OpenZeppelin | security

# Uniswap Labs TWAP Auction Audit


September 28, 2025

# Table of Contents

# Summary

| | |
|---|---|
| Type | DeFi |
| Timeline | From 2025-08-25<br>To 2025-09-23 |
| Languages | Solidity |

| | |
|---|---|
| Total Issues | 50 (0 resolved) |
| Critical Severity Issues | 3 (0 resolved) |
| High Severity Issues | 3 (0 resolved) |
| Medium Severity Issues | 6 (0 resolved) |
| Low Severity Issues | 22 (0 resolved) |
| Notes & Additional Information | 16 (0 resolved) |

# Scope

OpenZeppelin initially audited the Uniswap/twap-auction repository at commit a450d73 and tag `v0.0.2-audit-oz`. Later, the audit commit was changed to 56519be to include pull request 86 and then once again to 4a25344 in order to include pull requests 92 and 94.

In scope were the following files:

```
src
├── Auction.sol
├── AuctionFactory.sol
├── AuctionStepStorage.sol
├── BidStorage.sol
├── CheckpointStorage.sol
├── PermitSingleForwarder.sol
├── TickStorage.sol
├── TokenCurrencyStorage.sol
├── interfaces
│   ├── IAuction.sol
│   ├── IAuctionFactory.sol
│   ├── IAuctionStepStorage.sol
│   ├── ICheckpointStorage.sol
│   ├── IPermitSingleForwarder.sol
│   ├── ITickStorage.sol
│   ├── ITokenCurrencyStorage.sol
│   ├── IValidationHook.sol
│   └── external
│       ├── IDistributionContract.sol
│       ├── IDistributionStrategy.sol
│       └── IERC20Minimal.sol
└── libraries
    ├── AuctionStepLib.sol
    ├── BidLib.sol
    ├── CheckpointLib.sol
    ├── CurrencyLibrary.sol
    ├── DemandLib.sol
    └── FixedPoint96.sol
```

# System Overview

The TWAP Auction protocol is a time-phased, uniform clearing-price token sale mechanism that releases supply over a programmable schedule while demand accumulates across discrete price ticks to discover a single fair market price over time. Participants place max-price bids (exact-in currency or exact-out tokens) and never pay above their stated limit. As bids arrive, the sale advances issuance and updates a monotonic clearing price that is snapped to a configured tick spacing and bounded by a floor, aligning allocations with sustained demand instead of momentary spikes and reducing sniping and launch-time volatility.

Each auction is deployed by a factory with clear, immutable parameters, recipients for collecting currency and unsold tokens, floor price, tick spacing, and a graduation threshold indicating success—alongside a compact step schedule that encodes how much supply is released per block. Mathematically, prices are represented in X96 fixed-point, demand is aggregated per tick, and the system uses checkpointed cumulative variables to compute fills and refunds without per-block iteration: bids strictly above the clearing price are filled completely, and bids at the clearing boundary receive proportional fills.

After the end block, outcomes are determined by graduation: successful sales enable withdrawing currency and claiming tokens after the configured claim block, while unsuccessful sales refund currency and return unsold tokens. The protocol supports using ETH and ERC-20 (with Permit2 for transfers) as currency, and any other ERC-20 token as the token being sold. The protocol also implements option-validation hooks for eligibility or rate-limits, along with deterministic deployments suitable for repeatable, auditable launches.

# Security Model and Trust Assumptions

The protocol does not implement a role-based access-control system that restricts the operations. However, there is an implicit responsibility on the owner or deployer of the `Auction` contract to provide the appropriate parameters for the correct functioning of the

auction. Bidders must rely upon and trust the fact that the deployer of the `Auction` contract will parse the correct parameters for the successful operation of the auction. Failing to do so might cause their funds to be stolen, stuck, or delayed.

In particular, these parameters define the:

- token and currency assets used
- the `totalSupply` of tokens for the auction
- the supply curve and, in particular, the step configuration
- the time when the auction starts, ends, and when the tokens can be claimed
- the token and currency recipients at the end of the auction
- the data to be called at the currency recipient during the currency transfer
- the validation contract that will validate bidders from joining the auction
- the initial price and the separation between future prices
- the threshold at which an auction can be considered graduated or not

Also, since the deployer is the one defining the recipients of the deposited assets after the auction ends, they have the opportunity to inflate the auction by bidding through sibling accounts that might increase the overall price of the token, or leave bidders out of the auction if their max-price is overpassed. In case the auction uses a validation contract, it must also be trusted to function properly, as otherwise, it can censor bidders from joining the auction.

# Integration

Protocols using or integrating with the TWAP Auction protocol need to consider the following characteristics of the implementation:

- To validate that the deposited token transfer is acceptable compared to the passed `totalSupply` value during deployment, it must be called the `onTokensReceived` function along with the call to deploy a new `Auction` contract instance through the factory, as there is no enforcement in the codebase.
- If the auction does not graduate, or less than `totalSupply` tokens are sold, the `tokensRecipient` address must implement a way to handle the tokens
- The `validationHook` contract is currently able to provide viewable functionality to whitelist bidders. However, passing the wrong contract might prevent all bids from happening.
- The `fundsRecipient` address must be able to receive the currency for the auction. Plus, it must implement the specified `fundsRecipientData` data that is passed

during the deployment. Otherwise, the currency will get stuck in the contract, as this data cannot be modified.
- Tokens that are above the `totalsupply` value and get parsed will not be able to be computed or rescued. As such, deposits should match the exact value of `totalSupply` to prevent losing these tokens.
- Values from the prices are in X96.
- Using tokens or currencies that present odd implementations or behaviors might be able to revert calls, transfers, or cause unpredictable outcomes.
- Assets that differ in their decimals by a significant amount or have a high difference in their intrinsic value might cause considerable rounding effects during the bidding process.

# Design Choices

The following design choices were identified in the TWAP Auction protocol:

- Users who get outbid cannot readjust their position internally. Instead, they need to partially exit their bid, refunding part of the currency, and then create a new bid with a higher max price.
- The protocol is meant to be used with the Token Launcher. However, it can still be used as standalone through the factory. However, token allocation is not an atomic process, and validations are not enforced by the protocol, meaning that the deposited balance might differ from the actual `totalSupply` value.
- Calculations do not make use of the balances in the contract. Instead, an internal account system is used to keep track of whatever enters the contract through the deployment and bids, and what is taken out from it in the form of claims, refunds, and withdrawals. However, there is an assumption that the real assets associated with a position from a bidder will not be able to interfere with those from another bidder.
- There is no limit to the window in which the auction can live. Using a large value for the end of the auction might trap bidders inside of it without the possibility of getting refunded or collecting the tokens. Moreover, the claiming window can push this process even further.

# Critical Severity

## C-01 Bidder Can Skip to the End of the Auction Due to Incorrectly Checkpointing the End Block

Users can partially exit a bid at any time during the auction. However, `partiallyExitBid` unconditionally calls `_unsafeCheckpoint(endBlock)`, which immediately creates a checkpoint at the auction's final block, effectively rolling the auction forward prematurely. This can prematurely finalize the state during mid-auction exits, distorting clearing behavior and creating inconsistent outcomes across bidders depending on the exit timing.

Consider checkpointing `endBlock` only after the auction has ended (e.g., when `block.number >= endBlock` or behind `onlyAfterAuctionIsOver`) to prevent premature checkpointing.

## C-02 Bidders Can Over-Exit By Passing Partially Filled Checkpoints As Fully Filled

When users partially exit, which might happen when they are outbid, they must provide their last fully filled checkpoint as a hint, which is validated in the code. The current validation only ensures that `lastFullyFilledCheckpoint.next` has either been outbid or partially filled: it does not require that `lastFullyFilledCheckpoint.clearingPrice` is strictly less than `bid.maxPrice`. This lets a bidder present a checkpoint that is only partially filled as "fully filled" so long as its `next` points to `outbidCheckpoint`, enabling extraction of more value than intended and creating a race where bidders who exit later may not have enough balance left to settle.

For example, consider a bid with `bid.maxPrice = 1e18`. The first checkpoint clears at `0.9e18`, checkpoints two, three, and four clear at `1.0e18`, and checkpoint five clears at `1.1e18`. Only the first checkpoint should be considered fully filled. However, the bidder can submit checkpoint four as their `lastFullyFilledCheckpoint`. The validation check `(1.1e18 < 1e18 || 4 < 1)` evaluates to `false`, so no revert occurs, and the contract incorrectly accepts checkpoint four as fully filled, allowing the bidder to exit with excess value.

Consider enforcing `lastFullyFilledCheckpoint.clearingPrice < bid.maxPrice` when accepting the hint to prevent partially filled checkpoints from being misrepresented as fully filled and to remove the settlement race.

## C-03 Bidder Can Bid for a Fraction

When a bidder places a bid, the required amount to deposit depends on if they are submitting it as denoted exactly in currency or token. When the bid is placed with `exactIn` set as `false`, the protocol uses the `inputAmount` function from the `BidLib` library to convert the token value into the maximum they are willing to pay for it.

However, as the result from the `fullMulDivUp` function is returned as a `uint256` value, the cast to `uint128` would silently truncate its value. For this to happen, the `amount * maxPrice / Q96` should be bigger than `uint128.max`, but as both the `amount` and `maxPrice` are controlled by the bidder, the malicious actor could pass a `maxPrice` big enough so that the `requiredCurrencyAmount` value is small due to the truncation. Then, the transferred currency would be small compared to the bid, but the flow would continue submitting the bid to the internal account system, which would probably fulfill the order due to the high `maxPrice` value, and it would hurt the owner due to the smaller deposit for those tokens.

Consider using a safe casting mechanism to prevent truncating the value.

# High Severity

## H-01 `onTokensReceived` Hook Execution Is Not Mandatory

The `Auction` contract is deployed using the `AuctionFactory` contract, which takes the `AuctionParameters` inputs and parses them to the `constructor` function. Among these parameters, `totalSupply` is stored without any checks. In the Token Launcher, the deployment, transfer of assets, and validation by calling the `onTokensReceived` function prevents sending less than expected tokens to the `Auction` contract. However, if the protocol is used in a standalone manner without the Token Launcher, the auction will not validate if the passed parameter matches the expected initial supply of tokens.

This results in all computed calculations being done expecting those assets, but in reality, bidders might not be able to receive their tokens. Plus, as there is no path to compensate these bidders, they will not be able to claim back the currency spent in those tokens. This is particularly important because a malicious attacker could create an auction through the `AuctionFactory` contract with all the right parameters, but missing the transfer of the total or partial tokens, in which bidders would start bidding and their funds would get stuck. Moreover, the attacker can deposit a fraction of the tokens that would resemble to the `totalSupply` value but orders of magnitude less so bidders might think that it has the expected tokens even though it does not.

Consider enforcing the call to the `onTokensReceived` function during the deployment of the `Auction` contract to assert that the auction will have the sufficient assets.

## H-02 Stale Graduation Check Can Allow Both Tokens and Currency to Be Swept

In the scenario where the latest checkpoint's cleared amount is below the graduation threshold but after snapshotting at `endBlock` rises above the threshold, the following issue arises:

When calling the `sweepUnsoldTokens` function, the `isGraduated` flag is evaluated based on the latest checkpoint. However, the `_getFinalCheckpoint` function is never invoked before this check, meaning the contract relies on stale data. If it has not been updated at the end of the auction, the auction creator can (through the assets' recipients) sweep both the partial currency and most of the tokens, leaving bidders without refunds and without tokens.

The following would be the flow:

1. Latest checkpoint exists before the end of the auction, while the current block is already past the end of the auction.
2. Auction owner calls `sweepUnsoldTokens`. In case the auction's graduation is not yet reflected in the latest checkpoint, the auction is treated as not graduated, and the owner sweeps the entire token supply.
3. The owner then transfers a small amount of tokens to the `Auction` contract to fill a bid, which internally triggers `_unsafeCheckpoint` through `exitBid` or `partiallyExitBid`.
4. `_unsafeCheckpoint` creates a new checkpoint, and now if such a bid is sufficient, the auction might be considered graduated.

5. The auction owner calls the `sweepCurrency` function, which does not share the same timestamp variable with the previous method, successfully withdrawing all the currency associated to the filled bids.
6. Bidders might no longer claim their bids, as there are not enough tokens to be claimed, while the owner keeps the tokens and the filled equivalent of currency.

Consider calling `_getFinalCheckpoint` before `isGraduated` to not operate on stale checkpoints.

## H-03 Setting `fundsRecipient` as Token Address Could Steal Currency and Tokens

The `Auction` contract sets the `fundsRecipient` address to the one that will receive the currency raised during the auction. Along with it, it can accept data to perform more complex actions, such as acknowledging the reception or handle the received assets. However, besides restricting the `fundsRecipient` address to not be the one of the `Auction` contract or the zero address, there is nothing else validating what is behind.

In particular, if the `fundsRecipient` address is set as the one for the auction's token address, it is possible to steal both currency and tokens during the currency's sweep. This is because that function first transfers the raised currency and then performs the arbitrary call, which can be the one of a `transfer` call with the owner's address and the `totalSupply` value as encoded parameters. This would either:

- get the tokens back to the owner while locking the currency raised in the token contract, if it cannot handle them (e.g., move them to another account)
- steal both assets if the token contract was previously crafted to forward the currency raised to the owner

As there are no limitations to what the arbitrary call can do on behalf of the `Auction` contract, the malicious owner could take advantage of it to compromise the outcome of the auction. Consider restricting `fundsRecipient` to addresses that do not involve the assets at play, permit2 functionalities, or any other module that might depend on a specific outcome from the `Auction` contract (in particular the Token Launcher). Moreover, consider hardcoding the function selector and its parameters to prevent crafting arbitrary calls.

# Medium Severity

## M-01 `Auction` Deployer Can Influence Auction's Final Conditions

The actor deploying the `Auction` contract has to set the `tokensRecipient` and `fundsRecipient` addresses that will collect unsold tokens and currency assets, respectively. Even though the flow from the Token Launcher would not accept direct bids from it, the account setting these values can still influence the outcome of the auction. In particular, the owner can inject more demand through bids that would eventually increase the `clearingPrice` of the auction.

As these addresses might be controlled by the same actor, both sides of the assets filled will return to the specified addresses, resulting in not losing any funds in the process but increasing the actual value for the ones bidding. Depending on the state of the auction, this might also cause some bidders to not be able to fully fill their bids as `clearingPrice` might increase over their `maxPrice` values.

Since preventing the aforementioned address from bidding would not resolve the situation, as sibling addresses controlled by the deployer could be used, consider documenting this to alert bidders about this behavior.

## M-02 Step Not Advanced When `blockNumber == step.end`, Causing Incorrect MPS

Whenever a checkpoint is saved, missed steps are reconciled with the `_advanceToCurrentStep` function of the `Auction` contract. Each step's `start` is set to the previous step's `end`, making the step's end block exclusive. However, when `blockNumber == step.end`, the `while` loop does not run and the step is not advanced.

At the next checkpoint, the `clearingPrice` value is computed using the previous step's `mps`, even though the next step has already begun. In cases where the new step's `mps` is lower and demand/price adjustments are required, bidders may place bids at a stale price and then be instantly outbid at the next checkpoint, causing their bids to be "filled" incorrectly.

Consider checking whether `lastCheckpointedBlock == step.end` and advancing the step to prevent using stale values when performing any calculation.

## M-03 Effective Demand Reduced Due to Rounding When Bids Are Split

Whenever a user places a bid, the effective demand is calculated as `amount * MPS / mpsDenominator`. In cases where the currency token has fewer decimals and a high unit price, this can lead to significant rounding down. This creates an incentive for users to split their bids into smaller amounts, as repeated smaller bids can yield a higher effective demand than a single larger bid. The issue is especially pronounced with expensive, low-decimal tokens such as WBTC (8 decimals).

For instance, a single bid of `0.0001 BTC` (\$11.27) repeated 100 times (\$1,127 total), with a `mpsDenominator` of 312243, results in an effective demand of **32,000**, while placing the same \$1,127 bid in one transaction results in an effective demand of **32,026**. Therefore, splitting into smaller bids could cause incorrect demand and price adjustments over time.

Consider normalizing the calculation to minimize rounding discrepancies, for example by using higher-precision arithmetic (scaling factors), or by enforcing a minimum bid size to prevent splitting into artificially small bids.

## M-04 Intermediate Steps' Contributions Dismissed When Last `step.mps == 0`

In the `Auction` contract, when the current `step.mps` value equals zero, then the logic advances the step (which updates it in the global variable), but the returned `checkpoint` with the transformed accumulators is dismissed.

However, in the scenario in which the latest `checkpoint` was originally in a step that had a zero `mps` and no bidders submitted a bid or checkpointed during several non-zero-`mps` steps, when the new checkpoint is being done, the logic will dismiss the new checkpoint after updating the current step, and all the contribution from the non-zero `mps` steps will be dismissed. This means that sums and clearing prices will be performed using incomplete accumulators, deviating from the expected values.

Consider taking into account the contribution of non-zero-`mps` steps under the conditions mentioned above.

## M-05 Clearing Price Can Drop Below Floor Price Due to Missing Validation

When the `Auction` contract is deployed, the `floorPrice` and `tickSpacing` parameters are set in the `constructor` function of the `TickStorage` contract. However, there is no validation that `floorPrice` lies on a tick boundary.

When the new clearing price is computed, it is compared against the `floorPrice` and the minimum clearing price, and then normalized to a tick boundary. If `floorPrice % tickSpacing != 0`, this normalization can push the finalized clearing price below the intended floor, violating the invariant.

For example, with `tickSpacing = 15` and a misaligned `floorPrice = 1e18`, a computed `newClearingPrice` value that is slightly above the floor (e.g., `1e18 + 2`) will normalize down to `1e18 - 10`, ending below the floor and breaking the invariant.

Consider validating that `floorPrice % tickSpacing == 0` during the deployment and, when calculating the new price, comparing against the tick-aligned (normalized) value.

## M-06 Auction Deadlock if Trailing Steps Have Zero `mps`

At each checkpoint, the per-block supply is computed as `(unclearedSupply * mps) / unclearedMps`. If the setup of the steps has trailing zero-`mps` steps (the totality of the tokens were distributed before the auction ends) then the calculation `AuctionStepLib.MPS - checkpoint.cumulativeMps == 0`, making the `unclearedMps` value zero and triggering a division-by-zero revert. This means that any function that internally calls the `checkpoint` function (e.g., `exitBid`, `partiallyExitBid`, `sweepCurrency`, `sweepUnsoldTokens`) will revert, effectively deadlocking the auction: bidders cannot exit, and the auction creator cannot sweep tokens or currency.

Consider either validating that `AuctionStepLib.MPS - checkpoint.cumulativeMps` is not 0 before calculating the supply with the aforementioned step configuration, or restricting the step configuration to not have zero-`mps` trailing steps.

# Low Severity

## L-01 Duplicate Initialization of Immutable Variables in Constructors

The immutable variables defined in `TokenCurrencyStorage` are initialized twice—once in `Auction`'s constructor and again within `TokenCurrencyStorage`. Since `TokenCurrencyStorage` already handles this, the assignments in `Auction` are redundant and should be removed.

Consider removing the redundant assignments.

## L-02 Movement Of Native Tokens Might Get Stuck

The `TokenCurrencyStorage` contract implements a sweeping functionality to allow the respective `fundsRecipient` and `tokensRecipient` addresses to get the resulting balances from the auction into their addresses. Since the contract is using the `transfer` function from the `CurrencyLibrary` library, when the asset involved is the native token, the transfer will be performed using a low-level assembly call. However, in the scenario where the destination address either is a contract or an EIP-7702 account that does not implement a `receive` or payable `fallback` function, this call will revert, preventing the final sweeping from happening.

To prevent getting the native token stuck in the `Auction` contract, consider validating that the `fundsRecipient` and `tokensRecipient` addresses are able to receive such assets in the first place, either with introspection or by sending a dust in native tokens during deployment to validate that it gets received without issues.

## L-03 Step Data Is Indirectly and Not Fully Validated

When deploying a new `Auction` contract, the `AuctionStepStorage` contract stores the `startBlock` and `endBlock` ranges, and takes the step's data to write them in storage and validate them.

However, the validation is done over the written data, and the read data is not directly compared to the original `_auctionStepsData` input. The following validations are performed:

- The recovered data has a length of zero
- The recovered data is not a multiple of 8 bytes (packed data per step)
- The recovered data does not have the same length as the original `_auctionStepsData` input
- The `blockDelta` sum of all recovered steps matches the difference between `startBlock` and `endBlock`
- The cumulative sum of all the `mps` adds up to `MPS`

In case the dependency used to write the data causes or crafts the stored data in a way that keeps the same number of steps but alters each step `blockDelta` and `mps` in such a way that it can keep the last 2 checks from the list, the `Auction` contract might use step data that might not match the expected one.

Moreover, the period ranges for the overall auction depend on the sum of all the `blockDelta`s. If the ranges were wrongly set, and the data recovered from storage recreates a different sum, the validation might allow the erroneous data to pass through instead of reverting.

Even though the situation is unlikely to materialize, consider validating that the stored data using the `SSTORE2` library matches the initial input from the constructor.

## L-04 Currency Could Get Stuck in Auction

The `TokenCurrencyStorage` contract implements the `_sweepCurrency` function which transfers the respective currency assets to the `fundsRecipient` address, defined during the deployment of the `Auction` contract. Along with this parameter, `fundsRecipientData` is also set. However, as the `fundsRecipient` address is not being validated beyond not being the zero address, this address might be a contract or an EIP-7702 address that does not implement the correct logic to execute the `fundsRecipientData` data. In such a scenario, the call will revert, and the currency assets will get stuck in the `Auction` contract because there is no other way to extract them.

Since `fundsRecipientData` cannot be changed after deployment, consider hardcoding the encoded function selector along with the required parameters to prevent incorrect data from being used during deployment, and validating if the `fundsRecipient` address is able to execute the same method used during the encoding by using an introspection standard such

as EIP-165. Furthermore, consider adding guidelines so that the `fundsRecipient` address has a general `fallback` function that would allow catch this call, regardless of the data contained in `fundsRecipientData`.

## L-05 Bidders Offering the Highest Prices Are Exposed to Front-Running Risks

When submitting a bid, a previous price hint must be provided in order to successfully link the new price to the existing chain. During tick initialization, it is validated that the provided hint does not point to a lower price. If it does, the transaction reverts, as another correct hint should be used instead. However, if a user's `maxPrice` is greater than the current last price, their transaction becomes vulnerable: an external actor can front-run the transaction and update `ticks[hint].next` to a value other than `MAX_TICK_PRICE`. This manipulation would cause the bidder's transaction to fail, as the expected linkage would no longer be valid.

Similarly, when dealing with chained ticks outside the extreme case, `ticks[hint].next` might get outdated and could end up being less than the `maxPrice` if another bid with a lower `maxPrice` (price that still does not have a tick) is mined first. In such a case, normal congestion of bidders could cause reversion of the bids present in the mempool that get mined afterwards.

Consider passing the lower tick hint and iterating until finding the correct place for inserting the new `maxPrice` value to mitigate both situations.

## L-06 Auction Setup Is Not Time-Bounded

During the deployment of a new `Auction` contract, the `startBlock` and `endBlock` values (which define the lifespan of the auction) are set as immutable values. Even though there is a validation done by comparing them against the sum of all decoded steps, there is no limitation on the duration of the auction. As a result, if the auction is incorrectly set up, a bidder bidding in the auction might need to wait until its expiration to be able to take back the assets.

In addition, since it is expected that most bidders might skip participating in such an auction, the auction's demand will not be sufficient to increase the `clearingPrice` value that would allow the bidder to exit the auction sooner with the `exitPartiallyFilledBid` function. Moreover, in case the auction was wrongly constructed and no one bids due to its duration, token assets from the owner of the `Auction` contract will also need to wait until its finalization to be swept back to the respective address, as there is no way to cancel the ongoing auction.

Consider implementing and setting a reasonable maximum duration of the auction, or documenting this behavior. Doing so will help auction deployers and bidders verify that the duration is the expected one.

## L-07 Transfer Call On Contract With Fallback Might Succeed

The `CurrencyLibrary` library implements the `transfer` method that sends either the native token or an arbitrary token to a destination address. When the token being transferred is not the native token, the `transfer` function call is made and later validated. As a result of this validation, either 1 (`true`) is returned and `returndatasize` is 32 bytes or more, or nothing is returned.

The latter case (i.e., nothing is returned) might catch tokens that do not return the boolean after completion. However, this also creates a possibility of calling a non-token contract that implements the `fallback` function, that could catch the `transfer` function call, which would continue the execution of the contract calling this method (e.g., the `Auction` and `TokenCurrencyStorage` contracts).

Even though the currency that should be called is already parsed during the initial setup of the auction, the `CurrencyLibrary` could still end up being used improperly in different projects, resulting in unexpected behavior.

Consider either documenting this possible succeeding behavior when it comes to non-token contracts, or prohibiting it by using introspection on the currency called to assert that it does implement the `transfer` functionality from an ERC-20 token.

## L-08 Steps Configuration Can Be Inappropriately Setup

The `AuctionStepStorage` contract implements logic to store the steps configuration as a separate contract, which will then be read in a more efficient manner. It also includes validations done over the sum of all the steps, which cannot exceed the MPS (100%) of the funds, and the block delta, which needs to match the independent `startBlock` and `endBlock` values.

However, it is worth noting that in case there are singularity steps that do not have any block delta, the `mps` value for such steps can be anything, as the `mps` will not contribute towards the sum being checked.

Even though most of the calculations use the `mps` value along with the respective `blockDelta`, having a non-zero `mps` that will not contribute due to the singularity step does create some attack surface.

In order to prevent such scenarios and reduce the attack surface, consider either reverting in case of a singularity step (a delta of zero) or enforcing that in case of such steps, the `mps` value must be also zero.

## L-09 Accumulator Variables Not Set to Zero

In the `AuctionStepStorage` contract, the `sumMps` and `sumBlockDelta` variables are used to add the partial contribution of each step towards the `mps` and `blockDelta` they have. As these variables add the new partial contribution to the previous value, these should be initialized to zero to reduce the attack surface that could allow for the validation of an erroneous step setup.

Consider setting the `sumMps` and `sumBlockDelta` variables to zero.

## L-10 Over-Deposited Tokens Cannot Be Rescued

The `Auction` contract implements the functionality to bid currency assets and receive token assets, depending on the `maxPrice` that has been set. To keep track of the balances, demand, and prices, the protocol uses an internal accounting system that does not take into consideration the actual balances of the contract. During the deployment of the `Auction` contract, the `AuctionFactory` contract passes its arguments, among which is the `totalSupply` immutable value of tokens that the `Auction` contract will have.

When using the Token Launcher workflow, the `onTokensReceived` function is called to validate that the balance of the auction is not less than the parameter passed during deployment. However, when sending more tokens than `totalSupply`, the validation will pass, yet the excess tokens will not count towards the internal balances. Also, as the sweeping mechanisms make use of the internal accounting, any over-deposit above the `totalSupply` value will be stuck in the `Auction` contract, as there is no function that could help rescue such funds. In addition, if excess tokens are added by mistake (users participating in the

auction or the initial deployer of the auction), these will also not be able to be rescued, even if among them is the token used for this `Auction` contract or a different one.

In order to prevent assets from getting stuck, consider taking into account all the token balances in the `Auction` contract and implementing a mechanism to rescue funds that are not accounted for in the auction.

## L-11 Native Token Can Get Stuck During Bidding

In the `Auction` contract, the `submitBid` function allows users to submit their bids, depositing the respective assets. As the native token can be used as currency, it is represented with the zero address and `msg.value` is compared against the required amount. When using a non-native token, the protocol uses permit2 to move the funds into the contract.

However, as the same `payable` function is used in both situations (i.e., native and non-native tokens), if the auction is meant to have a non-native token as currency, users can still pass the native token along with the call to the `submitBid` function even though it will then transfer the non-native token with permit2. Since the native token is not accounted for in any sense, native assets will get stuck in the contract.

Consider requiring that `msg.value` is zero when the currency is not the native token.

## L-12 Rounding-to-Zero Supply Can Prevent Auction Graduation

When adjusting demand above the clearing price, the `Auction` contract computes the current block's supply as `totalSupply * mps / MPS`. Due to integer division, this can evaluate to zero, which causes both the demand adjustment and the clearing price update to be skipped, effectively pinning the auction at the floor price and preventing it from graduating.

In the `_validate` function of the `AuctionStepStorage` contract, consider enforcing that `totalSupply * mps / MPS` must be greater than zero for non-zero `mps` so that steps cannot round down to zero supply.

## L-13 Different Pragma Directives Are Used

In order to clearly identify the Solidity version with which the contracts will be compiled, pragma directives should be fixed and consistent across file imports.

Throughout the codebase, multiple instances of varying pragma directives were identified, such as the `pragma solidity 0.8.26;` directive in `Auction.sol`, or the `pragma solidity ^0.8.0;` directive in `CurrencyLibrary.sol`.

Consider using the same, fixed pragma version across all the files.

## L-14 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Throughout the codebase, multiple instances of floating pragma directives were identified:

- `IAuction.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IAuctionFactory.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IAuctionStepStorage.sol` has the `solidity ^0.8.0` floating pragma directive.
- `ICheckpointStorage.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IPermitSingleForwarder.sol` has the `solidity ^0.8.0` floating pragma directive.
- `ITickStorage.sol` has the `solidity ^0.8.0` floating pragma directive.
- `ITokenCurrencyStorage.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IValidationHook.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IDistributionContract.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IDistributionStrategy.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IERC20Minimal.sol` has the `solidity ^0.8.0` floating pragma directive.
- `AuctionStepLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- `BidLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- `CheckpointLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- `CurrencyLibrary.sol` has the `solidity ^0.8.0` floating pragma directive.
- `DemandLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- `FixedPoint96.sol` has the `solidity ^0.8.0` floating pragma directive.

Consider using fixed pragma directives.

## L-15 Missing Zero-Address Checks

When operations with address parameters are performed, it is crucial to ensure the address is not set to zero. Setting an address to zero is problematic because it has special burn/renounce semantics. As such, this action should be handled by a separate function to prevent accidental loss of access during value or ownership transfers.

The `_tokensRecipient` input within the `TokenCurrencyStorage` contract does not validate if it is the zero address or not, which could end up burning the unsold tokens after the auction ends.

If this behavior is intentional, consider either adding a zero-address check before assigning the state variable or documenting the rationale for omitting it.

## L-16 Missing Auto-Checkpoint When Partially Exiting

Users are expected to partially exit as soon as they are outbid via the `partiallyExitBid` function in which they can get a refund for the un-filled currency. However, this function does not call the `checkpoint` function internally, which can lead to cases where the next checkpoint's price exceeds a bid's max price. In such situations, users must first call the `checkpoint` function to update the internal variables, and then call the `partiallyExitBid` function, resulting in a two-step (plus later claiming tokens) exit instead of a single action.

Consider invoking the `checkpoint` function whenever `block.number <= endBlock` to enable one-step exits.

## L-17 Abstract Contracts Allow Direct Modification of State Variables

`internal` and `public` state variables in `abstract` contracts allow them to be directly modified by child contracts. This may break the expected properties for the state variables and limit off-chain monitoring capabilities due to the lack of event emissions for changes to the variables.

Throughout the codebase, multiple instances of `abstract` contracts containing `public` or `internal` state variables were identified:

- The `pointer` state variable in `AuctionStepStorage` is `public`.
- The `offset` state variable in `AuctionStepStorage` is `public`.
- The `step` state variable in `AuctionStepStorage` is `public`.
- The `nextBidId` state variable in `BidStorage` is `public`.
- The `bids` state variable in `BidStorage` is `public`.
- The `checkpoints` state variable in `CheckpointStorage` is `public`.
- The `lastCheckpointedBlock` state variable in `CheckpointStorage` is `public`.
- The `ticks` state variable in `TickStorage` is `public`.
- The `nextActiveTickPrice` state variable in `TickStorage` is `public`.
- The `sweepCurrencyBlock` state variable in `TokenCurrencyStorage` is `public`.
- The `sweepUnsoldTokensBlock` state variable in `TokenCurrencyStorage` is `public`.
- The `fundsRecipientData` state variable in `TokenCurrencyStorage` is `public`.

Consider using `private` visibility for state variables in abstract contracts. Additionally, consider creating `internal` functions for updating these variables that emit appropriate events, and verifying if the desirable conditions are met.

## L-18 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `AuctionFactory.sol`, the `USE_MSG_SENDER` state variable
- In `BidStorage.sol`, the `BidStorage` abstract contract
- In `CheckpointStorage.sol`, the `MAX_BLOCK_NUMBER` state variable
- In `TickStorage.sol`, the `ticks` state variable
- In `IAuctionStepStorage.sol`, the `IAuctionStepStorage` interface
- In `ICheckpointStorage.sol`, the `ICheckpointStorage` interface
- In `ITokenCurrencyStorage.sol`, the `ITokenCurrencyStorage` interface
- In `IValidationHook.sol`, the `IValidationHook` interface
- In `AuctionStepLib.sol`, the `AuctionStepLib` library
- In `AuctionStepLib.sol`, the `MPS` state variable
- In `BidLib.sol`, the `PRECISION` state variable
- In `DemandLib.sol`, the `DemandLib` library

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).

## L-19 Graduation Calculation Allows for Premature Graduation

In the `Auction` contract, the `isGraduated` function compares the `totalCleared` value in the last checkpoint against the minimum supply sold accepted based on the `graduationThresholdMps` parameter. However, because the formula uses the `fullMulDiv` method, which rounds down, the minimum accepted value after the calculation is also rounded down, meaning that it would need to sell less tokens to be considered a successful auction.

Even though the difference would be probably small, this could affect the final outcome, the expected currency in exchange for the tokens, or the supplied tokens after the sale. As such, consider rounding the result up.

## L-20 Silent Truncation to `uint128` in `resolveCurrencyDemand` Miscomputes Demand

In the `DemandLib` library, the `resolveCurrencyDemand` function computes `amount * 2**96 / price` (with `price` in Q96) and then narrows the quotient down to `uint128` without a bounds check. If the true result exceeds `type(uint128).max`, the cast discards the high bits (i.e., modulo `2**128`), silently under-reporting demand and distorting accounting/pricing.

For example, with `amount = 2**120 + 1` and `price = 2**80` (Q96 = 2^-16), the demand would be `(2**120 + 1) * 2**96 / 2**80 = 2**136`. However, since the output is cast to `uint128`, the function would return `65536`.

Consider using a safe cast mechanism to prevent demand from being miscalculated.

## L-21 Misleading Docstrings

Throughout the codebase, multiple instances of misleading docstrings were identified:

- In line 15 of `BidLib.sol`, the comment for the owner field states "Who is allowed to exit the bid", but any address can exit someone's bid, not just the owner.
- In lines 139-140 of `IAuction.sol`, the docstring states that the function "can only be called by the funds recipient after the auction has ended" and "must be called before the claimBlock". However, the implementation neither restricts the caller to the funds recipient nor enforces that it be called after the auction ends.
- In line 82 of `CheckpointStorage.sol`, the comment states the function "calculates the tokens sold, proportion of input used, and the block number of the next checkpoint under the bid's max price". However, it only returns `tokensFilled` and `currencySpent`.

To improve code quality and maintainability, consider updating the docstrings to accurately reflect the actual behavior of the codebase.

## L-22 `graduationThresholdMps` Effect Decreases On Smaller Token Count

The `Auction` contract uses the `graduationThresholdMps` parameters to calculate if the auction successfully sold the minimum amount of token supply or not.

However, due to the truncation in the calculation, the effect of such parameter (that should have a range between 0 and `1e7`) over the threshold where the auction is graduated or not decreases when using tokens with low decimals or lower token counts.

For instance, when selling 9 units of token without decimals, the range for the `graduationThresholdMps` parameter between [8888889, 9999999] would put a graduation threshold of 8 tokens, even though there is a 11.11% difference between the bounds. As such, `Auction` contract deployers could abuse on this, move the parameter, and signal bidders a false expectation when using the extremes of the range (but without any effect over the threshold). Moreover, as this parameter is used in the `AuctionFactory` contract for the deployment, `Auction` contract that do not have the same `graduationThresholdMps` parameter might behave exactly alike during the operation.

Consider documenting this behavior or lowering/increasing the parameter to the previous/next tipping point where the threshold output has a different value.

# Notes & Additional Information

## N-01 Redundant Validation

In the `AuctionStepStorage` contract, once the step data is stored by Solady's `SSTORE2.write` function, a validation is performed over the pointer to assert that it is not zero. However, the `write` function from the aforementioned library already validates the same, as otherwise, it would revert.

Consider removing the duplicated validation.

## N-02 Inconsistent Refactor

In the `BidLib` library, the `demand` function uses the `resolveCurrencyDemand` function from the `DemandLib` library when the bid has an `exactIn == true` input. However, when it is the opposite scenario, it does not use the `resolveTokenDemand` function. Even though the latter just wraps the input, if the logic changes, the `demand` function might get outdated in its conversion, which could cause calculation errors. In addition, tackling the changes to the logic at different places could result in increased maintenance cost in the long run.

Consider using the `resolveTokenDemand` function, even if currently just wraps the input amount, to reduce the maintenance cost in case changes are made to the calculations.

## N-03 Redundant Definition of Variables

During the deployment of the `Auction` contract, a set of variables were defined in its `constructor` function. However, these same variables are defined once again in the inherited `TokenCurrencyStorage` contract. In addition, a few of the checks are also redundant. This not only increases the gas cost of the deployment, but it also might introduce bugs if some change is introduced, such as adjusting one input before assigning it, that could then get overwritten by the redundant assignment.

Consider removing the redundant assignments of and validations on the same variables to reduce the gas cost and to prevent maintenance issues.

## N-04 Inconsistent Asset Handling

In the `TokenCurrencyStorage` contract, the `token` input is stored as a `IERC20Minimal` type so that it can call the `balanceOf` function of the `Auction` contract directly during the validation of the auction's balance. However, there are a few cases ([1], [2]) where the `token` is cast into the `address` type and then wrapped as `Currency`. This casting and wrapping process increases the gas cost for bidders. Apart from this, since there is a nomenclature distinction in the protocol about the underlying token being sold and the currency used to pay for it, treating the token as `Currency` along with its `CurrencyLibrary` library could cause confusion.

In case a shared library is intended to be used, consider changing the name of the `CurrencyLibrary` library to `ssetsLibrary`, the `Currency` type to `Asset`, and then storing the `token` as `Asset` instead of `IERC20Minimal`. Doing so would also allow the `token` to use the `CurrencyLibrary` contract's `balanceOf` function.

## N-05 Redundant `CurrencyLibrary` Import

Since the `CurrencyLibrary` is applied to the `Currency` type globally via `using for` in `CurrencyLibrary.sol`, additional imports/`using` statements are unnecessary in contracts importing it.

Multiple instances of redundant import/`using` statements were identified in the `Auction` and `TokenCurrencyStorage` contracts.

Consider removing redundant imports to improve the readability and maintainability of the codebase.

## N-06 Unused Constant in `BidLib`

The `PRECISION` constant in `BidLib` is never used.

Consider removing any instances of unused constants to improve code clarity and maintainability.

## N-07 `AuctionStepStorage.pointer` Could Be Immutable

To store an auction's step data, the `AuctionStepStorage` contract uses the `SSTORE2` library which deploys its bytes as bytecode. Since this occurs in the `constructor` function of the aforementioned contract, and because the pointer is never modified afterwards, the `pointer` variable could be declared as `immutable`.

Consider marking the `pointer` variable as `immutable`.

## N-08 Unused Imports

Throughout the codebase, multiple instances of unused imports were identified:

- The import `import {AuctionStep} from './libraries/AuctionStepLib.sol';` in `CheckpointStorage.sol`
- The import `import {SafeCastLib} from 'solady/utils/SafeCastLib.sol';` in `CheckpointStorage.sol`
- The import `import {Bid} from './libraries/BidLib.sol';` in `TickStorage.sol`
- The import `import {FixedPoint96} from './libraries/FixedPoint96.sol';` in `TickStorage.sol`
- The import `import {Tick} from '../TickStorage.sol';` in `ITickStorage.sol`
- The import `import {Bid} from '../libraries/BidLib.sol';` in `IValidationHook.sol`
- The import `import {BidLib} from './BidLib.sol';` in `CheckpointLib.sol`
- The import `import {SafeCastLib} from 'solady/utils/SafeCastLib.sol';` in `BidLib.sol`
- The import `import {SafeTransferLib} from 'solady/utils/SafeTransferLib.sol';` in `CurrencyLibrary.sol`

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

## N-09 Event Emits Stale `configData` After In-Memory Modification

In the `initializeDistribution` function of the `AuctionFactory` contract, the factory replaces the `fundsRecipient` address with `msg.sender` when the input passes the `USE_MSG_SENDER` constant, which is then used to deploy the `Auction` contract. However, during the emission of the `AuctionCreated` event, the original `configData` (with the `USE_MSG_SENDER`) is used. As a result, logs may show `fundsRecipient = 0x0000000000000000000000000000000000000001` while the contract stores `msg.sender`, causing a persistent log/state mismatch.

Consider adjusting the `configData` to reflect it in the event.

## N-10 Unused Variable in `Auction` contract

The `Auction` contract defines and passes a hardcoded `PERMIT2` constant to its `PermitSingleForwarder` base constructor. However, this value is not used anywhere within the `Auction` contract.

To improve code readability and modularity, consider moving the `PERMIT2` instance to the `PermitSingleForwarder` contract.

## N-11 Incomplete Docstrings

Throughout the codebase, multiple instances of incomplete docstrings were identified:

- In `Auction.sol`, the docstrings above `isGraduated` function do not document return values.
- In `TickStorage.sol`, the docstrings above `getTick` function do not document return values.
- In `IAuction.sol`, the docstrings above `checkpoint` and `isGraduated` functions do not document return values.
- In `IAuctionStepStorage.sol`, the docstrings above `startBlock` and `endBlock` functions does not document return values.
- In `ICheckpointStorage.sol`, the docstrings above `latestCheckpoint`, `clearingPrice`, `currencyRaised` and `lastCheckpointedBlock` functions do not document return values.

- In `ITickStorage.sol`, the docstrings above `nextActiveTickPrice`, `floorPrice` and `tickSpacing` functions do not document return values.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

## N-12 Unnecessary Casts

Within `TokenCurrencyStorage.sol`, instances of unnecessary casts were identified:

- The `address(fundsRecipient)` cast
- The `address(fundsRecipient)` cast

To improve the overall clarity and intent of the codebase, consider removing any unnecessary casts.

## N-13 Inconsistent Use of Returns in a Function

To improve the readability of the function, use the same return style.

Within `Auction.sol`, there are multiple instances where functions have inconsistent usage of returns. For instance, the `_unsafeCheckpoint function` uses a named return parameter but no explicit return statement, whereas the `_getFinalCheckpoint function` uses a named return parameter which is never initialised and has a return statement.

Consider being consistent with the use of returns throughout the functions.

## N-14 Inconsistent Order Within Contracts

Throughout the codebase, there are multiple contracts that deviate from the Solidity Style Guide due to having inconsistent ordering of functions:

- The `Auction` contract.
- The `AuctionStepStorage` contract.
- The `IAuctionStepStorage` contract.

To improve the project's overall legibility, consider standardizing ordering throughout the codebase as recommended by the Solidity Style Guide (Order of functions).

## N-15 Lack of Security Contact

Embedding a dedicated security contact (email or ENS) in a smart contract streamlines vulnerability reporting by letting developers define the disclosure channel and avoid miscommunication. It also ensures third-party library maintainers can quickly reach the right person for fixes and guidance.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contract` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

## N-16 Non-Conforming Error Name

The `IDistributionContract__InvalidAmountReceived` error is not following the Solidity style guide. It should be named in CapWords style, such as `IDistributionContractInvalidAmountReceived`.

To improve the project's overall legibility, consider adhering to the Solidity style guide by naming the contracts, structs, enums, events, and errors using the CapWords style.

# Conclusion

The reviewed codebase implements a configurable, time-phased uniform clearing-price auction for token distributions, featuring programmable issuance schedules, max-price bidding, optional eligibility hooks, support for raising both ETH and ERC-20 tokens, and deterministic deployments. The system consists of the `AuctionFactory` contract, which acts as the entry point for the deployer to parse the parameters of the auction and get a new instance. The deployer can be the Token Launcher or an independent actor. The `Auction` contract is a core contract in the protocol and inherits logic from different areas of the account system: steps, bids, checkpoints, Permit2, ticks, and asset management. Each of these comes with its own respective library to handle the calculations or movement of assets.

The audit identified 3 critical- and 3 high-severity issues, along with additional lower-severity observations.

Overall, the system introduces a flexible solution for auction events that incorporates several particular characteristics, such as tick prices, data in `SSTORE2` contracts, and `permit2` currency handling. Even though, the development of the code aims in the right direction, several items seen in the audit should be resolved and improved to reach a more robust and production-ready state. In particular, the protocol could benefits from simplifying and reducing the complexity of the calculations of the demand, supply, and checkpoint transformations while, at the same time, restricting degrees of freedom, namely hints or setup of auction's configuration, to prevent the erroneous or malicious usage of its inputs. Moreover, the codebase was initially though to be used alongside the Token Launcher protocol, but when its use-case is extended as standalone, this results in a lack of essential validations that are not enforced by default in the TWAP Auction protocol. Sensible addresses participating from the protocol (asset recipients and validation hook) do not pose limitations, which can end up in the possibility of assets getting stuck. Also, its time-dependent accounting introduces complexity that increases the attack surface for this use case.

In addition, the partial refactors with leftover or redundant logic, and misleading or missing documentation suggest that the overall codebase could be improved to reach a more production-ready state. Even though, the protocol implements good-quality tests, additional test cases increasing the number of actions could be added that would cover more complex attacks or edge scenarios in the operation of the protocol. Upon addressing the issues

mentioned in the report, and adding changes in its roadmap, it would be recommended to follow another round of audits to assert its robustness.

The Uniswap team is appreciated for their exceptional collaboration and responsiveness throughout the whole process, which significantly streamlined the review of the different versions of the code.