Uniswap - TWAP Auction

Security Review

Cantina Managed review by:
Kaden, Lead Security Researcher
Devtooligan, Security Researcher

November 16, 2025

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

From Oct 19th to Oct 25th the Cantina team conducted a review of continuous-clearing-auction on commit hash 93c0c780. The team identified a total of **11** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	1	1	0
Low Risk	2	2	0
Gas Optimizations	2	2	0
Informational	5	4	1
Total	**11**	**10**	**1**

2.1 Scope

The security review had the following components in scope for continuous-clearing-auction on commit hash 93c0c780:

```
src
├── Auction.sol
├── AuctionFactory.sol
├── AuctionStepStorage.sol
├── BidStorage.sol
├── CheckpointStorage.sol
├── libraries
│   ├── AuctionStepLib.sol
│   ├── BidLib.sol
│   ├── CheckpointLib.sol
│   ├── ConstantsLib.sol
│   ├── CurrencyLibrary.sol
│   ├── FixedPoint128.sol
│   ├── FixedPoint96.sol
│   ├── ValidationHookLib.sol
│   └── ValueX7Lib.sol
├── PermitSingleForwarder.sol
├── TickStorage.sol
└── TokenCurrencyStorage.sol
```

3 Findings

3.1 High Risk

3.1.1 Unsold tokens permanently locked after successful auction

Severity: High Risk

Context: Auction.sol#L621

Summary: The `sweepUnsoldTokens()` function contains flawed logic that prevents the recovery of unsold tokens when an auction graduates (meets its reserve). The function only allows sweeping either 0 tokens (graduated auction) or the entire token supply (non-graduated auction), resulting in unsold tokens being permanently locked in the contract after a successful auction.

Description: The issue lies in this line of the `sweepUnsoldTokens()` function:

```
_sweepUnsoldTokens(_isGraduated() ? 0 : TOTAL_SUPPLY);
```

This logic creates two problematic scenarios:

1. Graduated auction (`_isGraduated() == true`): The function attempts to sweep 0 tokens, leaving any unsold tokens permanently locked in the contract.
2. Non-graduated auction (`_isGraduated() == false`): The function attempts to sweep the entire `TOTAL_SUPPLY`, which will fail if any tokens have been sold (as they would have been transferred out via `exitBid()`).

The function is only useful in the edge case where an auction completes without any bids and fails to meet its reserve. In all other scenarios, tokens either cannot be swept or the sweep will revert.

Additionally, if the reserve requirement is set to 0, the auction is automatically considered graduated, preventing any token recovery regardless of sales.

Impact Explanation: The impact is high as unsold tokens in successful auctions become permanently locked in the contract with no recovery mechanism. This represents a direct loss of assets for the token recipient.

Likelihood Explanation: The likelihood is high for auctions that graduate without selling their entire token supply, which is a common scenario. Any auction that meets its reserve requirement but doesn't sell all tokens will experience this issue.

Recommendation: Modify the `sweepUnsoldTokens()` function to calculate and sweep the actual amount of unsold tokens:

```
  function sweepUnsoldTokens() external onlyAfterAuctionIsOver ensureCheckpointed {
      if (sweepUnsoldTokensBlock != 0) revert CannotSweepTokens();
-     _sweepUnsoldTokens(_isGraduated() ? 0 : TOTAL_SUPPLY);
+     uint256 unsoldTokens = TOKEN.balanceOf(address(this));
+     _sweepUnsoldTokens(unsoldTokens);
  }
```

Uniswap Labs: Fixed in PR 231.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Address prediction mismatch in `getAuctionAddress()` function

Severity: Medium Risk

Context: AuctionFactory.sol#L44-L46

Summary: The `getAuctionAddress()` function in the AuctionFactory contract incorrectly calculates the deployment address when `ActionConstants.MSG_SENDER` is used for recipient parameters. This mismatch can lead to loss of tokens that are transferred to the incorrectly predicted address, which is particularly severe since the typical deployment pattern involves sending the entire token supply to the auction address before deployment.

Description: The `getAuctionAddress()` function is designed to predict the deterministic address where an auction contract will be deployed. This is critical for deployment scripts that need to pre-fund the auction address with tokens before the actual deployment occurs.

In the `initializeDistribution()` function, when `ActionConstants.MSG_SENDER` is encountered, it's correctly replaced with `msg.sender` (the actual deployer):

```
if (parameters.tokensRecipient == ActionConstants.MSG_SENDER) parameters.tokensRecipient
↪  = msg.sender;
if (parameters.fundsRecipient == ActionConstants.MSG_SENDER) parameters.fundsRecipient =
↪  msg.sender;
```

However, in the `getAuctionAddress()` view function, the same logic uses `msg.sender` instead of the `sender` parameter:

```
// Current implementation (incorrect)
if (parameters.tokensRecipient == ActionConstants.MSG_SENDER) parameters.tokensRecipient
↪  = msg.sender;
if (parameters.fundsRecipient == ActionConstants.MSG_SENDER) parameters.fundsRecipient =
↪  msg.sender;
```

Since `getAuctionAddress()` is a view function, `msg.sender` refers to whoever is calling the view function to predict the address, not the intended deployer. This causes the address computation to use different parameters than what would be used during actual deployment.

The typical deployment flow would be:

1. Call `getAuctionAddress()` to predict the auction contract address.
2. Transfer the entire token supply to the predicted address.
3. Call `initializeDistribution()` to deploy the auction.

If the predicted address is incorrect, the tokens sent in step 2 would be permanently lost, as the actual deployment would occur at a different address.

Impact Explanation: The impact is high because the standard deployment pattern for auction contracts involves pre-funding the predicted address with the entire token supply before deployment. If the address prediction is incorrect, these tokens would be sent to an address where no contract will ever be deployed, resulting in permanent loss of all tokens intended for the auction.

Likelihood Explanation: The likelihood is medium because exploitation requires specific conditions: the deployer must use `ActionConstants.MSG_SENDER` for recipient parameters, call `getAuctionAddress()` from a different address than the deployment address (such as through a deployment script or multi-step process), and transfer tokens before realizing the address mismatch. While these conditions are plausible in typical deployment workflows, they require a specific sequence of actions without verification between steps.

Recommendation: Update the `getAuctionAddress()` function to use the `sender` parameter instead of `msg.sender` when replacing `ActionConstants.MSG_SENDER` placeholders:

```
  function getAuctionAddress(address token, uint256 amount, bytes calldata configData,
  ↪  bytes32 salt, address sender)
      public
      view
      returns (address)
  {
      if (amount > type(uint128).max) revert InvalidAmount(amount);
      AuctionParameters memory parameters = abi.decode(configData, (AuctionParameters));
-     if (parameters.tokensRecipient == ActionConstants.MSG_SENDER)
↪   parameters.tokensRecipient = msg.sender;
+     if (parameters.tokensRecipient == ActionConstants.MSG_SENDER)
↪   parameters.tokensRecipient = sender;
-     if (parameters.fundsRecipient == ActionConstants.MSG_SENDER)
↪   parameters.fundsRecipient = msg.sender;
+     if (parameters.fundsRecipient == ActionConstants.MSG_SENDER)
↪   parameters.fundsRecipient = sender;
```

```
        bytes32 initCodeHash =
            keccak256(abi.encodePacked(type(Auction).creationCode, abi.encode(token,
            ↪  uint128(amount), parameters)));
        salt = keccak256(abi.encode(sender, salt));
        return Create2.computeAddress(salt, initCodeHash, address(this));
    }
```

Uniswap Labs: Fixed in PR 219.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Incorrect comparison operator

Severity: Low Risk

Context: AuctionStepStorage.sol#L69-L70

Description: In `AuctionStepStorage._advanceStep`, we revert if `$_offset > _LENGTH`:

```
if ($_offset > _LENGTH) revert AuctionIsOver();
```

In the case that `$_offset == _LENGTH`, we will not revert. Instead we will read empty step data and write zero values to `$step` in storage. In practice this should not be possible as we only `_advanceStep` if the block to checkpoint is strictly greater than the end block of the step and we never checkpoint with a block greater than the end block. However, this is recommended to be fixed regardless for added protection.

Recommendation: Revert in `_advanceStep` if the `$_offset` is greater than or equal to the `_LENGTH`:

```
- if ($_offset > _LENGTH) revert AuctionIsOver();
+ if ($_offset >= _LENGTH) revert AuctionIsOver();
```

Uniswap Labs: Fixed in PR 240.

Cantina Managed: Fix verified.

3.3.2 Bids can be claimed multiple times

Severity: Low Risk

Context: Auction.sol#L597-L607

Description: In the token claim logic used by both `Auction.claimTokens` and `Auction.claimTokensBatch`, we never validate that the bid has not already been claimed.

```
function claimTokens(uint256 _bidId) external onlyAfterClaimBlock ensureCheckpointed {
    if (!_isGraduated()) revert NotGraduated();

    (address owner, uint256 tokensFilled) = _internalClaimTokens(_bidId);
    Currency.wrap(address(TOKEN)).transfer(owner, tokensFilled);

    emit TokensClaimed(_bidId, owner, tokensFilled);
}

// ...

function _internalClaimTokens(uint256 bidId) internal returns (address owner, uint256
↪  tokensFilled) {
    Bid storage $bid = _getBid(bidId);
    if ($bid.exitedBlock == 0) revert BidNotExited();

    // Set return values
    owner = $bid.owner;
    tokensFilled = $bid.tokensFilled;

    // Set the tokens filled to 0
```

```
    $bid.tokensFilled = 0;
}
```

As a result, it's possible to call `claimTokens` or `claimTokensBatch` multiple times for the same bid. Since `tokensFilled` is set to 0, this does not pose a significant threat, although it does allow the ability to cause multiple unexpected event emissions for the same claim.

Recommendation: Revert in `_internalClaimTokens` if `$bid.tokensFilled == 0`:

```
  function _internalClaimTokens(uint256 bidId) internal returns (address owner, uint256
  ↪  tokensFilled) {
      Bid storage $bid = _getBid(bidId);
      if ($bid.exitedBlock == 0) revert BidNotExited();
+     if ($bid.tokensFilled == 0) revert BidAlreadyClaimed();


      // Set return values
      owner = $bid.owner;
      tokensFilled = $bid.tokensFilled;


      // Set the tokens filled to 0
      $bid.tokensFilled = 0;
  }
```

Uniswap Labs: Fixed in PR 240.

Cantina Managed: Fix verified.

3.4 Gas Optimization

3.4.1 Redundant conditional case

Severity: Gas Optimization

Context: Auction.sol#L478-L481

Description: In `Auction.exitPartiallyFilledBid`, we conditionally retrieve the `lastFullyFilledCheckpoint` as the `currentBlockCheckpoint` in case the `lastFullyFilledCheckpointBlock` is the current `block.number`:

```
// If the provided hint is the current block, use the checkpoint returned by
↪  `checkpoint()` instead of getting it from storage
Checkpoint memory lastFullyFilledCheckpoint = lastFullyFilledCheckpointBlock ==
↪  block.number
    ? currentBlockCheckpoint
    : _getCheckpoint(lastFullyFilledCheckpointBlock);
```

However, we later revert if the next checkpoint `clearingPrice` is less than the `bidMaxPrice`:

```
// Since `lower` points to the last fully filled Checkpoint, it must be < bid.maxPrice
// The next Checkpoint after `lower` must be partially or fully filled (clearingPrice >=
↪  bid.maxPrice)
// `lower` also cannot be before the bid's startCheckpoint
if (
    lastFullyFilledCheckpoint.clearingPrice >= bidMaxPrice
        // @audit we revert if next checkpoint clearingPrice < bidMaxPrice
        || _getCheckpoint(lastFullyFilledCheckpoint.next).clearingPrice < bidMaxPrice
        || lastFullyFilledCheckpointBlock < bidStartBlock
) {
    revert InvalidLastFullyFilledCheckpointHint();
}
```

Since the checkpoint of the current block will not yet have a `next` checkpoint, it's not possible for the `lastFullyFilledCheckpoint` to be the checkpoint of the current block as it will always revert due to the unset `clearingPrice`.

Recommendation: Remove the conditional case and simplify the assignment of `lastFullyFilledCheckpoint`:

```
Checkpoint memory lastFullyFilledCheckpoint =
↪  _getCheckpoint(lastFullyFilledCheckpointBlock);
```

Uniswap Labs: Fixed in PR 237.

Cantina Managed: Fix verified.

3.4.2 Redundant `mulDiv`

Severity: Gas Optimization

Context: Auction.sol#L269, Auction.sol#L287

Description: In `Auction._iterateOverTicksAndFindClearingPrice`, we compute the `clearingPrice`, in two separate places, as follows:

```
uint256 clearingPrice = sumCurrencyDemandAboveClearingQ96_.fullMulDivUp(1, TOTAL_SUPPLY);

// ...

clearingPrice = sumCurrencyDemandAboveClearingQ96_.fullMulDivUp(1, TOTAL_SUPPLY);
```

Since we are multiplying the `sumCurrencyDemandAboveClearingQ96_` by 1 before dividing by `TOTAL_SUPPLY`, this is identical to simply dividing `sumCurrencyDemandAboveClearingQ96_` by `TOTAL_SUPPLY` (rounding up).

Recommendation: Replace the use of `fullMulDivUp` in each of these cases with an upward rounding division:

```
-  uint256 clearingPrice = sumCurrencyDemandAboveClearingQ96_.fullMulDivUp(1,
↪   TOTAL_SUPPLY);
+  uint256 clearingPrice = sumCurrencyDemandAboveClearingQ96_.ceilDiv(TOTAL_SUPPLY);

   // ...

-  clearingPrice = sumCurrencyDemandAboveClearingQ96_.fullMulDivUp(1, TOTAL_SUPPLY);
+  clearingPrice = sumCurrencyDemandAboveClearingQ96_.ceilDiv(TOTAL_SUPPLY);
```

Uniswap Labs: Fixed in PR 245.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 Incorrect variable name in comment

Severity: Informational

Context: Auction.sol#L485-L487

Description: In `Auction.exitPartiallyFilledBid`, we include the following logic with a comment referring to the `lower` variable:

```
// Since `lower` points to the last fully filled Checkpoint, it must be < bid.maxPrice
// The next Checkpoint after `lower` must be partially or fully filled (clearingPrice >=
↪  bid.maxPrice)
// `lower` also cannot be before the bid's startCheckpoint
if (
    lastFullyFilledCheckpoint.clearingPrice >= bidMaxPrice
        || _getCheckpoint(lastFullyFilledCheckpoint.next).clearingPrice < bidMaxPrice
        || lastFullyFilledCheckpointBlock < bidStartBlock
) {
    revert InvalidLastFullyFilledCheckpointHint();
}
```

It appears to be the case that the `lower` variable name was updated to `lastFullyFilledCheckpoint` but the comment has not also been updated accordingly as the `lower` variable does not exist in this context.

Recommendation: Update the comment to reference the updated variable name:

```
- // Since `lower` points to the last fully filled Checkpoint, it must be < bid.maxPrice
+ // Since `lastFullyFilledCheckpoint` points to the last fully filled Checkpoint, it
 ↪  must be < bid.maxPrice
- // The next Checkpoint after `lower` must be partially or fully filled (clearingPrice
 ↪  >= bid.maxPrice)
+ // The next Checkpoint after `lastFullyFilledCheckpoint` must be partially or fully
 ↪  filled (clearingPrice >= bid.maxPrice)
- // `lower` also cannot be before the bid's startCheckpoint
+ // `lastFullyFilledCheckpoint` also cannot be before the bid's startCheckpoint
```

Uniswap Labs: Fixed in PR 237.

Cantina Managed: Fix verified.

3.5.2 Checkpoint function can be denied service through tick iteration exploitation

Severity: Informational

Context: Auction.sol#L270-L288

Summary: The auction's checkpoint mechanism can be forced to iterate over many price ticks, leading to excessive gas consumption. An attacker can place numerous small bids across different price levels to maximize the gas cost of subsequent checkpoint operations.

Description: The vulnerability exists in the tick iteration logic that determines the clearing price during checkpoint operations. When the auction needs to find the new clearing price, it iterates through all initialized ticks between the current clearing price and the target price.

The problematic loop continues while there is sufficient demand to warrant a higher clearing price. An attacker can exploit this by placing many small bids at different price levels below a larger bid. This forces the checkpoint function to iterate through all initialized ticks when the clearing price moves up, consuming excessive gas. With smaller tick spacing values, hundreds of ticks can exist within a price range, potentially making checkpoint operations expensive or causing out-of-gas reverts.

However, the feasibility of this attack may be currently limited by the gas cost of bid placement. According to the project team, each bid costs approximately 150,000 gas, meaning 200 bids would approach or exceed typical block gas limits, making large-scale exploitation impractical under current conditions.

Recommendation: To prevent potential issues with tick iteration:

1. Document recommended tick spacing values clearly, emphasizing that tick spacing should be at least 1% of the floor price or expected clearing price. Small values should be explicitly discouraged.
2. Consider enforcing minimum tick spacing at auction creation to prevent configuration errors:

```
// Require tick spacing to be at least 1% of floor price
if (tickSpacing < floorPrice / 100) revert TickSpacingTooSmall();
```

3. Provide clear configuration guidance to auction creators about the relationship between tick spacing, gas costs, and potential iteration issues to ensure informed decision-making.

Uniswap Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.5.3 Unused custom error

Severity: Informational

Context: ITickStorage.sol#L10

Description/Recommendation: This error is currently unused. If this error is not intended to be used it may be safely removed.

Uniswap Labs: Fixed in PR 219.

Cantina Managed: Fix verified.

3.5.4 Incorrect comment

Severity: Informational

Context: Auction.sol#L191-L193

Description: In `Auction._sellTokensAtClearingPrice`, we include the following comment:

```
// Conversely, if `expectedCurrencyRaisedAtClearingPriceTickQ96_X7` is less than
↪  `demandAtClearingPriceQ96_X7`,
// then the amount of currency we expect to raise from bids at clearing price is greater
// than the actual demand at clearing price which is incorrect for a partial fill.
```

This comment is incorrect and instead should say, "if `expectedCurrencyRaisedAtClearingPriceTickQ96_X7` is *greater* than `demandAtClearingPriceQ96_X7`..." as we are referring to the `expectedCurrencyRaisedAtClearingPriceTickQ96_X7 > demandAtClearingPriceQ96_X7` case.

Recommendation: Update the comment to use the correct comparison:

```
-  // Conversely, if `expectedCurrencyRaisedAtClearingPriceTickQ96_X7` is less than
↪  `demandAtClearingPriceQ96_X7`,
+  // Conversely, if `expectedCurrencyRaisedAtClearingPriceTickQ96_X7` is greater than
↪  `demandAtClearingPriceQ96_X7`,
  // then the amount of currency we expect to raise from bids at clearing price is
  ↪  greater
  // than the actual demand at clearing price which is incorrect for a partial fill.
```

Uniswap Labs: Fixed in PR 254.

Cantina Managed: Fix verified.

3.5.5 Missing validation for floor price against maximum bid price limit

Severity: Informational

Context: Auction.sol#L85

Summary: The contract calculates a `MAX_BID_PRICE` constant as `type(uint256).max / TOTAL_SUPPLY` but does not enforce that the floor price parameter stays below this maximum threshold. This missing validation could allow configuration of floor prices that exceed the system's maximum bid price, potentially leading to inconsistent auction behavior or preventing valid bids from being placed.

Description: The contract defines a maximum bid price limit through the calculation `MAX_BID_PRICE = type(uint256).max / TOTAL_SUPPLY`, which establishes an upper bound for bid prices to prevent overflow issues when calculating total values. However, when setting auction parameters, there is no validation to ensure that `_parameters.floorPrice` remains below this `MAX_BID_PRICE` threshold.

The contract already includes an unused error definition `FloorPriceAboveMaxBidPrice` that suggests this validation was intended but not implemented. Without this check, administrators could inadvertently configure auctions with floor prices that exceed the maximum allowable bid price, creating a logical inconsistency where the minimum acceptable price is higher than the maximum processable price.

Impact: The absence of this validation could result in auctions being configured with floor prices that exceed the system's maximum bid price limit. This configuration error would effectively make the auction impossible to fulfill, as no bid could simultaneously satisfy the floor price requirement while staying within the maximum bid price constraint. Users attempting to participate in such auctions would be unable to place valid bids, leading to failed auctions and potential confusion.

Likelihood: The likelihood of this issue occurring is low to medium. It requires an administrator to set a floor price value that exceeds `type(uint256).max / TOTAL_SUPPLY` during auction parameter configuration.

While this would likely be caught during testing or initial deployment, the lack of programmatic enforcement means the issue could arise during parameter updates or in edge cases where extremely high floor prices are considered.

Recommendation: Consider implementing the validation check to ensure floor prices remain within acceptable bounds. One approach could be to add this validation when auction parameters are set:

```
// Add this validation when setting auction parameters
if (_parameters.floorPrice >= MAX_BID_PRICE) {
    revert FloorPriceAboveMaxBidPrice();
}
```

Uniswap Labs: Fixed in PR 263.

Cantina Managed: Fix verified.