



Uniswap Liquidity Launcher Security Review

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

January 21, 2026

Contents

1	Introduction	2
1.1	About Cantina	2
1.2	Disclaimer	2
1.3	Risk assessment	2
1.3.1	Severity Classification	2
2	Security Review Summary	3
2.1	Review 1: Initial Security Review	3
2.1.1	Scope - Review 1	3
2.2	Review 2: PR 103 Security Review	4
2.2.1	Scope - Review 2	4
2.3	Combined Summary	4
3	Findings	5
4	Initial Review Findings	5
4.1	Low Risk	5
4.1.1	Lack of slippage protection in collectFees allows value extraction through front-running	5
4.2	Gas Optimization	6
4.2.1	Unnecessary ownership check in requireOwned modifier increases gas costs	6
4.3	Informational	7
4.3.1	Periphery position recipient contracts require separate deployment per strategy	7
4.3.2	Protocol requires EIP-1153 transient storage support without documentation	8
4.3.3	Missing validation for maxCurrencyAmountForLP parameter	9
4.3.4	Transient storage in DynamicArray adds complexity and gas overhead	10
4.3.5	Unused imports across the scope	12
5	PR 103 Review Findings	13
5.1	Gas Optimization	13
5.1.1	Redundant external call to fetch LBP initialization parameters	13
5.2	Informational	13
5.2.1	Check-effects-interactions pattern violation in initializer setup	13
5.2.2	Consider using timestamps instead of block numbers for time-based checks	14
5.2.3	Hardcoded interface ID may lead to regression issues	15

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).

2.1 Review 1: Initial Security Review

From Dec 28th to Jan 1st the Cantina team conducted a review of liquidity-launcher on commit hash [994f4b09](#). The team identified a total of **7** issues:

Issues Found - Review 1

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	1	0
Gas Optimizations	1	1	0
Informational	5	4	1
Total	7	6	1

2.1.1 Scope - Review 1

The security review had the following components in scope for liquidity-launcher on commit hash [994f4b09](#):

```
src
└── factories
    ├── lbp
    │   ├── AdvancedLBPStrategyFactory.sol
    │   ├── FullRangeLBPStrategyFactory.sol
    │   └── GovernedLBPStrategyFactory.sol
    ├── periphery
    │   └── MerkleClaimFactory.sol
    └── StrategyFactory.sol
└── LiquidityLauncher.sol
└── Multicall.sol
└── periphery
    ├── BuybackAndBurnPositionRecipient.sol
    ├── hooks
    │   └── SelfInitializerHook.sol
    ├── PositionFeesForwarder.sol
    └── TimelockedPositionRecipient.sol
└── Permit2Forwarder.sol
└── strategies
    ├── lbp
    │   ├── AdvancedLBPStrategy.sol
    │   ├── FullRangeLBPStrategy.sol
    │   ├── GovernedLBPStrategy.sol
    │   ├── LBPStrategyBase.sol
    │   └── VirtualGovernedLBPStrategy.sol
    └── MerkleClaim.sol
└── types
    ├── Distribution.sol
    ├── MigrationData.sol
    ├── MigratorParameters.sol
    └── PositionTypes.sol
```

2.2 Review 2: PR 103 Security Review

From Jan 11th to Jan 13th Phaze conducted a review of liquidity-launcher on commit hash 3cabf93a. Phaze identified a total of **4** issues:

Issues Found - Review 2

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	0	0	0
Gas Optimizations	1	1	0
Informational	3	2	1
Total	4	3	1

2.2.1 Scope - Review 2

The security review had the following components in scope for liquidity-launcher on commit hash 3cabf93a:

```
src
└── libraries
    ├── ParamsBuilder.sol
    ├── StrategyPlanner.sol
    ├── TokenDistribution.sol
    └── TokenPricing.sol
└── periphery
    └── TimelockedPositionRecipient.sol
└── strategies
    └── lbp
        ├── AdvancedLBPStrategy.sol
        ├── FullRangeLBPStrategy.sol
        ├── GovernedLBPStrategy.sol
        ├── LBPStrategyBase.sol
        └── VirtualGovernedLBPStrategy.sol
└── types
    ├── MigrationData.sol
    └── MigratorParameters.sol
```

2.3 Combined Summary

Across both security reviews, the Cantina team identified a total of **11** issues:

Total Issues Found Across Both Reviews

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	1	0
Gas Optimizations	2	2	0
Informational	8	6	2
Total	11	9	2

3 Findings

4 Initial Review Findings

4.1 Low Risk

4.1.1 Lack of slippage protection in `collectFees` allows value extraction through front-running

Severity: Low Risk

Context: [BuybackAndBurnPositionRecipient.sol#L55-L77](#)

Summary: The `collectFees()` function in `BuybackAndBurnPositionRecipient.sol` requires callers to burn tokens upfront but does not enforce a minimum amount of currency received in return. This creates a front-running vulnerability where an attacker can extract fees from the underlying pool before a user's transaction executes, causing the user to burn tokens while receiving little to no currency compensation.

Description: The `collectFees()` function of `BuybackAndBurnPositionRecipient.sol` implements a buyback-and-burn mechanism where users pay a fixed amount of tokens (`minTokenBurnAmount`) to claim accumulated fees from a Uniswap V4 position. The function executes in the following order:

1. Transfers `minTokenBurnAmount` of token from the caller to the burn address.
2. Collects fees from the position via `DECREASE_LIQUIDITY` with 0 liquidity.
3. Sends the token portion of fees to the burn address.
4. Sends the currency portion of fees to the caller using `OPEN_DELTA`.

The vulnerability arises because the caller commits to burning tokens before knowing how much currency they will receive. The `OPEN_DELTA` constant instructs the position manager to transfer whatever amount was collected, with no minimum threshold enforced. If an attacker monitors the mempool and front-runs the transaction, they can manipulate pool state or collect fees through their own positions, leaving minimal or zero currency for the original caller.

Consider the following scenario:

1. A position has accumulated 1000 USDC in fees.
2. Alice submits a transaction to call `collectFees()`, expecting to receive ~1000 USDC for burning her tokens.
3. Bob (who may be the token creator with aligned incentives) observes Alice's transaction.
4. Bob front-runs Alice by collecting fees from a related position or manipulating the pool.
5. Alice's transaction executes successfully, burning her `minTokenBurnAmount` tokens.
6. Alice receives 0 or minimal USDC because the fees were already extracted.

The token creator may have particular incentive to perform this attack, as it allows them to acquire currency from the pool while forcing others to burn tokens (supporting the token price) without fair compensation.

Impact: Users calling `collectFees()` may lose their burned tokens (`minTokenBurnAmount`) without receiving the expected currency compensation. While the loss is bounded by `minTokenBurnAmount`, users have no protection against receiving disproportionately low returns. This undermines the intended economic model where users voluntarily burn tokens in exchange for accumulated fees. In extreme cases, users may burn tokens and receive zero currency, representing a complete loss of the burned amount.

Likelihood: The likelihood is medium because executing this attack requires front-running capabilities and mempool monitoring, but the economic incentives make it attractive. Token creators have structural motivation to reduce circulating supply while acquiring the currency portion of fees. The attack is technically straightforward once mempool access is established, and transactions calling `collectFees()` are easily identifiable. The profitability of the attack increases with higher accumulated fees, making it more likely when the function is called during optimal conditions.

Recommendation: Consider adding a slippage protection parameter to `collectFees()` that allows callers to specify the minimum amount of currency they expect to receive. The function should revert if the collected currency falls below this threshold, preventing users from burning tokens without adequate compensation.

One approach could be:

```
- function collectFees(uint256 _tokenId) external nonReentrant requireOwned(_tokenId) {
+ function collectFees(uint256 _tokenId, uint256 minCurrencyAmount) external nonReentrant
→  requireOwned(_tokenId) {
    // Require the caller to burn at least the minimum amount of `token`
    SafeTransferLib.safeTransferFrom(token, msg.sender, BURN_ADDRESS,
→   minTokenBurnAmount);
    emit TokensBurned(minTokenBurnAmount);

    // Collect the fees from the position
    bytes memory actions =
        abi.encodePacked(uint8(Actions.DECREASE_LIQUIDITY), uint8(Actions.TAKE),
→   uint8(Actions.TAKE));
    bytes[] memory params = new bytes[](3);
    // Call DECREASE_LIQUIDITY with a liquidity of 0 to collect fees
    params[0] = abi.encode(_tokenId, 0, 0, 0, bytes(""));
    // Call TAKE to send the tokens to the burn address
    params[1] = abi.encode(token, BURN_ADDRESS, ActionConstants.OPEN_DELTA);
-   // Call TAKE to send the currency to the caller
-   params[2] = abi.encode(currency, msg.sender, ActionConstants.OPEN_DELTA);
+   // Call TAKE to send the currency to this contract temporarily
+   params[2] = abi.encode(currency, address(this), ActionConstants.OPEN_DELTA);

    // Set deadline to the current block
    positionManager.modifyLiquidity(abi.encode(actions, params), block.timestamp);

+   // Check received currency amount and transfer to caller
+   uint256 currencyReceived = Currency.wrap(currency).balanceOfSelf();
+   if (currencyReceived < minCurrencyAmount) {
+       revert InsufficientCurrencyReceived(currencyReceived, minCurrencyAmount);
+   }
+   SafeTransferLib.safeTransfer(currency, msg.sender, currencyReceived);

    emit FeesCollected(msg.sender);
}
```

This modification allows users to protect themselves against front-running by specifying their minimum acceptable return, ensuring the economic exchange remains fair even under adversarial conditions.

Uniswap Labs: Fixed in PR 107.

Cantina Managed: Fix verified.

4.2 Gas Optimization

4.2.1 Unnecessary ownership check in `requireOwned` modifier increases gas costs

Severity: Gas Optimization

Context: `TimelockedPositionRecipient.sol#L28-L33`

Description: The `requireOwned` modifier in `TimelockedPositionRecipient` performs an explicit ownership check before operations on LP positions. This check is unnecessary because the underlying position manager will revert if the contract doesn't own the position when `modifyLiquidity()` is called.

The modifier is defined in `TimelockedPositionRecipient.sol#L30-L33`:

```
modifier requireOwned(uint256 _tokenId) {
    if (IERC721(address(positionManager)).ownerOf(_tokenId) != address(this)) revert
→   NotPositionOwner();
→   _;
}
```

This modifier is used in two functions:

- `BuybackAndBurnPositionRecipient.collectFees()`.
- `'PositionFeesForwarder.collectFees()`.

Both functions call `positionManager.modifyLiquidities()` with the `_tokenId` parameter. The position manager internally validates ownership before allowing modifications to a position, so the explicit check is redundant.

Gas cost impact:

Each use of the `requireOwned` modifier adds:

- One external call to `IERC721.ownerOf()`.
- Additional keccak and sload operations.

Error message trade-off: The only benefit of the explicit check is providing a more specific error message (`NotPositionOwner()`) compared to whatever error the position manager would bubble up. However, this comes at the cost of increased gas consumption for every valid operation.

Recommendation: Consider removing the `requireOwned` modifier to reduce gas costs for users calling `collectFees()`. The position manager's internal validation will still prevent unauthorized operations, though with a less specific error message:

```
- function collectFees(uint256 _tokenId) external nonReentrant requireOwned(_tokenId) {
+ function collectFees(uint256 _tokenId) external nonReentrant {
```

This change would apply to both `BuybackAndBurnPositionRecipient` and `PositionFeesForwarder`. The trade-off is slightly less clear error messages in exchange for reduced gas costs on every successful fee collection operation. Given that fee collection is expected to be a frequent operation, optimizing for the success case may be preferable to optimizing for clearer error messages in the failure case.

Alternatively, if the improved error messaging is deemed valuable, the current implementation can be retained with the understanding that it prioritizes user experience over gas optimization.

Uniswap Labs: Fixed in PR 106.

Cantina Managed: Fix verified.

4.3 Informational

4.3.1 Periphery position recipient contracts require separate deployment per strategy

Severity: Informational

Context: `TimelockedPositionRecipient.sol#L9-L46`

Description: The current implementation of position management utilities uses standalone periphery contracts that must be deployed separately for each strategy instance. The system includes three recipient contract types:

- `TimelockedPositionRecipient`: Base contract that holds Uniswap v4 LP positions until a specified timelock period passes.
- `BuybackAndBurnPositionRecipient`: Extends the base to collect fees and burn a specified token portion.
- `PositionFeesForwarder`: Extends the base to collect and forward fees to a designated recipient.

Each recipient contract is deployed with immutable parameters tailored to specific strategy requirements (`positionManager`, `operator`, `timelockBlockNumber`, and contract-specific parameters like `token`, `currency`, `minTokenBurnAmount`, or `feeRecipient`).

Current implementation characteristics: The separate periphery approach provides clean separation between strategy logic and position management utilities. However, this architectural pattern introduces several complexities:

- Each strategy requiring position recipient functionality must deploy a dedicated periphery contract instance.

- The strategy must correctly configure and reference the appropriate LP position recipient during execution.
- Deployment orchestration becomes more complex, as the periphery contract must be deployed before or during strategy setup.
- Additional gas costs are incurred for deploying multiple contract instances across different strategies.
- An extra level of indirection exists between strategies and the position manager.

Alternative approach: Position recipient functionality could be integrated directly into the main strategy contracts, which would:

- Eliminate the need for separate periphery contract deployments.
- Reduce deployment gas costs.
- Remove the complexity of coordinating between strategy and recipient contracts.
- Simplify the deployment process by consolidating logic into a single contract per strategy.
- Remove the configuration requirement for specifying the correct LP position recipient.

However, integration would increase the complexity within strategy contracts themselves, as the position management logic (timelock mechanics, fee collection, burning/forwarding) would need to be incorporated into each strategy's implementation and state management.

Recommendation: Consider evaluating whether integrating position recipient functionality directly into strategy contracts aligns with the protocol's architectural priorities.

- If deployment efficiency and reduced coordination complexity are priorities: The position recipient functionality could be incorporated as optional features within the main strategy contracts. This would eliminate the overhead of separate deployments and remove the configuration complexity of ensuring correct recipient contract association. The logic could be conditionally enabled based on strategy parameters, allowing flexibility while maintaining a single deployment per strategy.
- If maintaining separation of concerns is preferred: The current approach maintains clean boundaries between strategy execution logic and position management utilities. While this separation is architecturally clear, it does introduce practical complexity through the requirement for coordinated deployments and correct recipient configuration. This trade-off may be acceptable if the architectural separation is highly valued for maintainability and testing purposes.

The fundamental decision centers on whether the benefits of consolidated deployment and reduced configuration complexity outweigh the increased code complexity within individual strategy contracts.

Uniswap Labs: Acknowledged.

Cantina Managed: Acknowledged.

4.3.2 Protocol requires EIP-1153 transient storage support without documentation

Severity: Informational

Context: [DynamicArray.sol#L4-L29](#)

Description: The protocol uses EIP-1153 transient storage opcodes (TLOAD/TSTORE) in multiple components, but this deployment requirement is not documented in the README or elsewhere. This creates an undocumented dependency on the Cancun network upgrade. Transient storage usage locations:

1. DynamicArray library ([DynamicArray.sol#L34-L63](#)):
 - Uses `tload` and `tstore` to manage dynamic array lengths transiently.
 - Critical for parameter management in position operations.
2. ReentrancyGuardTransient (via Solady dependency) ([ReentrancyGuardTransient.sol#L11](#)):
 - Used by `TimelockedPositionRecipient` (`src/periphery/TimelockedPositionRecipient.sol:11`).
 - Inherited by `BuybackAndBurnPositionRecipient` and `PositionFeesForwarder`.
 - Provides gas-efficient reentrancy protection using transient storage.

Current deployment status: The README shows deployments on Mainnet, Unichain, Base, and Sepolia. All of these networks currently support the Cancun upgrade and thus have EIP-1153 available. However, the documentation does not mention this as a deployment prerequisite.

Potential implications: While the currently deployed chains support Cancun, the lack of documentation could lead to:

- Failed deployments on chains without Cancun support.
- Confusion during deployment planning for new chains.
- Issues with custom rollups or testnets that haven't upgraded to Cancun.
- Wasted development time attempting deployments on incompatible chains.

The Cancun upgrade (which includes EIP-1153) was activated on Ethereum mainnet in March 2024, and most major L2s have since upgraded. However, not all EVM-compatible chains necessarily support these opcodes, particularly custom rollups or alternative L1s.

Recommendation: Consider adding a deployment requirements section to the README that explicitly documents the EIP-1153/Cancun upgrade requirement. This would help prevent deployment issues and set clear expectations for potential integrators. A suggested addition to the README could include:

Deployment Requirements

This protocol requires deployment on networks that support the Cancun upgrade,
→ specifically:

- **EIP-1153 (Transient Storage)**: Used by the `DynamicArray` library and
→ `ReentrancyGuardTransient` for gas-efficient temporary storage

Supported networks include Ethereum mainnet (post-Cancun), Base, Optimism, Arbitrum, and
→ other L2s that have implemented the Cancun upgrade. Before deploying on a new
→ network, verify that transient storage opcodes (`TLOAD`/`TSTORE`) are supported.

This documentation would provide clarity for teams planning deployments and help avoid unexpected failures on networks without Cancun support.

Uniswap Labs: Fixed in PR 110.

Cantina Managed: Fix verified.

4.3.3 Missing validation for maxCurrencyAmountForLP parameter

Severity: Informational

Context: LBPStrategyBase.sol#L79-L107

Description: The LBPStrategyBase constructor does not validate that maxCurrencyAmountForLP is non-zero. While this parameter can be intentionally set to zero for specific use cases, a zero value will cause migration to fail even when the auction successfully raises funds.

The maxCurrencyAmountForLP parameter determines the maximum amount of currency that can be used to create the initial liquidity position in the Uniswap V4 pool. When set to zero, the following execution flow occurs during migration:

1. In `_prepareMigrationData()` (src/strategies/lbp/LBPStrategyBase.sol:282), the effective currency amount is calculated as:

```
uint128 currencyAmount = uint128(FixedPointMathLib.min(auction.currencyRaised(),  
→ maxCurrencyAmountForLP));
```

If `maxCurrencyAmountForLP = 0`, then `currencyAmount = 0` regardless of how much the auction raised.

2. The `TokenPricing.calculateAmounts()` function (src/libraries/TokenPricing.sol:90-121) receives `currencyAmount = 0` and returns both `initialTokenAmount = 0` and `initialCurrencyAmount = 0`.
3. The liquidity calculation (src/strategies/lbp/LBPStrategyBase.sol:293-299) receives zero amounts and produces `liquidity = 0`, which will cause the migration to revert.

Validation gap:

The `_validateMigration()` function (LBPStrategyBase.sol#L268-L271) checks that `auction.currencyRaised() != 0`, but this validation occurs before the `min()` operation in `_prepareMigrationData()`. A strategy with `maxCurrencyAmountForLP = 0` can pass validation but fail during execution.

As a consequence, when migration fails due to `maxCurrencyAmountForLP = 0`:

- The V4 pool cannot be initialized and liquidity cannot be deployed.
- Raised currency and reserve tokens remain in the strategy contract until `sweepBlock`.
- Users must wait for the operator to sweep funds after `sweepBlock`.

As noted in the README's warnings, "it is trivially easy to create a LBPStrategy and corresponding Auction with malicious parameters," and users must validate all parameters before interacting with a strategy. A zero `maxCurrencyAmountForLP` represents another parameter that could be misconfigured accidentally or set maliciously.

Recommendation: Consider adding validation in `_validateMigratorParams()` to ensure `maxCurrencyAmountForLP` is non-zero, which would provide clearer feedback at construction time:

```
function _validateMigratorParams(uint128 _totalSupply, MigratorParameters memory
→ migratorParams) internal pure {
    // sweep block validation (cannot be before or equal to the migration block)
    if (migratorParams.sweepBlock <= migratorParams.migrationBlock) {
        revert InvalidSweepBlock(migratorParams.sweepBlock,
        → migratorParams.migrationBlock);
    }
+   // maxCurrencyAmountForLP validation (cannot be zero)
+   else if (migratorParams.maxCurrencyAmountForLP == 0) {
+       revert MaxCurrencyAmountForLPIsZero();
+   }
    // token split validation (cannot be greater than or equal to 100%)
    else if (migratorParams.tokenSplitToAuction >= TokenDistribution.MAX_TOKEN_SPLIT) {
        revert TokenSplitTooHigh(migratorParams.tokenSplitToAuction,
        → TokenDistribution.MAX_TOKEN_SPLIT);
    }
    // ... rest of validations
}
```

This validation would prevent deployment of strategies with this particular misconfiguration and provide immediate feedback rather than allowing the issue to surface during migration. However, given the existing guidance that users must validate all strategy parameters before interaction, this validation is not strictly necessary for security.

Uniswap Labs: Fixed in PR 105.

Cantina Managed: Fix verified.

4.3.4 Transient storage in DynamicArray adds complexity and gas overhead

Severity: Informational

Context: DynamicArray.sol#L31-L59

Description: The `DynamicArray` library uses transient storage to track array length separately from the memory array itself, adding complexity and gas costs. A simpler alternative exists that stores length directly in the memory array, eliminating the need for transient storage while maintaining the same functionality.

The current design in `src/libraries/DynamicArray.sol` uses transient storage (EIP-1153) to track array state:

1. Transient length tracking: Each `append()` operation reads and writes to transient storage via `tload/tstore`.
2. Initialization guard: Uses a bitmask (`INITIALIZED_MASK`) in transient storage to prevent multiple initializations.

3. Separate truncation step: Requires calling `truncate()` to set the final array length in memory.
4. Complex bitmask logic: Packs initialization flag and length into a single transient storage slot using masks.
5. Strict usage requirements: Documentation warns "Do NOT use `append` and `truncate` on arrays not created via this library as the behavior will be undefined".

Gas costs:

- `init()`: One `tstore` operation (~100 gas).
- Each `append()`: One `tload` + one `tstore` operation (~200 gas per append).
- Total overhead for a typical 3-parameter array: ~700 gas.

Alternative memory-based approach: Memory arrays in Solidity can have their length modified directly via assembly. The simpler alternative would be to:

1. Allocate fixed-size array: `params = new bytes[] (MAX_PARAMS_SIZE)`.
2. Set initial length to 0: `assembly { mstore(params, 0) }`.
3. On append:
 - Read current length from memory via assembly.
 - Check bounds: `if (length >= MAX_PARAMS_SIZE) revert()`.
 - Store parameter at calculated offset.
 - Increment length: `mstore(params, add(length, 1))`.
4. No truncation needed - length is already correct in memory.

This approach:

- Eliminates all transient storage operations (saves ~700 gas for typical usage).
- Removes the need for `truncate()`.
- Simplifies the implementation (no bitmask logic).
- Removes the multiple initialization guard (could be handled by standard reentrancy protection).
- Makes the code more straightforward to understand and audit.

Optimizer considerations: While `data.length` in Solidity compiles to `mload(data)`, there is a theoretical risk with optimizer behavior. Since memory arrays are normally immutable in length (no `.push()` support), the optimizer could potentially:

1. Cache the length value on the stack after the first read.
2. Reuse the cached value instead of re-reading from memory.
3. Replace subsequent `mload(data)` operations with stack dup operations.

This optimization would be based on the assumption that memory array lengths never change, which would be violated when manually modifying length via assembly. While current testing shows the optimizer doesn't do this, and it's unlikely to optimize across function boundaries, relying on this behavior involves a non-standard assumption about Solidity's memory management.

Mitigation for optimizer risk: To safely implement the memory-based approach, length should always be read via explicit assembly:

```
assembly {
    length := mload(params)
}
```

The optimizer does not optimize within assembly blocks, ensuring `mload` operations are never replaced with cached stack values. This provides a safe way to implement the simpler approach without relying on optimizer behavior.

Recommendation: Consider refactoring the `DynamicArray` library to use memory-based length tracking instead of transient storage. To avoid potential optimizer issues, always read length explicitly via assembly:

```

library DynamicArray {
    error LengthOverflow();

    uint24 constant MAX_PARAMS_SIZE = 6;

    function init() internal pure returns (bytes[] memory params) {
        params = new bytes[](MAX_PARAMS_SIZE);
        assembly {
            mstore(params, 0) // Set initial length to 0
        }
    }

    function append(bytes[] memory params, bytes memory param) internal pure returns
    → (bytes[] memory) {
        assembly {
            // Always read length via assembly to avoid optimizer assumptions
            let length := mload(params)
            if iszero(lt(length, MAX_PARAMS_SIZE)) {
                mstore(0x00, 0x8ecbb27e) // LengthOverflow() selector
                revert(0x1c, 0x04)
            }
            let slot := add(add(params, 0x20), mul(length, 0x20))
            mstore(slot, param)
            mstore(params, add(length, 1)) // Increment length
        }
        return params;
    }

    // No truncate() needed - length is already correct
}

```

Benefits of this approach:

- Simpler implementation with less assembly and no bitmask logic.
- ~700 gas savings for typical usage (eliminates transient storage operations).
- No `truncate()` step required.
- Safer optimizer behavior by using explicit assembly for length reads.
- Easier to understand and maintain.
- Removes dependency on EIP-1153 (though this is now a deployment requirement anyway).

The multiple initialization guard can be handled through standard reentrancy protection patterns if needed, separating that concern from the dynamic array implementation.

Uniswap Labs: Fixed in PR 104.

Cantina Managed: Fix verified.

4.3.5 Unused imports across the scope

Severity: Informational

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

Description: The following dependency imports from the specified contracts are unused:

- GovernedLBPStrategy:

```

import {ModifyLiquidityParams, SwapParams} from
→ "@uniswap/v4-core/src/types/PoolOperation.sol";

```

- FullRangeLBPStrategy:

```

import {IHooks} from "@uniswap/v4-core/src/interfaces/IHooks.sol";

```

- MerkleClaim:

```
import "src/libraries/external/MerkleClaimHelpers.sol";
```
- BuybackAndBurnPositionRecipient:

```
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";
```
- PositionFeesForwarder:

```
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";
```
- AdvancedLBPSstrategy:

```
import {IHooks} from "@uniswap/v4-core/src/interfaces/IHooks.sol";
```

Recommendation: Remove the following imports from the contracts specified.

Uniswap Labs: Fixed in PR 109.

Cantina Managed: Fix verified.

5 PR 103 Review Findings

5.1 Gas Optimization

5.1.1 Redundant external call to fetch LBP initialization parameters

Severity: Gas Optimization

Context: LBPSstrategyBase.sol#L265-L293

Description: The `migrate()` function calls both `_validateMigration()` and `_prepareMigrationData()` sequentially. Both of these internal functions make an external call to `initializer.lbpInitializationParams()` to retrieve the same `LBPInitializationParams` data:

```
function migrate() external {
    _validateMigration(); // Calls initializer.lbpInitializationParams()
    MigrationData memory data = _prepareMigrationData(); // Calls
    ↳ initializer.lbpInitializationParams() again
    // ...
}
```

This results in a duplicate external call that fetches identical data, unnecessarily increasing gas costs for the migration operation.

Recommendation: Consider refactoring to fetch the initialization parameters once and pass them to the validation and preparation functions:

```
function migrate() external {
    LBPInitializationParams memory lbpParams = initializer.lbpInitializationParams();
    _validateMigration(lbpParams);
    MigrationData memory data = _prepareMigrationData(lbpParams);
    // ...
}
```

Uniswap Labs: Fixed in PR 113.

Cantina Managed: Fix verified.

5.2 Informational

5.2.1 Check-effects-interactions pattern violation in initializer setup

Severity: Informational

Context: LBPStrategyBase.sol#L118-L145

Description: The `onTokensReceived()` function in `LBPStrategyBase.sol` does not follow the check-effects-interactions (CEI) pattern. After deploying and validating the initializer contract, the function calls the external `onTokensReceived()` hook before updating the `initializer` storage variable:

```
// Transfer the tokens to the initializer contract
Currency.wrap(token).transfer(address(_initializer), supply);
// Call the `onTokensReceived` hook
_initializer.onTokensReceived(); // External call before state update
_initializer = _initializer; // State update after external call
```

This creates a potential reentrancy vector where a malicious initializer could call back while `initializer` is still `address(0)`. In the current controlled environment with trusted factories and clearing auctions, this presents minimal practical risk.

Recommendation: Consider moving the state update before the external call to align with CEI best practices:

```
// Transfer the tokens to the initializer contract
Currency.wrap(token).transfer(address(_initializer), supply);
+ _initializer = _initializer;
// Call the `onTokensReceived` hook
_initializer.onTokensReceived();
- _initializer = _initializer;

emit InitializerCreated(address(_initializer));
```

Uniswap Labs: Fixed in PR 113.

Cantina Managed: Fix verified.

5.2.2 Consider using timestamps instead of block numbers for time-based checks

Severity: Informational

Context: LBPStrategyBase.sol#L36

Description: The `LBPStrategyBase` contract inherits from `BlockNumberish` and uses block numbers for time-based validations such as `migrationBlock` and `sweepBlock`. Functions like `migrate()` and `sweepToken()` check the current block number via `_getBlockNumberish()` to determine whether operations are permitted.

While block numbers provide a measure of time progression, timestamps may offer better user experience by allowing operators to specify exact times rather than estimating block numbers. Timestamps are also more intuitive for off-chain systems and user interfaces, reducing potential confusion when planning migrations or sweep operations. Additionally, on chains like Arbitrum, block numbers can be ambiguous. It may not be immediately clear whether Arbitrum block numbers or L1 block numbers should be used, which could lead to human errors during configuration.

Recommendation: Consider refactoring the contract to use `block.timestamp` instead of block numbers for time-based checks. This would involve:

```
- abstract contract LBPStrategyBase is ILBPStrategyBase, SelfInitializerHook,
→ BlockNumberish {
+ abstract contract LBPStrategyBase is ILBPStrategyBase, SelfInitializerHook {
```

And updating the time parameters from `uint64` block numbers to `uint256` timestamps, along with corresponding validation logic. This change would make the contract more user-friendly and align with common patterns in time-dependent DeFi protocols.

Uniswap Labs: Acknowledged. Won't make this change since auction is block based but agree with the sentiment

Cantina Managed: Acknowledged.

5.2.3 Hardcoded interface ID may lead to regression issues

Severity: Informational

Context: ILBPIinitializer.sol#L9

Description: The ILBPIinitializer interface defines a hardcoded interface ID constant:

```
bytes4 constant ILBP_INITIALIZER_INTERFACE_ID = 0x66981dad;
```

This hardcoded value must be manually updated if the interface changes. If the interface is modified but the constant is not updated accordingly, the interface ID will become stale, potentially causing validation failures or incorrect behavior in contracts that rely on this identifier for ERC165 interface checks.

Recommendation: Consider using Solidity's built-in `type().interfaceId` to automatically derive the interface ID from the interface definition:

```
- bytes4 constant ILBP_INITIALIZER_INTERFACE_ID = 0x66981dad;  
+ bytes4 constant ILBP_INITIALIZER_INTERFACE_ID = type(ILBPIinitializer).interfaceId;
```

This approach ensures the interface ID automatically stays in sync with any changes to the interface, reducing the risk of regression issues and eliminating the need for manual updates.

Uniswap Labs: Fixed in PR 113.

Cantina Managed: Fix verified.