OpenZeppelin | security

Uniswap Token Launcher Audit

October 1, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-08-26 To 2025-09-05
Languages	Solidity

Total Issues	28 (25 resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	7 (6 resolved)
Low Severity Issues	9 (7 resolved)
Notes & Additional Information	9 (9 resolved)

Scope

OpenZeppelin audited the Uniswap/token-launcher repository at commit 9eff2a7.

In scope were the following files:

```
src
├── distributionContracts
│   ├── LBPStrategyBasic
│   └── MerkleClaim
└── distributionStrategies
│   ├── LBPStrategyBasicFactory
│   └── MerkleFactory
└── interaces
│   └── external
│       └── IERC20
│   ├── IDistributionContract
│   ├── IDistributionStrategy
│   ├── ILBPStrategyBasic
│   ├── IMerkleClaim
│   ├── IMultiCall
│   ├── IPermit2Forwarder
│   └── ITokenLauncher
└── libraries
│   └── TickCalculations
└── types
│   ├── Distribution
│   └── MigratorParams
└── utils
│   └── HookBasic
├── Multicall
├── Permit2Forwarder
└── TokenLauncher
```

System Overview

The Token Launcher is a token-distribution and liquidity-bootstrapping system built on Uniswap V4 that orchestrates token distribution through configurable strategies. It aims to provide fair token distribution and immediate liquidity by combining price discovery with automated liquidity deployment. Whether a team creates a new token via an external factory or uses an existing token, the launcher supports the complete distribution and liquidity workflow.

`TokenLauncher`

The `TokenLauncher` contract coordinates a launch end to end. Teams may first create a token via an external factory (optional) or use an existing token. A launch starts by calling `distributeToken()` with a strategy configuration. The launcher then:

1. deploys the selected strategy via its factory
2. transfers the configured token amount to that strategy
3. lets the strategy run its process

This keeps the launcher simple and consistent while allowing each strategy to implement its own logic behind a shared interface.

Distribution Strategies

`LBPStrategyBasic` **- Price Discovery and Liquidity Bootstrapping**: This strategy splits the token supply into two parts: one part participates in an external TWAP auction to discover a fair market price, and the other part is reserved to provide liquidity on Uniswap V4 after the auction. At least 50% of the tokens are reserved for liquidity (so up to 50% can be auctioned). `LBPStrategyBasic` is implemented as a Uniswap V4 hook that prevents pool initialization until the auction results are validated. When the auction completes, it calls the strategy's `validate()` function, which reads the clearing price and currency raised, checks them against Uniswap V4 constraints, and stores them for migration. After a short, configured block delay, anyone can call `migrate()` to initialize a Uniswap V4 pool at the discovered price which deploys both full-range and potentially additional one-sided liquidity positions by utilizing the currency raised and the tokens that are reserved in the contract.

`MerkleClaim` - **Claim Based Distribution**: `MerkleClaim` publishes a Merkle root on-chain and lets recipients claim allocations by providing Merkle proofs. It scales to large recipient sets and fits to airdrops or predetermined distributions where price discovery is not required.

Extensibility: The distribution layer is designed to be pluggable. New strategies can be added without modifying the launcher by implementing the shared interfaces and, where needed, supplying a factory for deterministic deployment. Projects can use a single strategy or combine multiple strategies for the same token.

Supporting Components

The system relies on a small set of helper contracts and integrations. Strategy factories deploy strategy instances, `Permit2Forwarder` handles permit-based approvals and transfers, and `Multicall` allows batching when needed. Uniswap V4 integration is built into the primary path: `LBPStrategyBasic` itself is the Uniswap V4 hook that gates pool initialization and enforces validated auction parameters, preventing premature or malicious pool creation.

Pool deployment and liquidity positions rely on Uniswap V4 core/periphery. The external TWAP auction supplies the clearing price and raised currency used during migration. Optionally, teams can use external token factory contracts when creating a new token before distribution. Otherwise, existing tokens flow through the same launcher interface.

Security Model and Trust Assumptions

During the review, the following trust assumptions were made:

- The `Auction` contract will never reach an irretrievable state.
- The `Auction` contract's interface will not change.
- The `Auction` contract will always call the `validate` function if an auction is successful.
- The `Auction` contract works as intended.
- The `positionRecipient` of the liquidity position created after pool initialization will not rug.
- The token creator will not rug after pool creation.
- The out-of-scope dependencies work as intended.

Recommendations on Integration and Invariant Testing

This codebase has multiple integrations but lacks a strong integration and invariants test suite. The review uncovered numerous issues in interactions with the out-of-scope Auction contracts. Since both codebases are under active development, a robust test suite is essential to prevent edge-case fund loss or denial-of-service bugs once deployed on-chain. Additionally, several basic happy and unhappy paths were found to be broken, further underscoring the need for comprehensive testing.

Privileged Roles

Throughout the in-scope codebase, the following privileged roles/actions were identified:

- The `MerkleClaim` contract has an `owner` that can call the `sweep` and `withdraw` functions to take out the remaining tokens after the claim duration has ended.
- The `validate` function of the `LBPStrategyBasic` contract can only be called by the `Auction` contract.

Critical Severity

C-01 `reserveSupply` Gets Stuck in `LBPStrategyBasic` if `Auction` Does Not Graduate

When `distributeToken` is called on the `TokenLauncher` contract, it calls `LBPStrategyBasicFactory` to deploy the `LBPStrategyBasic` contract, sends tokens to the deployed contract, and calls `onTokensReceived` on it which deploys the `Auction` contract. The auction then takes place according to the parameters supplied to it during construction.

If the auction sales cross a certain threshold of the total available tokens, then the auction is considered graduated. After the auction is over, if it was successful (it graduated), bidders can claim the tokens that they bid for and anyone can transfer the raised funds to the `fundsRecipient`. The `sweepCurrency` function transfers the funds to the `fundsRecipient` and calls it if the `fundsRecipientData` is not empty and `fundsRecipient` has code. If the auction does not graduate, then the raised currency is refunded to the bidders and the total unsold supply of tokens on auction can be transferred to the `tokensRecipient`.

The `LBPStrategyBasic` requires the `fundsRecipient` to be itself and `fundsRecipientData` to be the function selector of the `validate` function to work correctly. This way, when the auction graduates, the `validate` function will be called and the raised funds will be transferred to it, so that a new pool can be created. However, if the auction does not graduate, then the `validate` function will not be called. This would result in the `reserveSupply`, which was reserved for pool creation, to get stuck as there is no way to get it out.

Consider adding a withdrawal function to get the `reserveSupply` out if the auction does not graduate.

Update: *Resolved in pull request #51 by adding a `sweepToken` and `sweepCurrency` function that an operator address can call at and after `sweepBlock` to recover funds. Its important to be noted that the `sweepBlock` can be immediately after the `migrationBlock` allowing the operator to sweep all funds before someone can call the `migrate` function.*

High Severity

H-01 Insufficient Input Validation During Contract Creation Can Break Normal Usage Flow

Without a successful `validate` function execution, migration to pool cannot occur, and the `validate` function can only be called by the `Auction` contract. It is expected to be invoked through the `sweepCurrency` function of the `Auction` contract after the auction is graduated. When `sweepCurrency` is called, accumulated currency in `Auction` will be transferred to `fundsRecipient`. `fundsRecipient` address will then be called with `fundsRecipientData` if it was provided during the auction deployment.

After this step, `sweepUnsoldTokens` from `Auction` contract should be called, which transfers the unsold tokens in Auction to `tokensRecipient`. According to the expected workflow, `fundsRecipient` should correspond to the `LBPStrategyBasic` contract, and `fundsRecipientData` should be a call to the `validate` function, while `tokensRecipient` should be the distribution owner who provided the tokens to the auction.

However, the `fundsRecipient`, `fundsRecipientData`, and `tokensRecipient` parameters are all user-supplied, and there is no validation of their values. Due to this lack of checks, if an address other than the `LBPStrategyBasic` is provided as the `fundsRecipient` along with an empty `fundsRecipientData`, all accumulated `currency` will be transferred to this address, the migration would not be possible and the `reserveSupply` would get stuck. In addition, a normal user might provide `tokensRecipient` as the `LBPStrategyBasic` address to be consistent with other variables. Since the `LBPStrategyBasic` contract does not have a way to handle these tokens, this would lead to tokens being locked in the contract.

Consider hardcoding `fundsRecipient` to the `LBPStrategyBasic` address and `fundsRecipientData` to the `validate` function selector during auction creation. Moreover, consider preventing `tokensRecipient` from being set to the `LBPStrategyBasic` contract.

Update: *Resolved in pull request #49. The team stated:*

> *Fixed by validating that the `fundsRecipient` in AuctionParameters data is set to the `LBPStrategy`.*

H-02 Signature Mismatch between Interface and Function Implementation

> ***NOTE:*** This issue was found in pull request #5 at commit 243bad9 which included the `MerkleClaim`, `MerkleFactory`, and `IMerkleClaim` files. This pull request was initially part of the review scope. After this issue was reported and fixed, the pull request was merged, the frozen commit was changed to 9eff2a7, and the review was continued with the combined scope.

The `MerkleClaimFactory` contract inherits from `IDistributionStrategy`. However, one of the inherited functions has a signature mismatch, which prevents the contract from being deployed.

The function with the mismatch is `initializeDistribution`. The mismatch specifically arises due to the following reasons:

- The function is missing the `bytes32 salt` parameter as the last parameter.
- The second parameter in the interface is `uint128 totalSupply`, whereas in `MerkleClaimFactory`, it is `uint256 amount`.

These differences result in both a parameter-count mismatch and a parameter-type mismatch, causing the function signature to diverge.

Consider adding the missing `bytes32 salt` parameter and aligning the second parameter's type with the interface.

Update: *Resolved in pull request #5. The team stated:*

> *Fixed, once we fixed foundry remapping issues and were able to test.*

Medium Severity

M-01 Overflow Possibility In the `validate` Function

The `validate` function of the `LBPStrategyBasic` contract fetches `clearingPrice` from the `Auction` contract. `clearingPrice` is the ratio of currency/token and is used for calculating the amount of both assets to be provided to the liquidity pool during its creation.

`clearingPrice` is computed and stored as a `uint256` value in the `Auction` contract and is returned as such. If the `clearingPrice` returned is greater than `type(uint160).max`, it will cause a silent overflow on this line in the `validate` function. This will cause the price to become a very small value which can either cause the pool creation to fail or, if it succeeds, result in the pool being initialized with the wrong price.

Consider adding a check to ensure that the price does not exceed `type(uint160).max` and handling it in the same manner as the other validation checks recommended for the `validate` function.

Update: *Resolved in pull request #48.*

M-02 Lack of Access Control in `distributeToken`

Initial supply of the token is minted to the `recipient` address provided as a parameter to `createToken` function. Hence, the caller of this function can choose to mint the tokens to any arbitrary address. The `distributeToken` function allows for choosing between two paths that can be used to extract the token using the `payerIsUser` parameter: it can be sent from `msg.sender`, or tokens from `TokenLauncher` can be utilized directly. However, if the tokens are minted to `TokenLauncher`, it is possible for any user to call `distributeToken` with an arbitrary `distribution` parameter because there is no access control in the `distributeToken` function.

Consider implementing access control for the `distributeToken` function. The token's `graffiti` can be utilized for such a restriction.

Update: *Acknowledged, not resolved in pull request #52. The team stated:*

> *This is by design - documented that create and distribute should only be called in a multicall with `payerIsUser` as `false`.*

M-03 `sweep` Function Not Usable

The `sweep` function of the `MerkleClaim` contract is designed to transfer any remaining tokens from the distribution to the `owner` after `endTime`. According to the documentation, this function should only be callable by the owner.

While `sweep` can be called by anyone, it executes `this.withdraw()` within its function body which is an external call to `withdraw` function from parent contract `MerkleDistributorWithDeadline`. However, `MerkleDistributorWithDeadline` has an `onlyOwner` modifier for `withdraw`, so it is only callable by the `owner` address. Since the call is made via `this.withdraw()`, the caller is set to `address(this)`, which prevents the owner from being able to call `sweep()`.

The only possible workaround would be to set the owner as `address(this)`. However, this approach contradicts the expected role of the `owner` and allows anyone to call `sweep()` successfully. While the parent contract's `withdraw` function remains available and can still be called directly by the `owner` (as it is not overridden), the `sweep` function itself does not behave as intended and is effectively unusable.

Consider performing a `delegatecall` to `address(this)` within `sweep`.

Update: *Resolved in pull request #21 by removing the `sweep` function completely.*

M-04 Lack of Hashing in `MerkleFactory` Can Lead to Front-Running of the Initialization

The `distributeToken` function of the `TokenLauncher` contract hashes `msg.sender` with the provided `salt` parameter to derive the final `salt` used in `initializeDistribution` within `MerkleFactory`. However, a malicious user can front-run this call by directly invoking `initializeDistribution` from `MerkleFactory` with the same salt and parameters. This would deploy a `MerkleClaim` contract at the same address, causing the subsequent `distributeToken` call to revert.

Consider hashing the `salt` received from `TokenLauncher` once more inside `initializeDistribution` combined with `msg.sender`.

Update: *Resolved in pull request #26.*

M-05 `validate` Function Reverting Will Lead to Loss of Funds

When an auction graduates, anyone can transfer the raised funds to the `fundsRecipient` of the auction by calling the `sweepCurrency` function in `Auction.sol`. For the pool creation to work, the `fundsRecipient` needs to be the `LBPStrategyBasic` contract, and the `fundsRecipientData` needs to be the function selector of the `validate` function.

If this call to the `validate` function fails, then the `sweepCurrency` function will also revert. This would lead to two problems: the funds raised in the auction will get stuck in the `Auction` contract, unable to be retrieved, and the `reserveSupply` will get stuck in the `LBPStrategyBasic` contract as a liquidity pool could not be created.

The mitigation of this issue is not straightforward and would possibly require changes in both the `LBPStrategyBasic` and the `Auction` contracts. One recommendation is presented below:

- Change the function signature of the `validate` function to return a boolean. While doing checks inside the function, if a condition fails, instead of reverting, just store a flag to ensure that the pool cannot be created and return `false`. Send the currency back to the Auction.
- Make a withdrawal function inside `LBPStrategyBasic` where all the reserve tokens can be withdrawn if the `validate` function has failed.
- Refund the users on the `Auction` side.

Consider implementing a mitigation that would prevent funds from getting stuck as a result of the `validate` function reverting.

Update: *Resolved in pull request #42. The team stated:*

> *Fixed by removing the `validate` function.*

M-06 Rebasing Tokens and Fee-on-Transfer Tokens Are Not Compatible with `TokenLauncher`

The `TokenLauncher` contract has been designed to work with any ERC-20 token. However, using fee-on-transfer or rebasing tokens can exhibit unexpected behavior and may result in locked funds. Fee-on-transfer tokens will likely fail during strategy initialization because of the `onTokensReceived` check implemented in strategies. The amount sent from the `TokenLauncher` contract will not match the amount received by the strategy. Since `onTokensReceived` is called in `distributeToken()`, the token distribution will revert.

Rebasing tokens can pass the initialization but may break later stages:

1. In `LBPStrategyBasic`, `reserveSupply` holds the portion of tokens reserved for liquidity. During migration, this amount is sent to the `positionManager`. If rebasing reduces the contract's balance below `reserveSupply`, the `migrate()` call will fail, preventing pool creation and locking both the tokens and the currency in the strategy.

2. In `MerkleClaim`, allocations are fixed at deployment. If rebasing reduces the overall supply, there may not be enough tokens left for later claimants, leading to failed claims.

Consider disallowing fee-on-transfer and rebasing tokens, and clearly documenting the aforementioned risks for users.

Update: *Resolved in pull request #52. The team stated:*

> *Documented that rebasing and fee on transfer tokens are not compatible.*

M-07 Tokens Lost to the `PositionManager` in `_createOneSidedPositionPlan`

The `LBPStrategyBasic` contract creates a single-sided position if excess `token` assets will be left in it after the creation of the full-range liquidity position. The calculation of the specifics of the single-sided position happens in the `_createOneSidedPositionPlan` function. The creation of a single-sided position is skipped in two cases:

- The maximum liquidity allowed per tick limit is exceeded (1, 2)
- The initial tick is too close to the upper or lower boundary (1, 2)

In both the cases, tokens that should have been used for the one-sided position remain in the `PositionManager` since all tokens have already been sent to it. After migration, anyone can extract these leftover tokens from the `PositionManager` contract by performing `SETTLE` and `TAKE` actions through Uniswap.

Some ways to address this issue are presented below:

- Only send the required tokens to the `PositionManager` contract. If the one-sided position is not created, the unused tokens will remain in the strategy contract. Pair this with a withdrawal function, callable only by the distribution owner, to recover the unused tokens after pool creation.
- If all tokens are transferred to the `PositionManager` contract at the start of `migrate()`, adjust the Uniswap V4 `actions` such that if a one-sided position was expected but not created, the sequence includes `SETTLE` and `TAKE` for the relevant token in the end in favor of the distribution owner.

Consider implementing one of the approaches outlined above to prevent token loss.

Update: *Resolved in pull request #43.*

Low Severity

L-01 Irrecoverable ETH

In the `Permit2Forwarder` and `Multicall` contracts, it is impossible to withdraw ETH, and the funds sent to them would be irrecoverable.

Consider removing the `payable` keyword from the `permit` and `multicall` functions of the `Permit2Forwarder` and `Multicall` contracts, respectively.

Update: *Resolved in pull request #39.*

L-02 Missing Zero-Address Checks

When operations with address parameters are performed, it is crucial to ensure the address is not mistakenly set to zero.

Throughout the codebase, multiple instances of missing zero-address checks were identified:

- The `recipient` operation within the `TokenLauncher` contract in `TokenLauncher.sol`
- The `_token` operation within the `LBPStrategyBasic` contract in `LBPStrategyBasic.sol`
- The `token` operation within the `MerkleClaimFactory` contract in `MerkleFactory.sol`

Consider always performing a zero-address check before assigning any state variable.

Update: *Resolved in pull request #40.*

L-03 `getLBPAddress` Function is Missing Parameter

The `initializeDistribution` function in `LBPStrategyBasicFactory.sol` derives the final `CREATE2` salt as `keccak256(abi.encode(msg.sender, salt))`, but `getLBPAddress` uses the raw salt directly. This inconsistency means that `getLBPAddress` will return an incorrect address unless the caller manually pre-hashes the sender with the salt off-chain. This is error-prone and contradicts the on-chain derivation logic.

Consider taking the sender as an input parameter to create the final `CREATE2` hash.

Update: *Resolved in pull request #27.*

L-04 Missing `view` function for Deterministic Address Pre-Computation in `MerkleClaimFactory`

The `MerkleClaimFactory` deploys `MerkleClaim` instances using `CREATE2` but does not expose a `view` function to pre-compute the resulting address with the same constructor arguments and salt used at deployment. Without this helper, integrators must reimplement the `CREATE2` address derivation off-chain, increasing the risk of incorrect calculations and operational mistakes.

Consider adding a `view` function in `MerkleClaimFactory` similar to `getLBPAddress` in `LBPStrategyBasicFactory` that returns the pre-computed deterministic address.

Update: *Resolved in pull request #41.*

L-05 Missing `onTokensReceived` Implementation in `MerkleClaim`

When `distributeToken` is called from `TokenLauncher.sol`, tokens are transferred to the strategy contract and the strategy contract's `onTokensReceived` function is called. This function checks if the expected amount of token is received by the strategy in `LBPStrategyBasic.sol`. However, this function is empty in `MerkleClaim.sol`.

Consider implementing the same check in `MerkleClaim.sol`.

Update: *Acknowledged, not resolved.*

L-06 Dust Is Foregone Instead of Being Swept

During pool creation, any extra tokens that are not used for seeding the pool are completely foregone (1, 2, 3) whereas they can instead be returned to the `positionRecipient`. While returning the tokens may incur more gas cost than the worth of the tokens, this conservative approach will shield in cases where the amount of dust tokens left after pool seeding is not insignificant.

Consider changing the approach and sweeping the dust to the `positionRecipient`.

Update: *Acknowledged, not resolved in pull request #52. The team stated:*

> *this is by design to save on gas since dust will be very minimal - made it clear in documentation*

L-07 Extra Tokens Sent to `LBPStrategyBasic` Upon Initialization Get Stuck

The `LBPStrategyBasic` contract is deployed with a `totalSupply` variable at construction time, but the funds are transferred to it in a later call. `onTokensReceived` is called after transferring funds to the contract, and it validates that the received funds are greater than or equal to `totalSupply`.

If extra tokens are sent to the `LBPStrategyBasic` contract, then the check will still pass, but the extra funds will not be used and will eventually get stuck. Consider adding a withdrawal function to the contract that allows the creator to retrieve any leftover funds after a successful pool migration.

Update: *Resolved in pull request #51. The team stated:*

> *fixed by allowing an operator address to withdraw tokens after a certain time period*

L-08 `onTokensReceived` Called Out of Preferred Execution Flow Can Lead to Loss of Funds

The `distributeToken` function deploys the `LBPStrategyBasic` contract, transfers funds to it, and calls its `onTokensReceived` function to trigger the deployment of the `Auction` contract. If the deployment of the `Auction` contract fails, then the entire execution chain reverts, including the fund transfer. However, the `LBPStrategyBasic` contract can be deployed directly without going through the `TokenLauncher` by calling `initializeDistribution` on the `LBPStrategyBasicFactory` contract.

If deployed this way, the `onTokensReceived` function may not necessarily be called in the same transaction as the funds transfer. This can lead to funds being stuck in the contract as the `onTokensReceived` function can revert due to failed deployment of the `Auction` contract, and there is no way to get out the funds transferred in the previous transaction.

Consider either using a pull method for the fund transfer or documenting this risk in the contract.

Update: *Resolved in pull request #51. The team stated:*

> *if tokens get stuck, an operator address can withdraw by calling the sweepTokens function*

L-09 `totalSupply` Type Mismatch Can Invalidate Deterministic Address Computation

`initializeDistribution` passes `totalSupply` to the `LBPStrategyBasic` constructor as a `uint128`, while `getLBPAddress` encodes the same argument as a `uint256`. If a caller provides a value that exceeds `2**128 - 1`, the helper will still encode the full 256-bit number when computing the `initCodeHash`, whereas the deployment path will revert (or, if ever truncated elsewhere, encode only the lower 128 bits). Either outcome makes the address predicted by `getLBPAddress` diverge from the one that would actually be deployed, defeating the purpose of deterministic address prediction and potentially breaking the integrations that rely on it.

Consider changing `uint256` to `uint128` for `totalSupply` in the `getLBPAddress` function signature.

Update: *Resolved in pull request #47.*

Notes & Additional Information

N-01 Lack of Indexed Event Parameter

Within `IDistributionStrategy.sol`, the `DistributionInitialized` event does not have indexed parameters.

To improve the ability of off-chain services to search and filter for specific events, consider indexing event parameters.

Update: *Resolved in pull request #31.*

N-02 File and Contract Names Mismatch

The `MerkleFactory.sol` file name does not match the `MerkleClaimFactory` contract name.

To make the codebase easier to understand for developers and reviewers, consider renaming the files to match the contract names.

Update: *Resolved in pull request #30.*

N-03 Missing Docstrings

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

- In `LBPStrategyBasic.sol`, the `MAX_TOKEN_SPLIT_TO_AUCTION` state variable
- In `LBPStrategyBasic.sol`, the `Q192` state variable
- In `LBPStrategyBasic.sol`, the `token` state variable
- In `LBPStrategyBasic.sol`, the `currency` state variable
- In `LBPStrategyBasic.sol`, the `poolLPFee` state variable
- In `LBPStrategyBasic.sol`, the `poolTickSpacing` state variable
- In `LBPStrategyBasic.sol`, the `totalSupply` state variable
- In `LBPStrategyBasic.sol`, the `reserveSupply` state variable
- In `LBPStrategyBasic.sol`, the `positionRecipient` state variable
- In `LBPStrategyBasic.sol`, the `migrationBlock` state variable
- In `LBPStrategyBasic.sol`, the `auctionFactory` state variable
- In `LBPStrategyBasic.sol`, the `positionManager` state variable
- In `LBPStrategyBasic.sol`, the `auction` state variable
- In `LBPStrategyBasic.sol`, the `initialSqrtPriceX96` state variable
- In `LBPStrategyBasic.sol`, the `initialTokenAmount` state variable
- In `LBPStrategyBasic.sol`, the `initialCurrencyAmount` state variable
- In `LBPStrategyBasic.sol`, the `auctionParameters` state variable
- In `LBPStrategyBasic.sol`, the `receive` function
- In `LBPStrategyBasicFactory.sol`, the `positionManager` state variable
- In `LBPStrategyBasicFactory.sol`, the `poolManager` state variable
- In `LBPStrategyBasicFactory.sol`, the `getLBPAddress` function
- In `MerkleFactory.sol`, the `MerkleClaimFactory` contract

- In `IMerkleClaim.sol`, the `IMerkleClaim` interface
- In `IERC20.sol`, the `balanceOf` function

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

Update: *Resolved in pull request #38.*

N-04 Incomplete Docstrings

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

- In `IDistributionStrategy.sol`, in the `DistributionInitialized` event, the `distributionContract`, `token`, and `totalSupply` parameters are not documented.
- In `IDistributionStrategy.sol`, in the `initializeDistribution` function, the `token`, `totalSupply`, and `salt` parameters are not documented.
- In `ILBPStrategyBasic.sol`, in the `Migrated` event, the `key` and `initialSqrtPriceX96` parameters are not documented.
- In `ITokenLauncher.sol`, in the `createToken` function, the `recipient` parameter is not documented.

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).

Update: *Resolved in pull request #32.*

N-05 Multiple Optimizable State Reads

In `LBPStrategyBasic.sol`, multiple instances of optimizable storage reads were identified:

- The `auction` storage reads (1, 2) in `onTokensReceived` in `LBPStrategyBasic.sol`
- The `auction` storage reads (1, 2, 3, 4) in `validate` in `LBPStrategyBasic.sol`
- The `initialSqrtPriceX96` storage reads (1, 2, 3, 4, 5) in `migrate` in `LBPStrategyBasic.sol`

Consider reducing SLOAD operations that consume unnecessary amounts of gas by caching the values in a memory variable.

Update: *Resolved in pull request #33.*

N-06 Missing Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts not having a security contact were identified:

- The `TokenLauncher` contract
- The `LBPStrategyBasic` contract
- The `MerkleClaim` contract
- The `LBPStrategyBasicFactory` contract
- The `MerkleClaimFactory` contract

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

Update: *Resolved in pull request #34.*

N-07 Functions Updating State Without Event Emissions

Throughout the codebase, multiple instances of functions updating the state without an event emission were identified:

- The `onTokensReceived` function in `LBPStrategyBasic.sol`
- The `validate` function in `LBPStrategyBasic.sol`

- The `validate` function in `LBPStrategyBasic.sol`
- The `validate` function in `LBPStrategyBasic.sol`

Consider emitting events whenever there are state changes to improve the clarity of the codebase and make it less error-prone.

Update: *Resolved in pull request #35.*

N-08 Unused Import

The `IDistributionStrategy` import in `LBPStrategyBasic.sol` is unused and can be removed.

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

Update: *Resolved in pull request #36. The team stated:*

> *Fixed (also rearranged ordering of imports).*

N-09 Misleading Comments

Throughout the codebase, multiple instances of misleading or imprecise comments were identified:

- Line 139 of `LBPStrategyBasic.sol` reads `will revert if cannot fit in uint128`, which is incorrect: the typecasting operation will not revert if the value is bigger than `uint128`.
- Line 31 of `TokenLauncher.sol` reads `Create token, with this contract as the recipient of the initial supply`, whereas, any address passed by the user can be the `recipient`.

Consider addressing the aforementioned instances of misleading comments.

Update: *Resolved in pull request #37.*

Conclusion

The Token Launcher system has been designed to create and launch tokens with various pluggable strategies. The Merkle distribution and liquidity bootstrap pool (LBP) strategies were part of the review. However, the LBP strategy relies on an auction system that was not in scope. Multiple critical-, high-, and medium-severity issues were identified concerning integrations with the auction contracts and Uniswap v4. In addition, low-severity issues and code-improvement recommendations were also reported.

The codebase would heavily benefit from a comprehensive integration and invariant testing suite to verify that nothing breaks due to a valid state change in one of the integrations. This would prevent edge-case issues from sneaking into the on-chain deployment.

The Uniswap Labs team is appreciated for being very helpful during the course of the review and for timely responding to all the questions posed by the audit team.