UniswapX
Security Review

Cantina Managed review by:
Desmon Ho, Lead Security Researcher
Jeiwan, Security Researcher
Jonatas Martins, Associate Security Researcher

May 8, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The UniswapX Protocol is a non-custodial trading structure that leverages Dutch auctions. It integrates on-chain and off-chain liquidity, providing a seamless trading experience. This unique approach also shields swappers from MEV by incorporating them into the price enhancement process, leading to gas-less swaps. Trading on UniswapX involves creating signed orders that outline specific swap conditions. Fillers, or participants, then utilize various strategies to compete with one another and fulfill these orders.

From Mar 12th to Mar 21st the Cantina team conducted a review of UniswapX on commit hash abd7a0b0. The team identified a total of **15** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 2
- Low Risk: 6
- Gas Optimizations: 3
- Informational: 3

3 Findings

3.1 High Risk

3.1.1 Protocol fees taken on input and output tokens are not mutually exclusive

Severity: High Risk

Context: ProtocolFees.sol#L81-L84

Description: When injecting protocol fees into an order, the protocol is allowed to take fees on the outputs (ProtocolFees.sol#L69-L79) or the input (ProtocolFees.sol#L81-L84) of an order. However, taking fees on the outputs and the input is not mutually exclusive: it's possible to take fees on the input and on one or multiple outputs of an order at the same time. As a result, protocol fees can be charged on an order twice.

Consider the following example:

1. The fee controller is configured to charge fees in WETH and USDC tokens.
2. A user submits an order that swaps WETH for USDC.
3. The fees injected into the order during execution will be taken in both WETH and USDC.
4. As a result, the order will pay double protocol fees on its value.

Recommendation: Consider disallowing taking protocol fee on the input of an order if a fee is also collected from one of its outputs. This can be achieved by modifying the ProtocolFees._injectFees() function so that a fee is applied to the input of an order only if there's no fees on its outputs. Additionally, fees shouldn't be applied to outputs if a fee has already been applied to the input.

Uniswap: Fixed in commit PR 247.

Cantina Managed: Fixed.

3.2 Medium Risk

3.2.1 Cosigner signature verification can be bypassed

Severity: Medium Risk

Context: V2DutchOrderReactor.sol#L126-L129

Description: When validating a Dutch auction order and verifying the cosigner's signature, there's no check for an invalid signature (V2DutchOrderReactor.sol#L126-L129):

```
address signer = ecrecover(keccak256(abi.encodePacked(orderHash, abi.encode(order.cosignerData))), v, r, s);
if (order.cosigner != signer && signer != address(0)) {
    revert InvalidCosignature();
}
```

The ecrecover function returns the zero address when fails to recover the address from an invalid signature. However, the `V2DutchOrderReactor._validateOrder()` function doesn't revert if the `signer` address is zero.

As a result, a malicious filler can spoof the signature and bypass the cosigner verification to execute an order with arbitrary cosigner data. For example, a malicious filler can bypass the exclusivity check and/or avoid paying the non-exclusive filler fee and/or always execute orders at the worst price (i.e. swap end input amounts for end output amounts).

Proof of concept:

```
// test/reactors/V2DutchOrderReactor.t.sol
function testOverrideInput_wrongSignature_AUDIT() public {
    uint256 outputAmount = 1 ether;
    uint256 overriddenInputAmount = 0.7 ether;
    tokenIn.mint(swapper, overriddenInputAmount);
    tokenOut.mint(address(fillContract), outputAmount);
    tokenIn.forceApprove(swapper, address(permit2), type(uint256).max);
    CosignerData memory cosignerData = CosignerData({
        decayStartTime: block.timestamp,
        decayEndTime: block.timestamp + 100,
        exclusiveFiller: address(0),
        exclusivityOverrideBps: 0,
        inputAmount: overriddenInputAmount,
        outputAmounts: ArrayBuilder.fill(1, 0)
    });

    V2DutchOrder memory order = V2DutchOrder({
        info: OrderInfoBuilder.init(address(reactor)).withSwapper(swapper),
        cosigner: address(0x31337), // @audit arbitrary cosigner address
        baseInput: DutchInput(tokenIn, 0.8 ether, 1 ether),
        baseOutputs: OutputsBuilder.singleDutch(address(tokenOut), outputAmount, outputAmount, swapper),
        cosignerData: cosignerData,
        cosignature: bytes("")
    });
    order.cosignature = bytes.concat(keccak256("spearbit"), keccak256("spearbit"), hex"33"); // @audit wrong
↪   signature
    SignedOrder memory signedOrder =
        SignedOrder(abi.encode(order), signOrder(swapperPrivateKey, address(permit2), order));

    _snapStart("InputOverride");
    fillContract.execute(signedOrder);
    snapEnd();

    assertEq(tokenIn.balanceOf(swapper), 0);
    assertEq(tokenOut.balanceOf(swapper), outputAmount);
    assertEq(tokenIn.balanceOf(address(fillContract)), overriddenInputAmount);
}
```

Recommendation: In the cosigner signature verification, consider reverting if the recovered address is zero.

Uniswap: Fixed in PR 243.

Cantina Managed: Fixed.

3.2.2 An invalid order can revert the execution of an entire batch

Severity: Medium Risk

Context: BaseReactor.sol#L102-L103, ResolvedOrderLib.sol#L18-L20, V2DutchOrderReactor.sol#L76

Description: When executing a batch of orders via the BaseReactor.executeBatch() or BaseReactor.executeBatchWithCallback() functions, each order is prepared before being filled. For each order in the batch, the _prepare() function calls:

1. ResolvedOrderLib.validate(), which calls the `additionalValidationContract` contract address specified by the order creator.
2. V2DutchOrderReactor.transferInputTokens(), which transfers tokens from the order creator via the Uniswap's `Permit2` contract.

Both of these calls can revert:

1. `additionalValidationContract.validate()` is expected to revert when the order creator rejects an invalid order.
2. `permit2.permitWitnessTransferFrom()` can revert when a permit has already been used or has expired.

In both of these cases, the execution of the entire batch will revert. While the call to `permit2.permitWitnessTransferFrom()` can revert randomly and occasionally (e.g. due to expired or used

permits), the call to `additionalValidationContract.validate()` can be deliberately abused to conduct a griefing attack on the protocol. Consider the following scenario:

1. A malicious actor submits multiple small orders.
2. In each of the orders, the `additionalValidationContract` address is specified.
3. The contract at the `additionalValidationContract` address implements a validate() function that reverts conditionally or unconditionally (e.g. it can revert only when detects the actual execution of the order).
4. Since batching is an important mechanism that allows fillers to reduce gas expenses and make swaps more profitable for traders, the malicious orders will be batched together with real orders.
5. As a result, the execution of multiple real orders can be intentionally blocked, causing a denial of service.

Recommendation: In the order preparation step, consider skipping orders that:

1. Reverted in the validation call;
2. or, failed to transfer input tokens to the filler.

Uniswap: I see your point here, and definitely agree that it would be nice to avoid full batch revert from a single revert. I have two main concerns, and think we will not do it this time around due to them but will consider for future upgrades:

- This may be quite disruptive to current/existing fillers.. They currently have a guarantee that if they pass in Orders 1, 2, 3 then they will either have to fill all 3 or none. Due to this guarantee, I think some of them may skip additional validation of the current ResolvedOrders during callback phase which could cause them to over-allocate assets assuming they are filling an order that in fact they are not.
- By squashing out revert messages with a try/catch or similar, off-chain tracking of orders becomes more difficult. For example, the OrderQuoter system is used to track the validity of an order over time, and try/catching the permit2 call and validation call make the results less granular.

Cantina Managed: Acknowledged.

3.3 Low Risk

3.3.1 Order amounts overwriting can cause decaying of both input and outputs

Severity: Low Risk

Context: V2DutchOrderReactor.sol#L131-L138, V2DutchOrderReactor.sol#L91, V2DutchOrderReactor.sol#L104

Description: Dutch auction orders' amounts can decay: the input amount can increase over time (DutchDecayLib.sol#L97-L101), or an output amount can decrease over time (DutchDecayLib.sol#L60-L64). As per the validation logic: if the input of an order decays, its outputs cannot decay (V2DutchOrderReactor.sol#L131-L138):

```
if (order.baseInput.startAmount != order.baseInput.endAmount) {
    for (uint256 i = 0; i < order.baseOutputs.length; i++) {
        DutchOutput memory output = order.baseOutputs[i];
        if (output.startAmount != output.endAmount) {
            revert InputAndOutputDecay();
        }
    }
}
```

However, this rule can be violated by a cosigner: cosigners are allowed to override the start amount of the input or the outputs of an order (V2DutchOrderReactor.sol#L87-L106). For example, the start input amount of an order with decaying outputs can be reduced by a cosigner, which makes the order invalid according to the V2DutchOrderReactor._validateOrder() function, but the violation won't be detected.

Recommendation: When validating a Dutch auction order, consider doing the decay validation after the cosigner specified amounts were applied to the order, not before it.

Uniswap: Thinking on this more, I'm not sure input and output decay is actually a problem, just a non-standard flow. Fixed in PR 245.

Cantina Managed: Fixed by removing the one-sided decay requirement.

3.3.2 Token transfer may revert if `feeController` uses a zero address as recipient

Severity: Low Risk

Context: ProtocolFees.sol#L91

Description: In the `ProtocolFees` contract, the `feeOutputs` are returned by the `feeController`. If there is a misconfigured `feeController` that sends an output recipient as address zero, it might revert for some tokens. Example: in implementation of OpenZeppelin ERC20.

Recommendation: It's recommended to not include the output in case the recipient's address is zero.

Uniswap: Thanks for this; not doing for now as there are many ways to create invalid orders, this covers just one of many - fillers can validate this on their own offchain

Cantina Managed: Acknowledged.

3.3.3 Inconsistent rounding directions

Severity: Low Risk

Context: DutchDecayLib.sol#L41-L45. ExclusivityLib.sol#L45

Description: The `decayedAmount` calculated for both decaying input and output rounds down. This leads to inconsistent behaviour: when specifying decaying inputs, it rounds in favour of the swapper, while for decaying output, in favour of the filler. In `ExclusivityLib`, the output amount rounds down as well (in favour of the filler).

Recommendation: Consider keeping rounding behaviour consistent, to be rounding in favour of either the swapper or the filler.

Uniswap: Fixed in PR 246.

Cantina Managed: Fixed in `DutchDecayLib`.

3.3.4 Cosigner can prematurely decay the order's dutch auction

Severity: Low Risk

Context: V2DutchOrderReactor.sol#L119-L121

Description: The following variants on the order timings are upheld:

```
order.cosignerData.decayStartTime < order.cosignerData.decayEndTime <= order.info.deadline
```

Notice that there are no comparisons with `block.timestamp`. Hence, the cosigner is able to set `decayEndTime` to be far in the past to extract value from the order, where the user either pays the maximum input / receives minimum output.

Recommendation: Consider adding an additional field into the `V2DutchOrder` struct that specifies the auction time start for the user to sign, and ensure that `order.cosignerData.decayStartTime` is greater than or equal to this value.

Uniswap: Acknowledged on this; we ended up not making this change, but understand the trust assumptions of the cosigner.

Cantina Managed: Acknowledged.

3.3.5 Protocol fee collection can negatively impact user transactions

Severity: Low Risk

Context: ProtocolFees.sol#L64, ProtocolFees.sol#L86, ProtocolFees.sol#L89, CurrencyLibrary.sol#L50-L51

Description: Protocol fee computation is delegated to an external contract that's called in the _injectFees() function when fees are injected into an order. The function then validates the computed fee amounts returned by the contract and:

1. Reverts when fee tokens are duplicated (ProtocolFees.sol#L64).
2. Reverts when the fee is taken in a wrong token (ProtocolFees.sol#L86).
3. Reverts when fee amount is too big (ProtocolFees.sol#L89).

Each of these cases will revert the order execution transaction since protocol fees injection is part of the order execution process (BaseReactor.sol#L101). As a result, if the protocol fee controller contract is misconfigured and/or there's a bug in the contract that results in wrong protocol fee amounts, all order execution transactions will be impacted and reverted, causing a denial of service for all protocol users.

In addition to the above scenarios, protocol fee collection can revert when a fee is taken in the native token and the fee recipient contract cannot receive native tokens. This will cause a revert in the CurrencyLibrary.transferNative() function when attempting to send native coins to a protocol fee recipient address (BaseReactor.sol#L120).

Recommendation: Consider reworking the protocol fee collection process so that any misconfigurations or potential bugs in the fee controller contract and/or fee recipient contracts cannot negatively impact order execution. For example, protocol fee collection can be skipped instead of reverting transactions, and a misconfigured or bugged fee controller can be detected using off-chain monitoring and analysis tools.

Uniswap: I see the argument here, but generally not too concerned about rogue protocol fee controllers causing reverts here. Worst case scenario given the usage of UniswapX is we redeploy a new reactor w/ no loss of user funds, only mild inconvenience. Protocol fee controller developers are not incentivized to do these DOS-style attacks as it means they don't get fees

Cantina Managed: Acknowledged.

3.3.6 Zero-value transfers might result in transaction revert

Severity: Low Risk

Context: V2DutchOrderReactor.sol#L114-L139, ProtocolFees.sol#L91

Description: The `V2DutchOrderReactor` contract should validate the order token amounts through the `_validateOrder()` function. However, this function does not verify the `baseInput` and the `baseOutput` amounts. If any token amount transferred in or out for the swapper is zero, the transfer could fail losing gas from `Filler`.

Similarly, in the `ProtocolFees` contract, there is no validation for `feeOutput.amount` being different from zero. A misconfigured controller could cause the transaction to fail.

Recommendation: It is recommended to verify that `baseInput` and `baseOutput` are not zero before executing transfers in `V2DutchOrderReactor`. Additionally, `ProtocolFees` should not include `feeOutput` if the amount is zero.

Uniswap: There are many types of misconfiguration of orders that could cause reverts -- amounts larger than balance, unapproved tokens, etc... I'm not sure this protection is worth it/necessary given fillers are already expected to validate an order's validity before submission.

Cantina Managed: Acknowledged. Agreed that is expected to validate the order before submission.

3.4 Gas Optimization

3.4.1 Output amounts overwriting can be skipped

Severity: Gas Optimization

Context: V2DutchOrderReactor.sol#L94-L96

Description: In Dutch auction orders, order creators delegate the setting of auction parameters to cosigners. Besides other parameters, cosigners are allowed to overwrite an order's input and output amounts to improve the execution price of an order. But this is optional: a cosigner can decide to not overwrite order's amounts by setting zero amounts (V2DutchOrderReactor.sol#L87, V2DutchOrderReactor.sol#L100). However, even when a cosigner doesn't overwrite output amounts, it still has to provide a list of zeroes which is always iterated:

```
if (order.cosignerData.outputAmounts.length != order.baseOutputs.length) {
    revert InvalidCosignerOutput();
}
for (uint256 i = 0; i < order.baseOutputs.length; i++) {
    DutchOutput memory output = order.baseOutputs[i];
    uint256 outputAmount = order.cosignerData.outputAmounts[i];
    if (outputAmount != 0) {
        if (outputAmount < output.startAmount) {
            revert InvalidCosignerOutput();
        }
        output.startAmount = outputAmount;
    }
}
```

This consumes additional gas when there is no intention to iterate and overwrite output amounts.

Recommendation: Consider allowing cosigners to specify an empty list of output amounts when they're not intending overwriting them.

Uniswap: Acknowledged, not doing for now due to code /integration readability and gas improvement was marginal.

Cantina Managed: Acknowledged.

3.4.2 Duplicate check on cosigner times

Severity: Gas Optimization

Context: V2DutchOrderReactor.sol#L119-L121, DutchDecayLib.sol#L31-L32

Description: The cosigner data validation on the start and end auction times is redundant in `V2DutchOrderReactor`, as it will be performed in `DutchDecayLib`.

Recommendation: Remove the validation check in `V2DutchOrderReactor`.

Uniswap: Fixed in PR 244.

Cantina Managed: Fixed.

3.4.3 Decaying can often be skipped to reduce gas consumption

Severity: Gas Optimization

Context: DutchDecayLib.sol#L31-L47

Description: In Dutch auction orders, input and output amounts can decay over time, which is implemented in the DutchDecayLib.decay() function. As per the order validation logic, in an order, only either the input or the outputs can decay (V2DutchOrderReactor.sol#L131-L138), thus either of the two will always hold true for Dutch auction orders:

1. Input's start and end amounts are equal.
2. Outputs' start and end amounts are equal.

However, the `DutchDecayLib.decay()` function doesn't return earlier when `startAmount` and `endAmount` are equal and always computes a decayed amount (DutchDecayLib.sol#L38-L46). This incurs an increased gas consumption on Dutch auction orders, especially orders with multiple non-decaying outputs.

Recommendation: In the `DutchDecayLib.decay()` function, consider returning earlier when `startAmount` equals `endAmount` to reduce gas consumption by the function.

Uniswap: Fixed in PR 241.

Cantina Managed: Fixed.

3.5 Informational

3.5.1 Inconsistent functions naming

Severity: Informational

Context: BaseReactor.sol#L143, BaseReactor.sol#L148

Description: In the BaseReactor contract, functions _prepare() and _fill() are internal and begin with an underscore, following the common functions naming convention. However, functions resolve() and transferInputTokens() don't begin with an underscore while being internal functions.

Recommendation: Consider following the naming convention in all cases.

Uniswap: Fixed in PR 242.

Cantina Managed: Fixed.

3.5.2 Remove unused function

Severity: Informational

Context: BaseReactor.sol#L25, ExclusivityLib.sol#L57

Description: The code contains unused function and error. Removing these is a best practice and will also save gas during deployment.

Recommendation: It's recommended to remove the unused function and error.

Uniswap: Fixed in PR 240.

Cantina Managed: Fixed.

3.5.3 Incorrect comments

Severity: Informational

Context: V2DutchOrderReactor.sol#L17, V2DutchOrderReactor.sol#L110-L111

Description: The following code contains corrections for some incorrect comments that need to be fixed.

- V2DutchOrderReactor.sol#L17:

```
  - /// - If inputAmount is nonzero, then ensure it is less than specified baseOutput and replace
↪  startAmount
  + /// - If inputAmount is nonzero, then ensure it is less than specified baseInput and replace
↪  startAmount
```

- V2DutchOrderReactor.sol#L110-L111:

```
  - /// - deadline must be greater than or equal than decayEndTime
  + /// - deadline must be greater than or equal to decayEndTime
  - /// - decayEndTime must be greater than or equal to decayStartTime
  + /// - decayEndTime must be greater than decayStartTime
```

Recommendation: It's recommended to fix the comments.

Uniswap: Fixed in PR 239.

Cantina Managed: Fixed.