Uniswap Token Launcher

Security Review

Cantina Managed review by:
Phaze, Lead Security Researcher
0xWeiss, Security Researcher

November 12, 2025

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

1.3 Risk assessment

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

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

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

From Oct 21st to Oct 26th the Cantina team conducted a review of token-launcher on commit hash 03ca1b3e. The team identified a total of **6** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	1	1	0
Low Risk	1	0	1
Gas Optimizations	1	1	0
Informational	1	0	1
Total	**6**	**4**	**2**

2.1 Scope

The security review had the following components in scope for token-launcher on commit hash 03ca1b3e:

```
src
├── distributionContracts
│   ├── LBPStrategyBasic.sol
│   └── MerkleClaim.sol
├── distributionStrategies
│   ├── LBPStrategyBasicFactory.sol
│   └── MerkleClaimFactory.sol
├── libraries
│   ├── ActionsBuilder.sol
│   ├── ParamsBuilder.sol
│   ├── StrategyPlanner.sol
│   ├── TickCalculations.sol
│   └── TokenPricing.sol
├── Multicall.sol
├── Permit2Forwarder.sol
├── token-factories
│   └── uerc20-factory
│       ├── factories
│       │   ├── UERC20Factory.sol
│       │   └── USUPERC20Factory.sol
│       ├── libraries
│       │   └── UERC20MetadataLibrary.sol
│       └── tokens
│           ├── BaseUERC20.sol
│           ├── UERC20.sol
│           └── USUPERC20.sol
├── TokenLauncher.sol
├── types
│   ├── Distribution.sol
│   ├── MigrationData.sol
│   ├── MigratorParams.sol
│   └── PositionTypes.sol
└── utils
    └── HookBasic.sol
```

3 Findings

3.1 High Risk

3.1.1 LBPStrategyBasic allows wrong-currency auctions, causing migration DoS

Severity: High Risk

Context: LBPStrategyBasic.sol#L220-L235

Summary: The LBPStrategyBasic contract fails to validate that auction parameters specify the same currency as the strategy's configured currency. This allows auctions to be created with mismatched currencies, which raises funds in the wrong asset and permanently blocks migration due to balance validation failures. The vulnerability presents a realistic vector for both accidental misconfiguration and intentional abuse that breaks core launch functionality.

Description: The strategy's `_validateAuctionParams()` function only performs limited validation on auction parameters, checking that the funds recipient matches the placeholder value and that the end block occurs before migration. However, it omits an important validation: ensuring the auction currency matches the strategy's configured currency.

When `onTokensReceived()` is called, the strategy forwards the auction parameters directly to the auction factory without additional validation. The factory creates an auction using whatever currency is encoded in those parameters, regardless of whether it matches the strategy's expectations.

This mismatch becomes problematic during migration. The `migrate()` function retrieves the amount raised from the auction (denominated in the auction's actual currency) but then checks the strategy's balance of its configured currency against this amount. When these currencies differ, the balance check fails and migration permanently reverts, blocking pool initialization and stranding funds until the sweep period.

Impact Explanation: The vulnerability causes permanent migration failure when auction and strategy currencies differ. The strategy cannot initialize the intended liquidity pool because currency balance validation fails during migration. This breaks core launch functionality, strands raised funds in the wrong asset until the sweep period, and undermines the auction's purpose for both bidders and issuers. The issue also enables potential abuse where malicious issuers can intentionally block migration and later sweep assets.

Likelihood Explanation: The likelihood is high because issuers have full control over auction parameters and can specify any currency without constraint. The contract provides no safeguards against currency mismatches, making both accidental misconfiguration and intentional abuse readily achievable in permissionless deployments. The vulnerability can be triggered without breaking external assumptions or requiring sophisticated attack techniques.

Recommendation: Consider adding currency validation to the `_validateAuctionParams()` function to ensure consistency between auction parameters and strategy configuration. One approach could be to decode the auction parameters and verify that the specified currency matches the strategy's configured currency before proceeding with auction creation.

The validation could be implemented by adding a currency comparison check alongside the existing validations:

```diff
  function _validateAuctionParams(bytes memory auctionParams, MigratorParameters memory
  ↪   migratorParams)
      private
-     pure
+     view
  {
      AuctionParameters memory _auctionParams = abi.decode(auctionParams,
       ↪  (AuctionParameters));
      if (_auctionParams.fundsRecipient != ActionConstants.MSG_SENDER) {
          revert InvalidFundsRecipient(_auctionParams.fundsRecipient,
           ↪  ActionConstants.MSG_SENDER);
      } else if (_auctionParams.endBlock >= migratorParams.migrationBlock) {
          revert InvalidEndBlock(_auctionParams.endBlock, migratorParams.migrationBlock);
+     } else if (_auctionParams.currency != currency) {
+         revert InvalidCurrency(_auctionParams.currency, currency);
```

```
        }
    }
```

This enhancement would prevent currency mismatches at the validation stage, ensuring that auctions operate with the expected currency and maintaining the integrity of the migration process.

Proof of Concept: `test/poc/LBPStrategyBasic.CurrencyMismatch.local.poc.t.sol`:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "forge-std/Test.sol";

import {LBPStrategyBasic} from "../../src/distributionContracts/LBPStrategyBasic.sol";
import {ILBPStrategyBasic} from "../../src/interfaces/ILBPStrategyBasic.sol";
import {MigratorParameters} from "../../src/types/MigratorParams.sol";
import {LBPStrategyBasicNoValidation} from "../mocks/LBPStrategyBasicNoValidation.sol";

import {AuctionFactory} from "twap-auction/src/AuctionFactory.sol";
import {IAuction} from "twap-auction/src/interfaces/IAuction.sol";
import {AuctionParameters} from "twap-auction/src/interfaces/IAuction.sol";
import {IAllowanceTransfer} from "permit2/src/interfaces/IAllowanceTransfer.sol";
import {DeployPermit2} from "permit2/test/utils/DeployPermit2.sol";

import {IPositionManager} from
↪   "@uniswap/v4-periphery/src/interfaces/IPositionManager.sol";
import {IPoolManager} from "@uniswap/v4-core/src/interfaces/IPoolManager.sol";
import {ActionConstants} from "@uniswap/v4-periphery/src/libraries/ActionConstants.sol";
import {FixedPoint96} from "@uniswap/v4-core/src/libraries/FixedPoint96.sol";

import {MockERC20} from "../mocks/MockERC20.sol";

// Proof-of-Concept: Missing currency consistency check between LBPStrategyBasic.currency
↪   (migratorParams)
// and auctionParameters.currency allows deploying an auction in a different asset,
↪   leading to migration DoS.
contract LBPStrategyBasicCurrencyMismatchLocalPoC is Test, DeployPermit2 {
    // Test actors
    address internal issuer = address(this);
    address internal positionRecipient = address(0xBEEF);
    address internal operator = address(0xCAFE);

    // Core contracts
    LBPStrategyBasic lbp;
    AuctionFactory auctionFactory;
    IAllowanceTransfer permit2;

    // Tokens
    MockERC20 token;        // project token being distributed
    MockERC20 daiLike;      // ERC20 used as the wrong auction currency

    // Constants
    uint128 constant TOTAL_SUPPLY = 1_000e18;
    uint24 constant TOKEN_SPLIT = 5e6; // 50% to auction

    function setUp() public {
        // Deploy Permit2 singleton for Auction to use
        permit2 = IAllowanceTransfer(deployPermit2());

        // Deploy currencies
        token = new MockERC20("ProjectToken", "PRJ", TOTAL_SUPPLY, address(this));
        daiLike = new MockERC20("Mock DAI", "DAI", 1_000_000e18, address(this));

        // Deploy real AuctionFactory from dependency
        auctionFactory = new AuctionFactory();
        // Build migrator params with currency = native ETH (address(0))
        MigratorParameters memory mp = MigratorParameters({
            migrationBlock: uint64(block.number + 500),
            currency: address(0), // EXPECTED currency by strategy (ETH)
            poolLPFee: 3000,
            poolTickSpacing: 10,
            tokenSplitToAuction: TOKEN_SPLIT,
            auctionFactory: address(auctionFactory),
            positionRecipient: positionRecipient,
            sweepBlock: uint64(block.number + 1000),
            operator: operator,
            createOneSidedTokenPosition: false,
            createOneSidedCurrencyPosition: false
        });

        // Encode auction parameters with a DIFFERENT currency (DAI-like ERC20)
        // Steps must sum to ConstantsLib.MPS across their block deltas. Use a single
        ↪  step covering 100 blocks with mps = 1e5
        // so total mps*blocks = 1e7.
        bytes memory steps = abi.encodePacked(uint24(100_000), uint40(100));
        AuctionParameters memory ap = AuctionParameters({
            currency: address(daiLike), // WRONG currency vs mp.currency
            tokensRecipient: address(0xA11CE),
            fundsRecipient: ActionConstants.MSG_SENDER, // rewritten to LBP by factory
            startBlock: uint64(block.number),
            endBlock: uint64(block.number + 100),
            claimBlock: uint64(block.number + 120),
            tickSpacing: 20 << FixedPoint96.RESOLUTION,
            validationHook: address(0),
            floorPrice: 100 << FixedPoint96.RESOLUTION,
            requiredCurrencyRaised: 0, // graduate unconditionally to allow sweeping
            auctionStepsData: steps
        });

        // Deploy strategy using test hook that skips hook address validation
        lbp = new LBPStrategyBasicNoValidation(
            address(token),
            TOTAL_SUPPLY,
            mp,
            abi.encode(ap),
            IPositionManager(address(0x1111)), // unused in this PoC (revert happens
            ↪  before use)
            IPoolManager(address(0x2222)) // unused in this PoC
        );

        // Transfer full token supply to the strategy to trigger onTokensReceived
        token.transfer(address(lbp), TOTAL_SUPPLY);
        lbp.onTokensReceived();
    }

    function test_poc_wrongCurrencyBlocksMigration() public {
        IAuction _auction = lbp.auction();

        // Sanity checks: auction currency is DAI-like while strategy expects ETH
        // Verify the auction currency by comparing the balance changes route and by
        ↪  expecting sweep to transfer DAI-like
        // We cannot cast Currency to address directly, so rely on functional behavior
        ↪  checks below.
        assertEq(lbp.currency(), address(0), "strategy currency should be ETH");

        // Approve Permit2 allowance for the Auction to pull DAI from this test contract
        permit2.approve(address(daiLike), address(_auction), type(uint160).max, 0);
        // Approve ERC20 allowance for Permit2 to spend on behalf of this test contract
        daiLike.approve(address(permit2), type(uint256).max);

        // Place a single bid to raise some DAI-like currency
        // Use a small, valid price in X96 terms and non-zero amount
        uint256 priceX96 = 120 << FixedPoint96.RESOLUTION; // above floor and aligned
         ↪  with tick spacing (20)
        uint128 amount = 10e18; // 10 DAI

        uint256 prevTick = _auction.floorPrice();
        _auction.submitBid(priceX96, amount, address(this), prevTick, bytes(""));

        // Move past the end block and sweep currency to the LBP (fundsRecipient was
         ↪  rewritten to LBP)
        vm.roll(_auction.endBlock() + 1);
        _auction.sweepCurrency();

        // Strategy now holds DAI-like (wrong asset) and zero ETH
        assertGt(daiLike.balanceOf(address(lbp)), 0, "strategy should hold DAI-like");
        assertEq(address(lbp).balance, 0, "strategy should have zero ETH");

        // Attempt to migrate after migration block; should revert due to insufficient
         ↪  ETH
        vm.roll(lbp.migrationBlock() + 1);

        uint256 needed = _auction.currencyRaised();
        uint256 available = address(lbp).balance;

        vm.expectRevert(abi.encodeWithSelector(ILBPStrategyBasic.InsufficientCurrency.se⌋
         ↪  lector, needed, available));
        lbp.migrate();
    }
}
```

Command:

```
forge test --match-path test/poc/LBPStrategyBasic.CurrencyMismatch.local.poc.t.sol -vv
```

Uniswap Labs: Fixed in PR 62.

Cantina Managed: Fix verified.

3.1.2 Migration in LBP strategy can revert due to overflow and liquidity caps

Severity: High Risk

Context: LBPStrategyBasic.sol#L448

Summary: Multiple paths in `migration()` can revert under parameter sets that are not obviously pathological. Overflows in price/liquidity math and exceeding Uniswap's per-tick liquidity cap lead to reverts in the migration path. In a live flow, an attacker or misconfigured deployment can intentionally cause persistent migration failure and later sweep funds after `sweepBlock`, risking loss of auction proceeds.

Description: Observed behaviors indicate that `LBPStrategyBasic.migrate()` can fail at several points due to:

- Liquidity exceeding Uniswap v4's `maxLiquidityPerTick`, causing an `InvalidLiquidity(max, liquidity)` revert at

  `src/distributionContracts/LBPStrategyBasic.sol:298`.
- Overflow/unsafe casts in price and liquidity calculations (e.g., `SafeCastOverflow`) when deriving clearing price, amounts, and liquidity for positions.
- Reverts during `positionManager.modifyLiquidities{value: currencyTransferAmount}(...)` at `src/distributionContracts/LBPStrategyBasic.sol:378` when computed parameters are not feasible.
- Failures within `StrategyPlanner.planOneSidedPosition()` and downstream `LiquidityAmounts.getLiquidityForAmounts` at or before `src/distributionContracts/LBPStrategyBasic.sol:448`.

The fuzz harness demonstrates reverts with:

- Very large minted token supplies,
- Clearing prices within allowed bounds yet still causing infeasible liquidity.

Given the product flow (auction $\rightarrow$ migrate $\rightarrow$ potential sweep), persistent `migration()` reverts enable a scenario where a malicious token creator configures parameters to force failure and, after `sweepBlock`, sweep auction proceeds.

Impact Explanation: Migration can be permanently blocked for some parameter sets, preventing liquidity creation and finalization. If sweep mechanics allow custody of funds without completing migration, proceeds may be swept instead of distributed, causing loss of user funds and breaking the intended core flow.

Likelihood Explanation: Triggering the failure is technically straightforward with adversarial or careless parameterization (e.g., extreme spacing, large supply, certain splits). The fuzzing results show failures within bounded, realistic ranges; an attacker controlling parameters can reliably hit such conditions.

Proof of Concept: Note: This proof of concept uses clamped values to demonstrate possible bounds on `totalSupply` and `clearingPrice`. If the upper bounds are removed/loosened, the test will fail.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import {LBPStrategyBasicTestBase} from "./base/LBPStrategyBasicTestBase.sol";
import {LBPStrategyBasicNoValidation} from "../mocks/LBPStrategyBasicNoValidation.sol";
import {TokenLauncher} from "../../src/TokenLauncher.sol";
import {UERC20Factory} from
↪  "../../src/token-factories/uerc20-factory/factories/UERC20Factory.sol";
import {UERC20Metadata} from
↪  "../../src/token-factories/uerc20-factory/libraries/UERC20MetadataLibrary.sol";
import {AuctionParameters} from "twap-auction/src/interfaces/IAuction.sol";
import {AuctionStepsBuilder} from "twap-auction/test/utils/AuctionStepsBuilder.sol";
import {IAuction} from "twap-auction/src/interfaces/IAuction.sol";
import {ICheckpointStorage} from "twap-auction/src/interfaces/ICheckpointStorage.sol";
import {IERC20} from "@openzeppelin-latest/contracts/token/ERC20/IERC20.sol";
import {Hooks} from "@uniswap/v4-core/src/libraries/Hooks.sol";
import {TickMath} from "@uniswap/v4-core/src/libraries/TickMath.sol";
import {FixedPoint96} from "@uniswap/v4-core/src/libraries/FixedPoint96.sol";
import {LBPStrategyBasic} from "../../src/distributionContracts/LBPStrategyBasic.sol";
import {AuctionFactory} from "twap-auction/src/AuctionFactory.sol";
import {ConstantsLib} from "twap-auction/src/libraries/ConstantsLib.sol";
import {IAllowanceTransfer} from "permit2/src/interfaces/IAllowanceTransfer.sol";
import {IPositionManager} from
↪  "@uniswap/v4-periphery/src/interfaces/IPositionManager.sol";
import {IPoolManager} from "@uniswap/v4-core/src/interfaces/IPoolManager.sol";
import {LPFeeLibrary} from "@uniswap/v4-core/src/libraries/LPFeeLibrary.sol";
import {MigratorParameters} from "../../src/distributionContracts/LBPStrategyBasic.sol";
import {ActionConstants} from "@uniswap/v4-periphery/src/libraries/ActionConstants.sol";
import {FullMath} from "@uniswap/v4-core/src/libraries/FullMath.sol";
import "forge-std/console.sol";

contract LBPStrategyBasicFuzzTest is LBPStrategyBasicTestBase {
    using AuctionStepsBuilder for bytes;

    uint24 constant MAX_TOKEN_SPLIT_CONST = 1e7;
    uint24 constant DEFAULT_MPS = 100_000;
    uint40 constant DEFAULT_BLOCK_DELTA = 50;

    int24 spacing;

    uint256 constant localTickSpacing = 1 << FixedPoint96.RESOLUTION;
    uint256 constant localFloorPrice = localTickSpacing;
    uint256 MAX_PRICE = FullMath.mulDiv(TickMath.MAX_SQRT_PRICE,
     ↪  TickMath.MAX_SQRT_PRICE, FixedPoint96.Q96);

    /// @dev Override setup — base contract deploys default fixtures we do not need.
```

```
function setUp() public override {}

uint256 constant K = 20;
uint256 constant MAX_AUCTION_SUPPLY = 2 ** (108 - K * 3 / 2 + 2);
uint256 constant MAX_CURRENCY_RAISED = 2 ** (108 - K / 2);
uint256 constant MIN_AUCTION_PRICE = 2 ** (96 - K) - 1;
uint256 constant MAX_AUCTION_PRICE = 2 ** (96 + K) - 1;

function test_fuzz_migrate_full_create(uint128 totalSupply, uint24 tokenSplit,
 ↪  uint256 clearingPriceQ96) public {
    vm.createSelectFork(vm.envString("FORK_URL"), FORK_BLOCK);

    auctionFactory = new AuctionFactory();
    tokenLauncher = new TokenLauncher(IAllowanceTransfer(PERMIT2));
    UERC20Factory factory = new UERC20Factory();

    // Minimum totalSupply to prevent 0 auctionSupply and liquidity
    totalSupply = uint128(bound(totalSupply, 1e18, MAX_AUCTION_SUPPLY));
    tokenSplit = uint24(bound(tokenSplit, 1, MAX_TOKEN_SPLIT_CONST - 1));
    clearingPriceQ96 = uint160(bound(clearingPriceQ96, MIN_AUCTION_PRICE,
     ↪  MAX_AUCTION_PRICE));

    console.log("----------------------------------");
    console.log("totalSupply %s e18", uint256(totalSupply) / 1e18);
    console.log("tokenSplit %s%", uint256(tokenSplit) * 100 / MAX_TOKEN_SPLIT_CONST);

    UERC20Metadata memory metadata;

    address token = tokenLauncher.createToken(
        address(factory), "FuzzToken", "FZT", 18, totalSupply,
         ↪  address(tokenLauncher), abi.encode(metadata)
    );

    bytes memory auctionStepsData = AuctionStepsBuilder.init().addStep(DEFAULT_MPS,
     ↪  DEFAULT_BLOCK_DELTA * 2);

    AuctionParameters memory auctionParams = AuctionParameters({
        currency: address(0),
        tokensRecipient: makeAddr("tokensRecipient"),
        fundsRecipient: ActionConstants.MSG_SENDER,
        startBlock: uint64(block.number),
        endBlock: uint64(block.number + 100),
        claimBlock: uint64(block.number + 200),
        tickSpacing: 1,
        validationHook: address(0),
        floorPrice: localFloorPrice,
        requiredCurrencyRaised: 0,
        auctionStepsData: auctionStepsData
    });

    migratorParams = MigratorParameters({
        currency: address(0),
        poolLPFee: 1,
        poolTickSpacing: 1,
        tokenSplitToAuction: tokenSplit,
        auctionFactory: address(auctionFactory),
        positionRecipient: address(3),
        migrationBlock: uint64(block.number + 300),
        sweepBlock: uint64(block.number + 400),
        operator: address(this),
        createOneSidedTokenPosition: true,
        createOneSidedCurrencyPosition: true
    });

    _deployLBPStrategyWithToken(token, totalSupply, abi.encode(auctionParams),
     ↪  spacing);
```

```
        // Seed the auction by transferring the freshly minted supply from the launcher
        sendTokensToLBP(address(tokenLauncher), IERC20(token), lbp, totalSupply);

        IAuction auction = lbp.auction();
        vm.roll(auctionParams.startBlock);

        uint128 auctionSupply = auction.totalSupply();
        uint256 currencyRaised = FullMath.mulDivRoundingUp(auctionSupply,
        ↪  clearingPriceQ96, FixedPoint96.Q96);

        console.log("----------------------------------");
        console.log("auctionSupply %18e ether", auctionSupply);
        console.log("currencyRaised %18e ether", currencyRaised);
        console.log("clearingPrice %8e Q96", clearingPriceQ96 * 1e8 / FixedPoint96.Q96);

        // require(currencyRaised < MAX_CURRENCY_RAISED, "currencyRaised: overflow");

        vm.deal(address(lbp), currencyRaised);

        vm.mockCall(
            address(auction),
            abi.encodeWithSelector(ICheckpointStorage.currencyRaised.selector),
            abi.encode(uint256(currencyRaised))
        );
        vm.mockCall(
            address(auction),
            abi.encodeWithSelector(ICheckpointStorage.clearingPrice.selector),
            abi.encode(clearingPriceQ96)
        );

        vm.roll(migratorParams.migrationBlock);
        lbp.migrate();

        IPositionManager positionManager = IPositionManager(POSITION_MANAGER);
        uint256 mintedId = positionManager.nextTokenId() - 1;
        vm.assertGt(positionManager.getPositionLiquidity(mintedId), 0, "liquidity should
        ↪  be added");
    }

    function _deployLBPStrategyWithToken(address tokenAddr, uint128 totalSupply, bytes
    ↪  memory params, int24) internal {
        address hookAddress = address(
            uint160(uint256(type(uint160).max) & CLEAR_ALL_HOOK_PERMISSIONS_MASK |
            ↪  Hooks.BEFORE_INITIALIZE_FLAG)
        );
        lbp = LBPStrategyBasic(payable(hookAddress));

        LBPStrategyBasicNoValidation lbpNoValidation = new LBPStrategyBasicNoValidation(
            tokenAddr,
            totalSupply,
            migratorParams,
            params,
            IPositionManager(POSITION_MANAGER),
            IPoolManager(POOL_MANAGER)
        );

        vm.etch(address(lbp), address(lbpNoValidation).code);
        LBPStrategyBasicNoValidation(payable(address(lbp))).setAuctionParameters(params);
    }
}
```

Recommendation: Consider implementing parameter validation with calculated bounds to ensure migration safety. Analysis shows that with appropriate bounds, the migration process can be made reliable:

1. Clamp and validate values to avoid protocol limits.

Implement parameter bounds inside the auction contract based on the Uniswap v4 price model where `p = (sqrtPriceX96 / 2^96)^2`. Given the price band $p \in [p_{\text{min}}, p_{\text{max}}] = [2^{-K}, 2^{K}]$, the general constraint is:

```
amount_0 ≤ 2^(106-K/2)
amount_1 ≤ 2^(106-K/2)
```

If we limit $\text{amount}_0 \leq 2^{106-3K/2}$ even further, it follows that $\text{amount}_1 \leq p_{\text{max}} \cdot \text{amount}_0 = 2^{106-3K/2+K} = 2^{106-K/2}$ will always hold.

Example:

- Price bound: `price < 2^20`.
- Auction supply bound: `auctionSupply < 2^76`.
- Currency raised bound: `currencyRaised ≈ p * auctionSupply < 2^20 * 2^76 = 2^96`.

The complete derivation is given below in section "Auction Liquidity Price Bounds".

Important note on enforcement location: While it's possible to enforce auction supply limits by restricting the total supply of tokens created within the token launcher, tokens created outside of this system could have higher total supply limits. Therefore, the auction contract itself must contain these safety limits to ensure proper bounds checking regardless of the token's origin. Relying solely on token creation constraints would leave the system vulnerable to tokens minted through alternative mechanisms.

2. Atomic pull and migrate (additional safety).

   Consider atomically pulling funds from the auction and performing migration in a single transaction guarded by an `onlyStrategy`-style modifier. If migration fails, revert the whole transaction so funds remain in the auction and eligible for refunds:

```
function pullAndMigrate() external onlyStrategy nonReentrant {
    // 1) Pull proceeds from auction to this contract
    auction.sweepCurrency(); // reverts if not graduated or insufficient

    // 2) Execute migration; revert on any failure so auction can refund
    _migrateInternal(); // builds plan, checks bounds, adds liquidity atomically
}
```

3. Migrate try/catch with strategy-side failover (fallback option).

   Consider wrapping migration in a self-call `try/catch` to detect failures, record a terminal "failed" state, and expose a controlled refund path:

```
function migrate() external nonReentrant {
    if (msg.sender != address(this)) {
        try this.migrate() {
            _markMigrationSuccess();
        } catch {
            _markMigrationFailed(); // gate sweeping; emit signal for refunds
        }
        return;
    } else {
        _migrateInternal();
    }
}
```

Liquidity amounts:

$$L0 = \text{amount}_0 \cdot (\sqrt{P_U} \cdot \sqrt{p})/(\sqrt{P_U} - \sqrt{p})$$
$$L1 = \text{amount}_1/(\sqrt{p} - \sqrt{P_L})$$

where $\sqrt{p}$ is the token1 / token0 square-root price; and $\sqrt{P_U}$ and $\sqrt{P_L}$ are the maximum upper and lower square-root price as defined in Uniswap.

Per-tick liquidity cap:

$$2^{107} < L_{\text{max}} = \frac{2^{128} - 1}{2 \cdot T_{\text{max}} + 1} < 2^{108}$$

where $T_{\text{max}} = 887272$.

Using liquidities:

$$\begin{aligned} L_0 = \quad & \text{amount}_0 \cdot (\sqrt{P_U} \cdot \sqrt{p})/(\sqrt{P_U} - \sqrt{p}) \leq L_{\text{max}} \\ \iff \quad & \text{amount}_0 \leq L_{\text{max}} \cdot (\sqrt{P_U} - \sqrt{p})/(\sqrt{P_U} \cdot \sqrt{p}) \qquad (0) \end{aligned}$$

$$\begin{aligned} L_1 = \quad & \text{amount}_1/(\sqrt{p} - \sqrt{P_L}) \leq L_{\text{max}} \\ \iff \quad & \text{amount}_1 \leq L_{\text{max}} \cdot (\sqrt{p} - \sqrt{P_L}) \qquad (1) \end{aligned}$$

Define a price band $p \in [p_{\text{min}}, p_{\text{max}}]$, then (1) is satisfied when

$$\text{amount}_1 \leq L_{\text{max}} \cdot (\sqrt{p_{\text{min}}} - \sqrt{P_L})$$

and (0) is satisfied when

$$\text{amount}_0 \leq L_{\text{max}} \cdot (\sqrt{P_U} - \sqrt{p_{\text{max}}})/(\sqrt{P_U} \cdot \sqrt{p_{\text{max}}})$$

Numerical bounds:

$$2^{-128} < P_L < P_U < 2^{128}$$

Define the price band $p \in [p_{\text{min}}, p_{\text{max}}] = [2^{-32}, 2^{32}]$.

$$(0)$$

$$\begin{aligned} & \text{amount}_0 \leq L_{\text{max}} \cdot (\sqrt{P_U} - \sqrt{p_{\text{max}}})/(\sqrt{P_U} \cdot \sqrt{p_{\text{max}}}) \\ \iff \quad & \text{amount}_0 \leq L_{\text{max}} \cdot (p_{\text{max}}^{-\frac{1}{2}} - P_U^{-\frac{1}{2}}) \\ \Leftarrow \quad & \text{amount}_0 \leq 2^{107} \cdot (2^{-16} - 2^{-32}) \\ \Leftarrow \quad & \text{amount}_0 \leq 2^{107} \cdot 2^{-17} \cdot (2 - 2^{-15}) \\ \iff \quad & \text{amount}_0 \leq 2^{90} \end{aligned}$$

$$(1)$$

$$\begin{aligned} & \text{amount}_1 \leq L_{\text{max}} \cdot (\sqrt{p_{\text{min}}} - \sqrt{P_L}) \\ \Leftarrow \quad & \text{amount}_1 \leq 2^{107} \cdot (2^{-16} - 2^{-64}) \\ \Leftarrow \quad & \text{amount}_1 \leq 2^{107} \cdot 2^{-17}(2 - 2^{-47}) \\ \Leftarrow \quad & \text{amount}_1 \leq 2^{107} \cdot 2^{-17} \\ \Leftarrow \quad & \text{amount}_1 \leq 2^{90} \end{aligned}$$

Note: If a one-sided position is added to the mix, we need to ensure that these constraints are applied to the sum of both liquidities.

General Formula: Given the price band $p \in [p_{\text{min}}, p_{\text{max}}] = [2^{-K}, 2^K]$, we must at least limit $\text{amount}_{0,1} \leq 2^{106-K/2} = \text{amount}_{\text{max}}$. If we limit $\text{amount}_0 \leq 2^{106-3K/2}$ even further, it follows that $\text{amount}_1 \leq p_{\text{max}} \cdot \text{amount}_{\text{max}} = 2^{106-3K/2+K} = 2^{106-K/2} = \text{amount}_{\text{max}}$ will always hold.

Asymmetrical Price Bands: Theoretically, only $\min(L_0, L_1)$ must be below 2^{128}. Assume $T = \text{amount}_0$, the total token supply, and $C = \text{amount}_1 = p \cdot T$ the currency supply and $p = C/T$ is the price.

$$
\begin{aligned}
L_0(p) &= T \cdot (\sqrt{P_U} \cdot \sqrt{p})/(\sqrt{P_U} - \sqrt{p}) \\
L_1(p) &= p \cdot T/(\sqrt{p} - \sqrt{P_L})
\end{aligned}
$$

When is the token side the minimum?

$$
\begin{aligned}
& L_0 < L_1 \\
\iff\quad & T \cdot (\sqrt{P_U} \cdot \sqrt{p})/(\sqrt{P_U} - \sqrt{p}) < p \cdot T/(\sqrt{p} - \sqrt{P_L}) \\
\iff\quad & (\sqrt{p} - \sqrt{P_L})(\sqrt{P_U} \cdot \sqrt{p}) < p(\sqrt{P_U} - \sqrt{p}) \\
\iff\quad & \sqrt{P_U} \cdot p - \sqrt{P_L}\sqrt{P_U} \cdot \sqrt{p} < p\sqrt{P_U} - p\sqrt{p} \\
\iff\quad & p\sqrt{p} < \sqrt{P_L}\sqrt{P_U} \cdot \sqrt{p} \\
\iff\quad & p < \sqrt{P_L P_U} \\
\iff\quad & p < 1 \\
\iff\quad & C < T
\end{aligned}
$$

Since $P_L = P_U^{-1}$.

Therefore:

- For prices below 1 (i.e. less currency raised than token supply: $C < T$), the token-side liquidity L_0 is the bottleneck (minimum) ($L_0 < L_1$).
- For prices above 1 ($C > T$), the currency side is the minimum: $L_0 > L_1$.

Swapping the sort order (token becomes currency1) mirrors this condition. In other words, whichever asset is token0, there is always the half of the symmetric band where the unbounded side drives the min.

Uniswap Labs: Fixed in commit d490f8ae.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Zero liquidity one-sided position can cause migration revert

Severity: Medium Risk

Context: StrategyPlanner.sol#L63-L109

Description: The `planOneSidedPosition()` function in StrategyPlanner does not validate that the calculated liquidity amount is greater than zero before proceeding to build a one-sided position. When the reserve supply nearly equals the required token amounts for the base position, leftover amounts may be too small to generate meaningful liquidity, resulting in `newLiquidity == 0`.

If this occurs, the function will still attempt to create a one-sided position with zero liquidity, which will likely cause the migration transaction to revert when the position manager tries to execute the mint operation.

The function currently has checks for:

- Tick bounds validity (returns early if bounds are 0,0).
- Maximum liquidity per tick limits.
- But no validation for zero liquidity amounts.

This edge case could occur when auction proceeds or leftover token amounts are very small relative to the current price, causing the `getLiquidityForAmounts()` calculation to round down to zero.

Recommendation: Add a check to validate that `newLiquidity` is greater than zero before proceeding with the one-sided position creation:

```
  uint128 newLiquidity = LiquidityAmounts.getLiquidityForAmounts(
      baseParams.initialSqrtPriceX96,
      TickMath.getSqrtPriceAtTick(bounds.lowerTick),
      TickMath.getSqrtPriceAtTick(bounds.upperTick),
      currencyIsCurrency0 == oneSidedParams.inToken ? 0 : oneSidedParams.amount,
      currencyIsCurrency0 == oneSidedParams.inToken ? oneSidedParams.amount : 0
  );

+  if (newLiquidity == 0) {
+      return (existingActions, ParamsBuilder.truncateParams(existingParams));
+  }

  if (baseParams.liquidity + newLiquidity >
  ↪   baseParams.poolTickSpacing.tickSpacingToMaxLiquidityPerTick()) {
      return (existingActions, ParamsBuilder.truncateParams(existingParams));
  }
```

This ensures that the function falls back to only creating the full-range position when the one-sided position would contribute zero liquidity, preventing potential reverts during migration execution.

Uniswap Labs: Fixed in PR 62.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Missing validation of the factory and strategy addresses

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `createToken` function, the `factory` address is not validated to be Uniswaps factory address, therefore anyone could set a custom factory and emit the `TokenCreated` event with whatever address they want. This could affect in case any off-chain system is indexing the events coming from the `TokenLauncher` contract.

```
function createToken(
    address factory,
    string calldata name,
    string calldata symbol,
    uint8 decimals,
    uint128 initialSupply,
    address recipient,
    bytes calldata tokenData
) external override returns (address tokenAddress) {
    if (recipient == address(0)) {
        revert RecipientCannotBeZeroAddress();
    }
        tokenAddress = ITokenFactory(factory).createToken( // <<<
        name, symbol, decimals, initialSupply, recipient, tokenData,
        ↪   getGraffiti(msg.sender)
    );

    emit TokenCreated(tokenAddress);
}
```

Additionally, in the `distributeToken` function, the strategy address is not validated either, which in case the tokens would be sitting in the contract, they could be stolen by setting `payerIsUser = false` and return a custom `distributionContract` address.

```
function distributeToken(address token, Distribution calldata distribution, bool
↪   payerIsUser, bytes32 salt)
    external
    override
    returns (IDistributionContract distributionContract)
```

```
{
    // Call the strategy: it might do distributions itself or deploy a new instance.
    // If it does distributions itself, distributionContract == dist.strategy
    distributionContract =
    ↪   IDistributionStrategy(distribution.strategy).initializeDistribution( // <<<
        token, distribution.amount, distribution.configData,
        ↪   keccak256(abi.encode(msg.sender, salt))
    );

        // Now transfer the tokens to the returned address
    // payerIsUser should be false if the tokens were created in the same call via
    ↪   multicall
    _transferToken(token, _mapPayer(payerIsUser), address(distributionContract),
    ↪   distribution.amount);

    // Notify the distribution contract that it has received the tokens
    distributionContract.onTokensReceived();

    emit TokenDistributed(token, address(distributionContract), distribution.amount);
}
```

Recommendation: Add a whitelist system for both addresses and verify that they are part of Uniswaps system.

Uniswap Labs: Acknowledged. Won't fix this, we can't support a whitelisted set of factories / strategies.

Cantina Managed: Acknowledged.

3.4 Gas Optimization

3.4.1 Gas optimization improvements

Severity: Gas Optimization

Context: ParamsBuilder.sol#L112-L121

Description and Recommendations:

1. Use assembly to truncate array length instead of copying elements.

   The current `truncateParams()` function creates a new array and copies elements, which is unnecessarily expensive. The function can be optimized by using assembly to directly modify the array length in memory.

   ```
         function truncateParams(bytes[] memory params) internal pure returns (bytes[]
         ↪   memory) {
   -       bytes[] memory truncated = new bytes[](FULL_RANGE_SIZE);
   -       for (uint256 i = 0; i < FULL_RANGE_SIZE; i++) {
   -           truncated[i] = params[i];
   -       }
   -       return truncated;
   +       assembly {
   +           mstore(params, FULL_RANGE_SIZE)
   +       }
   +       return params;
         }
   ```

   This optimization eliminates the need for array allocation and element copying, reducing gas consumption by directly modifying the array's length field in memory.

Uniswap Labs: Fixed in PR 62.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 Race condition allows simultaneous claiming and withdrawing at deadline

Severity: Informational

Context: MerkleClaim.sol#L10-L23

Description: The MerkleClaim contract allows both user claiming and admin withdrawal operations to occur simultaneously at the deadline block (`block.timestamp == endTime`). This creates a race condition where administrators can potentially withdraw tokens that users are attempting to claim within the same block.

```
function claim(uint256 index, address account, uint256 amount, bytes32[] calldata
↪   merkleProof) public override {
    if (block.timestamp > endTime) revert ClaimWindowFinished();
    super.claim(index, account, amount, merkleProof);
}

function withdraw() external onlyOwner {
    if (block.timestamp < endTime) revert NoWithdrawDuringClaim();
    IERC20(token).safeTransfer(msg.sender, IERC20(token).balanceOf(address(this)));
}
```

The issue occurs because the inherited MerkleDistributorWithDeadline contract permits both claiming and withdrawal operations when the current timestamp equals the deadline. While this doesn't create a security vulnerability, it can lead to inconsistent user experiences where valid claim transactions may fail due to the owner's withdrawal.

Recommendation: Consider implementing a grace period or modifying the withdrawal timing to prevent simultaneous operations at the deadline. One approach is to use strict inequality checks in the withdrawal function:

```
require(block.timestamp > endTime, "Claiming period not yet ended");
```

This ensures users have exclusive access to claim tokens until the deadline passes, while administrators can withdraw unclaimed tokens only after the deadline has definitively expired.

Uniswap Labs: Acknowledged.

Cantina Managed: Acknowledged.