For Quote (RFQ). This system polls a group of selected quoters (which may also be fillers) to suggest a competitive quote for the swapper's request. The winning quoter temporarily obtains exclusive rights to fill the order via the exclusive Dutch order reactor.

Quoters are expected not to misbehave and always provide the best possible quotes for the end user. During the current beta phase, quoters have to be vetted by Uniswap Labs. In the future, this system will become permissionless and will operate under a system of rewards and penalties to make sure traders' interests are always the first priority.

Privileged Roles

For each type of order supported by UniswapX, Uniswap Labs will deploy a reactor contract. The reactor contracts enable fillers to execute one or many signed orders of the same type in a single batch. Fillers and swappers incur a fee per order, as enforced by the `ProtocolFees` contract, which will be owned by Uniswap Governance. The owner has the ability to set the `FeeController` address. The `ProtocolFees` contract queries the `FeeController` with every order for the fees to be charged, up to 0.05% of the total order value. Note that if a reactor is initialized without a `ProtocolFees` owner, or if ownership is renounced at some point, the reactor will be unable to change the `FeeController`.

Trust Assumptions

The current version of the UniswapX RFQ system operates through a closed vetted group of quoters, who are not expected to misbehave. This helps maintain a good swapper experience and prevent any potential misuse of the system. Users trust the RFQ system to provide them with sensible parameters for their orders.

Swappers and fillers trust the Uniswap API, where orders are exchanged, that all information is accurate and no malicious data can be injected by any party.

Medium Severity

Fillers Can Incur a Loss Without a Chance to Revert

equivalently does not send right after the callback. Instead, once the callback is over, the output balances are transferred from the filler to the swapper by the reactor.

If one of the output tokens is either a malicious ERC-20 implementation controlled by the swapper or a legitimate ERC-777 implementation, the swapper can receive one more callback within the `transferFrom` function call to perform arbitrary operations. These operations might be gas intensive and will be funded by the filler. At this point, the filler has no more execution control. As such, given a high enough overall transaction gas limit, even if a profit was made on the trade, the filler could incur a loss.

A malicious swapper might use this technique to perform a legitimate swap as well as let the filler subsidize a potentially gas-intensive operation (e.g., minting gas tokens on applicable chains, performing more swaps, deploying contracts, etc).

Consider adding a final callback to the filler to allow reverts if a profit was not made after fees. This will turn this vector into a griefing attack since the filler will be charged for the revert, but at least the swapper will not profit from the filler. In order to make the extra callback more gas efficient, consider either making it optional or having the filler call an external version of `_fill` as the last step on their callback. It is imperative to ensure the filler has called the `_fill` function, but this must be done without checking the balances of the recipients.

Update: *Acknowledged, not resolved. This issue is only applicable if the filler is using an EOA to target the various* `execute` *entrypoints directly (without using an intermediate contract of their own) and / or not using a private mempool to relay their transactions in order to avoid paying for reverts. The Uniswap team stated:*

> *Fillers generally estimateGas on their transaction before including them anyways, and they will only send it if it is profitable net of gas. They also generally use private mempools like flashbots or mevblocker to get protection from loss due to gas. Further,* they do have the chance to revert after the order is filled in their `fillContract`, after the execution is finished.

Gas Limitation Can Disable Smart Contract Wallets

The `CurrencyLibrary` is used to handle ERC-20 tokens and native currency transfers in UniswapX. The native transfer is performed with a low-level call without calldata and a hard-coded gas limit of 6900.

- EVM side-chains and L2s can have different gas costs. This has previously caused funds to become stuck as transfers run out of gas and revert.

Both arguments could lead to unreliable functionality over time and across chains. A smart contract wallet can have logic in its `receive` or `fallback` function which exceeds the 6900 gas limit, thereby disabling that wallet to use the service for swaps where the native currency is involved. Furthermore, the protocol injects fees into the orders that are sent to the fee recipient, including ETH. Hence, a fee vault or a multi-sig wallet as a fee recipient can be equally affected by this problem.

Note that the concern of reentrancy should be tackled at the entry point level, which was not identified as a risk for this scope. Further, for the concern regarding the swapper being able to spend the filler's gas maliciously, please see *M-01*.

Consider removing the gas limitation to guarantee the functionality of smart contract wallets when engaging with UniswapX.
Update: *Resolved in pull request #189 at commit 0cdf97e.*

Low Severity

Floating Pragma

The contracts in the codebase are using the floating pragma `^0.8.0` . This version indicates that any compiler version starting from `0.8.0` can be used to compile the source code.

However, the code will not compile when using version `0.8.3` or earlier since the protocol is using custom errors that were introduced in Solidity `0.8.4`. In addition, the use of Solidity `0.8.20` and later might cause a problem with a deployment to Arbitrum and Optimism, which do not yet support opcode `PUSH0` that has been introduced with Solidity version `0.8.20`.

It is recommended to set the pragma to `0.8.19` to align with the best practice of locking the pragma for all contracts. This precautionary measure also helps prevent the accidental deployment of contracts with outdated compiler versions that could potentially introduce vulnerabilities.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

The protocol allows fees to be collected. To do so, a protocol fee controller needs to be set to a non-zero address. Uniswap governance has the ability to collect up to 0.05% of the token amounts, as enforced by the `ProtocolFees` contract. The `FeeController` is queried by the `ProtocolFees` contract to get a list of fees that need to be taken, specifying the amounts, the token addresses, and the recipients.

Any mistakes in that return value will cause the `_injectFees` function to revert if:

- The fee token is not part of the original order.
- The fee is too large.
- There is a duplicated entry.

Since the fees are fetched for every order from any reactor, an incorrect response will halt all trading activity. This can be the result of an attack or a bug in the fee controller.

If this is not intended behavior, consider not charging fees when one of the above cases occurs in order to not affect trading activity. Instead of reverting, an event can be emitted and be caught by a monitoring solution.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *This is a known issue, and we acknowledge it for now. We can always redeploy a reactor in such circumstances, which would only cause a minor inconvenience. We will likely decide on a fix before the next version is deployed.*

Dutch Orders Without Duration Benefit the Filler

The Dutch order implements a linear decay function. From `decayStartTime` to `decayEndTime` the amount of input or output tokens goes from `startAmount` (best rate for swapper) to `endAmount` (worst rate the swapper agrees with).

However, in the current implementation, it is possible to provide the same value for the `decayStartTime` and `decayEndTime` . In this case, the Dutch order is acting like a limit

Consider changing the price of Dutch orders without duration to be equal to the `startAmount` instead of the `endAmount` by swapping the two `else-if` blocks checking that the timeframe is in favor of the swapper. Alternatively, consider restricting the usage of Dutch orders to actual Dutch orders with a non-zero duration and price decay.

Update: *Resolved in pull request #194 at commit 068076f. Dutch orders with zero duration are now disallowed to avoid this ambiguity.*

Missing or Incorrect Docstrings

Throughout the codebase, there are several parts that have missing or incorrect docstrings:

- Missing docstrings for the `ResolvedOrderLib` library.
- Missing docstrings for the `ExclusiveFillerValidation` contract.
- Incorrect order of parameters on line 11-12 in `ReactorEvents.sol` . The `swapper` parameter should be described before `nonce`.
- Missing description of `callbackData` in `IReactor.sol`.
- The `parseRevertReason` function misses the `reason` parameter's docstring. The `@param order` docstring should be a `@return` stating it is the resolved order.

Consider thoroughly documenting all functions and their parameters that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #192 at commit ce6b1f5.*

Inaccurate Documentation

There are multiple areas where the protocol would benefit from more accurate documentation:

- The `_validateOrder` function documentation of the `DutchOrderReactor` and `ExclusiveDutchOrderReactor` contracts describe order properties that must be fulfilled. However, these functions only validate a subset of those properties.
- The documentation of the `checkExclusivity` function does not mention the timestamp impact on the exclusivity. Furthermore, the function name does not fully align with

check.

- The Fees section of the whitepaper states that a fee is only taken on the output tokens of the swap, while the code also allows a fee on the input token.
- The documentation on the website elaborating on Direct Fill appears to be outdated.

Consider updating the documentation to be aligned with the intentions of the code.
Update: *Resolved in pull request #193 of the UniswapX repository at commit cdd4e33 and pull request #626 of the docs repository at commit 24d90ab.*

Incomplete Test Coverage

While the overall coverage is close to 100%, there are several lines of code lacking test coverage:

- In the `ExclusiveDutchOrderReactor` contract, the validation logic is not tested for orders with input decay. Consider adding at least one succeeding and one reverting test case with input decay to the `ExclusiveDutchOrderReactor` test file.
- In the `ProtocolFees` contract, the `_injectFees` function is not tested for input fees.

To further enhance the quality of the tests, here are some suggested improvements:

- Add tests of `executeBatch` for `ExclusiveDutchOrders` (already present for `DutchOrder` and `LimitOrder`).
- Add integration tests for limit orders and exclusive Dutch orders since currently only the Dutch order reactor is being integration-tested.

Update: *Resolved in pull request #197 at commit e327540 by adding tests for input fees and exclusive Dutch auctions with decaying input. The Uniswap team explained that additional integration tests are expected to be added in the future.*

Redundant Code

Redundant code reduces the protocol's gas efficiency, especially when considering on-chain trading, as it impacts the filler's profit and the swapper's execution price. The following opportunities have been identified for potential enhancements:

trades more gas-optimized while reverting trades will use a bit more gas.

- The exclusive Dutch order reactor does not have to validate that `endDecayTime` is larger than or equal to `startDecayTime`, since it will be validated again when applying the `decay`.
- The `CurrencyLibrary` implements a `balanceOf` function that is not used throughout the protocol. Consider removing unused code to improve the readability.

Consider applying the suggested changes in order to make the code more efficient. Note that for the removed checks, it is important to document where the actual check is performed.
Update: *Resolved in pull request #195 at commit 36a0a28. The `balanceOf` function of the `CurrencyLibrary` was not removed since it might be used by the fill contracts.*

Magic Numbers

In the `OrderQuoter` contract, there are occurrences of literal values with unexplained meanings:

- `ORDER_INFO_OFFSET = 64`, which is used to skip a bytes length and struct offset slot to point to the offset slot of the inner `OrderInfo` struct. In addition, it lacks an explicitly declared visibility.
- `reason.length < 192`, which checks whether `reason` is less than 6 slots to distinguish a revert reason from an encoded order. Note that the `OrderInfo` struct has a minimal encoded length of 8 slots to account for the struct and dynamic bytes offsets.

Consider documenting these numbers to improve the readability of the codebase.

Update: *Resolved in pull request #196 at commit 2b881d4.*

Notes & Additional Information

Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as the function's output. They are an alternative

- `order` within the `parseRevertReason` function.
- `pass` within the `checkExclusivity` function.

Consider either using or removing any unused named return variables.

Update: *Resolved in pull request #199 at commit 5e5ef41. Both unused named return variables were removed.*

Unused Imports

Throughout the codebase there are imports that are unused and could be removed. For instance:

- Imports `ResolvedOrder` and `IReactorCallback` of `IReactor.sol`
- Import `OrderInfo` of `IValidationCallback.sol`
- Imports `BaseReactor` and `OrderInfo` of `OrderQuoter.sol`
- Imports `IPermit2` and SafeCast of `CurrencyLibrary.sol`
- Import `ERC20` of `Permit2Lib.sol`
- Import `IValidationCallback` of `ResolvedOrderLib.sol`
- Import `NATIVE` of `BaseReactor.sol`
- Imports `InputToken`, `OutputToken`, and `OrderInfo` of `DutchOrderReactor.sol`
- Import `OrderInfo` of `ExclusiveDutchOrderReactor.sol`
- Imports `OrderInfo`, `OutputToken`, and `InputToken` of `LimitOrderReactor.sol`
- Import `OutputToken` of `SwapRouter02Executor.sol`
- Import `OrderInfo` of `ExclusiveFillerValidation.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #191 at commit ae17cd3.*

Typographical Errors

Consider addressing the following typographical errors:

Update: *Resolved in pull request #190 at commit 19307ff.*

Potential Double Fees on Cross-Chain Swaps

Every swap through UniswapX can charge a fee on the input and output tokens. As per the documentation, this fee must not exceed 0.05% "of the order outputs". This is checked by accumulating the input and output token amounts per token address and calculating whether the fee amount is within the 0.05% threshold.

When swapping a token for itself, such as when bridging it from one chain to another (assuming the same token address and value), the token amount would be doubled through the accumulation. Hence, a double fee could be erroneously charged.

Consider handling this specific case such that only the input or the output amount is taken into account when calculating the fee.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *These contracts are not intended for cross-chain swaps. A cross-chain reactor will likely also include* `chainId` *specification for tokens to differentiate the same token across chains.*

Gas Optimizations

There are some areas where gas savings can be achieved in order to improve the system's performance. For instance:

- There are multiple `for` loops throughout the codebase that use the postfix increment operator. Consider using the prefix increment operator `++i` instead in order to save gas. This skips storing the value before the incremental operation, as the return value of the expression is ignored.
- When filling single orders, consider removing the overhead of turning a single order into a 1-item array of orders which will be looped through when preparing and filling it. The main preparation and filling logic could be extracted into an internal function called from inside the loop when working with batches, and individually called from single-order executions.
- Validating that the transaction is targeting the proper reactor and has not yet expired could be performed as soon as possible in the transaction lifecycle. This will save some gas by failing

length of an array to stack and reuse the stack variable. Multiple instances were identified where this optimization could be applied. For instance:

- On line 87 of `DutchOrderLib.sol`
- On line 43 of `ExclusivityOverrideLib.sol`
- On line 50 of `LimitOrderLib.sol`
- On line 71 of `DutchOrderReactor.sol`
- On line 81 of `ExclusiveDutchOrderReactor.sol`
- On line 75 of `SwapRouter02Executor.sol`
- On line 79 of `SwapRouter02Executor.sol`

To improve gas consumption, readability and code quality, consider refactoring wherever possible, carefully reviewing the entire codebase and making sure the test suite still passes.

Update: *Partially resolved in pull request #198 at commit b0d62e3. The prefix increment operator is now being used. The second suggestion was disregarded in favor of code readability. The change of validation logic was not applied because the Uniswap team preferred the current execution flow and they did not want to prioritize one check over the other. The recommendation of reading array lengths from the stack rather than from memory when looping was not applied since on the current test suite, given the low expected iterations on these loops, it showed an increase in gas usage rather than actual savings.*

Usage of Insecure Approval Functions

Within the sample executor contract, there are instances of insecure approvals that may lead to security issues.

For this specific implementation, it does not pose a security risk. However, as this contract is intended to be used as the starting point for fillers, consider implementing best practices by using the `SafeERC20` contract's `forceApprove function` in order to avoid having fillers run into issues when dealing with non-standard ERC-20 implementations.

Update: *Acknowledged, not resolved. The recommendation is not applicable to the current implementation of the sample executor contract.*

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production, the Uniswap team is encouraged to consider incorporating monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps identify potential threats and issues affecting production environments. With the goal of providing a complete security assessment, the monitoring recommendations section raises several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Access Control

Medium: Regarding issue *L-02*, consider monitoring the `ProtocolFeeControllerSet` event to track when `FeeController` address has been changed unexpectedly. While this means that the account has been compromised, in the audited version a malicious change could manipulate the returning fees and thereby halt the trading activity.

Technical

Low: Regarding the recommendation of *L-02*, if an event is implemented as a reaction to a wrong fee, consider monitoring it to be alerted about bugs in the `FeeController`.

Suspicious Activity

Medium: Consider monitoring the parametrization of orders for outlier detection. Activity outside the norm, such as a rapid linear decay or no duration (see *L-03*), can indicate malicious intent by the quoter to trick the swapper in signing an adverse order to make an exceptional profit. Note that due to the inherent filler competition this is rather applicable for exclusive Dutch orders.

Low: As of now, the allowed order reactors in UniswapX need to be explicitly enabled. In the future, this restriction may be lifted, allowing for arbitrary reactor contracts to be used. Consider monitoring that these reactors do not pose a threat for swappers or fillers if they are injected maliciously.

Conclusion

lowest possible fees, the Uniswap team took great care in optimizing every code section without compromising security by following best practices. The system correctly implements the logic for filling signed orders and performs all required checks to ensure the safety of swappers and fillers. Nevertheless, some improvements have been proposed to further adhere to best practices and enhance gas efficiency. Throughout the audit, the Uniswap team has been very helpful with the questions raised by the audit team.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin