# Euler Swap

Security Review

Cantina Managed review by:

**Cryptara**, Security Researcher
**Slowfi**, Security Researcher

September 18, 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

Euler Labs is a team of developers and quantitative analysts building DeFi applications for the future of finance.

From Aug 31st to Sep 6th the Cantina team conducted a review of euler-swap on commit hash 6f5bde45. The team identified a total of **17** issues:

**Issues Found**

| **Severity** | **Count** | **Fixed** | **Acknowledged** |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 1 | 1 | 0 |
| Medium Risk | 3 | 2 | 1 |
| Low Risk | 3 | 2 | 1 |
| Gas Optimizations | 3 | 3 | 0 |
| Informational | 7 | 5 | 2 |
| **Total** | **17** | **13** | **4** |

# 3 Findings

## 3.1 High Risk

### 3.1.1 Reentrancy in `redeemValidityBond`

**Severity:** High Risk

**Context:** EulerSwapRegistry.sol#L298-L300

**Description:** In `EulerSwapRegistry` `redeemValidityBond` transfers ETH to `recipient` before clearing `validityBonds` mapping. Because `recipient` is externally controlled, its fallback can reenter registry while the bond mapping still holds the value, allowing repeated withdrawals of the same bond and interleaving state changes in a single transaction. After draining the registry's bond balance, it is not possible to unregister other pools because there is no ETH left to pay their bonds; doing so would require sending funds back to the contract first. This finding did not put LP funds at risk.

**Recommendation:** Consider to clear state before the external transfer, set `validityBonds[pool] = 0` prior to the call and protect all entry points that can reach bond payout with a simple `nonReentrant` guard; optionally adopt a pull-payment pattern (record owed amounts and let recipients withdraw) to eliminate this surface entirely, ensuring the registry cannot be locked out of uninstalling pools unless the drained funds are first returned.

```diff
+ validityBonds[pool] = 0;
  (bool success,) = recipient.call{value: bondAmount}("");
  require(success, ChallengeMissingBond());
- validityBonds[pool] = 0;
```

**Euler:** A reentrancy guard was added, and a storage write was reordered to preserve the checks-effects-interactions pattern. Note that this did not affect LP funds, just the posted liquidity bonds (which are expected to be small -- just enough to cover gas costs of a challenge).

**Cantina Managed:** Fix verified.

## 3.2 Medium Risk

### 3.2.1 Exploitable false positive in `challengePool` validation

**Severity:** Medium Risk

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

**Description:** The `challengePool` function in the `EulerSwapRegistry` contract is vulnerable to manipulation by tokens with transfer hooks or malicious tokens. The current implementation wraps both the `IERC20(tokenIn).safeTransferFrom(...)` and the swap call inside a self-call (`challengePoolAttempt`). This design allows a token's transfer hook to revert with `E_AccountLiquidity.selector`, which is indistinguishable from the intended failure path (a genuine liquidity failure during the swap).

As a result, a malicious challenger can exploit this mechanism to force a false positive, redeem the validity bond, and drain the contract. This issue is further exacerbated by the potential for re-entrancy attacks, which could amplify the impact of the exploit.

**Proof of Concept:**

```solidity
// test/Basic.t.sol:
 function test_swap_hook() public monotonicHolderNAV {

        deal(holder, 1e18);

        eulerSwap = createEulerSwap(60e18, 60e18, 0, 1e18, 1e18, 0.4e18, 0.85e18);

        uint256 amountIn = 1e18;
        uint256 amountOut =
            periphery.quoteExactInput(address(eulerSwap), address(assetTST), address(assetTST2), amountIn);
        assertApproxEqAbs(amountOut, 0.9974e18, 0.0001e18);

        assetTST.mint(address(this), amountIn);

        assetTST.transfer(address(eulerSwap), amountIn);
        eulerSwap.swap(0, amountOut, address(this), "");
        assertEq(assetTST2.balanceOf(address(this)), amountOut);

        // The bond should not be 0
        assertNotEq(eulerSwapRegistry.validityBond(address(eulerSwap)), 0);

        // Expecte behaviour
        uint256 snapshot = vm.snapshotState();
        vm.expectRevert();
        eulerSwapRegistry.challengePool(
            address(eulerSwap), address(assetTST), address(assetTST2), 0, true, address(5555)
        );
        vm.revertToState(snapshot);

        // Lets emulate a transfer hook reverting on assetTST with E_AccountLiquidity.selector
        // showcasing why we need to have the `safeTransferFrom` outside of the call
        vm.mockCallRevert(
            address(assetTST),
            0,
            // data
            abi.encodeWithSelector(
                TestERC20.transferFrom.selector
            ),
            abi.encodeWithSelector(E_AccountLiquidity.selector)
        );

        eulerSwapRegistry.challengePool(
            address(eulerSwap), address(assetTST), address(assetTST2), 0, true, address(5555)
        );

        // The bond will be 0
        assertEq(eulerSwapRegistry.validityBond(address(eulerSwap)), 0);
    }
```

**Recommendation:** Refactor the `challengePool` function to separate the `safeTransferFrom` call from the swap execution. This can be achieved by using a `try/catch` block for the swap logic only preventing from untrusted parts of the code to manipulate the returned selector.

**Euler:** Acknowledged. The risk is minimal. Tokens with transfer hooks are extremely rare and not supported in their EVK. Additionally, even if such tokens were used, recipients must opt in, and the potential impact would be limited.

**Cantina Managed:** Acknowledged.

### 3.2.2 Unchecked borrow vault access in `QuoteLib` causes swap revert when borrowing is disabled

**Severity:** Medium Risk

**Context:** QuoteLib.sol#L132

**Description:** In `QuoteLib` contract the `caclLimis` reads debt via `debtOf` on the borrow vault unconditionally. Pool configurations allow `borrowVault0/1` to be `address(0)` to disable borrowing; in that case, any quote or swap that reaches this path will attempt to call `debtOf` on the zero address and revert due to empty return data or a call to a non-contract, breaking otherwise valid ”no-borrow” configurations.

**Recommendation:** Consider to guard the optional borrow vault before use and treat missing borrowing as zero debt; when the borrow vault is unset, compute `debt = 0` and proceed with limit math, while ensuring execution paths that would require a borrow cleanly reject with a clear error (e.g., `BorrowDisabled` or `SwapRejected`). You may also validate at activation or reconfigure that when a borrow vault is unset the configuration cannot rely on borrowing, but the quote path should not hard revert solely because the borrow vault is `address(0)`.

**Euler:** Fixed in commit 598a1fb9.

**Cantina Managed:** Fix verified.

### 3.2.3 Flash loan path always reverts

**Severity:** Medium Risk

**Context:** EulerSwap.sol#L255-L277

**Description:** `EulerSwap` was intended to support flash loans on the direct path (in addition to swaps), but the current flow computes amounts before the callback. Specifically, amounts are accounted in the call to `SwapLib.amounts`, which measures the contract balances and snapshots them ahead of invoking the user callback. After that pre-callback snapshot, the contract performs withdraws, calls the callee, then deposits "all available" funds and verifies the curve. Because the snapshot happens before the callback, any tokens returned during the callback are not validated against an exact repay target: a direct flash attempt that requests output with zero net input will always revert at the invariant check, and if a caller sends back more than required under permissive reserves, that surplus is treated as extra input and can remain on the pool contract rather than being refunded.

This is medium severity because it breaks flash-loan usability on the direct path and can strand tokens on the pool contract if users transfer back during the callback; it is not a direct theft vector, but it can cause operational loss and DoS for integrators expecting flash-loan semantics.

**Proof of Concept:** The test below activates the pool on the boundary and tries to "flash borrow" token0 by asking for output with zero input, returning funds during the callback. The swap reverts with `CurveLib.CurveViolation`, confirming the flash path is not usable. After revert, no funds remain stuck on either the pool or the callee.

```
function test_direct_flashloan_reverts_and_no_stuck_funds() public {
    // Boundary activation (equilibrium == reserves), any out with zero input should fail
    (IEulerSwap.StaticParams memory s, IEulerSwap.DynamicParams memory d, IEulerSwap.InitialState memory i) =
        _params(60e18, 60e18, 0, 1e18, 1e18, 0.4e18, 0.85e18, true);
    // Activate on boundary
    i.reserve0 = 60e18; i.reserve1 = 60e18;
    EulerSwap pool = _deployHookPool(s, d, i);

    FlashReturnCallee callee = new FlashReturnCallee(address(pool));

    uint256 out0 = 1e18;
    bytes memory data = abi.encode(address(asset0), out0);

    vm.expectRevert(CurveLib.CurveViolation.selector);
    pool.swap(out0, 0, address(callee), data);

    // Entire swap reverted; no tokens stranded on pool or callee
    assertEq(asset0.balanceOf(address(pool)), 0);
    assertEq(asset1.balanceOf(address(pool)), 0);
    assertEq(asset0.balanceOf(address(callee)), 0);
    assertEq(asset1.balanceOf(address(callee)), 0);
    assertFalse(callee.called());
}
```

**Recommendation:** Consider to implement explicit flash-loan semantics: snapshot balances, transfer the loan, invoke the callback, then require an exact repay plus fee before deposits and invariant checks, refunding any surplus to the caller. If flash loans are not desired, document that the callback is not a V2-style flash mechanism and add safeguards to prevent surplus from being silently retained.

**Euler:** Fixed in commit df6b0900.

**Cantina Managed:** Fix verified.

## 3.3 Low Risk

### 3.3.1 Missing Validation for `swapHook` and `swapHookedOperations`

**Severity:** Low Risk

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

**Description:** The `EulerSwap` contract lacks proper validation for the `swapHook` and `swapHookedOperations` parameters during configuration and reconfiguration. Specifically, if `(swapHookedOperations & 3) != 0`, the `swapHook` is expected to contain a valid non-zero address. Failure to validate this can lead to misconfigured pools that always revert during operations such as `QuoteLib.getFee()` or `SwapLib.finish()`. This misconfiguration causes the pool to become unusable and prevents challenges from being executed, as the `challengePool` mechanism relies on specific revert selectors like `E_AccountLiquidity`.

Additionally, the `challengePool` mechanism does not account for pools that fail due to invalid hooks or other misconfigurations. For example, a pool owner could set an invalid `swapHook` that always reverts, effectively preventing the pool from being challenged and allowing the bond to remain locked indefinitely. This creates a loophole where invalid pools cannot be penalized or removed from the registry.

**Recommendation:** Validation During Configuration and Reconfiguration:

- Ensure that if `(swapHookedOperations & 3) != 0`, the `swapHook` is a valid non-zero address.
- This validation should be enforced during both the `activate` and `reconfigure` functions to prevent misconfigured pools.

Enhance `challengePool` Mechanism:

- Extend the `challengePool` logic to handle additional failure scenarios beyond `E_AccountLiquidity`.
- Use a `try/catch` block to capture all reverts during the challenge process.
- Maintain a whitelist of valid revert selectors (e.g., `E_AccountLiquidity.selector`, `HookError`) and treat any other reverts as invalid.
- If the revert reason is not whitelisted, the bond should not be redeemed, and the challenge should fail.

Document Failure Scenarios:

- Clearly document the various ways a pool can fail (e.g., liquidity issues, invalid hooks, misconfigurations) and ensure the `challengePool` mechanism accounts for these scenarios.

**Euler:** Fixed in commits 70cd473b and 22bb23aa.

**Cantina Managed:** The client addressed the issue by implementing fixes in two parts (70cd473b and 22bb23aa). They added sanity checks to the hook configuration as recommended, and introduced a custom `HookError()` wrapper for hook call failures. This ensures that pools encountering such errors (e.g., during `afterSwap`) can be safely removed from the registry via a challenge.

### 3.3.2 Unnecessary Transfer When `feeAmount` is Zero

**Severity:** Low Risk

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

**Description:** In the `SwapLib` library, the `doDeposit` function performs a transfer to the `feeRecipient` even when `feeAmount` is zero:

```
IERC20(assetInput).safeTransfer(ctx.sParams.feeRecipient, feeAmount);
```

This can cause issues with certain tokens that revert on zero-value transfers, leading to unnecessary failures during the swap process. While most ERC20 tokens allow zero-value transfers, some implementations do not, which can make the contract incompatible with such tokens.

**Recommendation:** Add a conditional check to ensure that the transfer is only executed when feeAmount is greater than zero. This will prevent unnecessary transfers and ensure compatibility with tokens that revert on zero-value transfers.

**Euler:** Fixed in commit 3b693c41.

**Cantina Managed:** Fix verified.

### 3.3.3 Single-step curator transfer

**Severity:** Low Risk

**Context:** EulerSwapRegistry.sol#L132-L134

**Description:** In `EulerSwapRegistry`, `transferCurator` is gated by `onlyCurator` and sets `curator` immediately in a single step. This makes the role change effective without confirmation by the new curator, which can be error-prone and reduces operational safety.

**Recommendation:** Consider to adopt a two-step transfer pattern:

- `transferCurator` $\rightarrow$ sets `pendingCurator` and emits a start event.

- `acceptCuratorship` → callable by `pendingCurator` to finalize and clear the pending value, emitting a completion event.

**Euler:** We have not done this in the majority of our contracts and it hasn't been an issue so far. In the couple places we've done this previously, it added a lot of operational overhead, so we are not planning on adopting this pattern more broadly.

**Cantina Managed:** Acknowledged.

## 3.4 Gas Optimization

### 3.4.1 Duplicate bond redemption in `challengePool`

**Severity:** Gas Optimization

**Context:** EulerSwapRegistry.sol#L190-L194

**Description:** In `EulerSwapRegistry` contract `challengePool` function calls `redeemValidityBond` is before `uninstall`, while `uninstall` also invokes `redeemValidityBond`. Because `uninstall` is executed immediately after, the first redemption is redundant on the success path and incurs an unnecessary external call and gas cost.

**Recommendation:** Consider to perform bond redemption in one place only. A simple change is to remove internal call to `redeemValidityBond` and let `uninstall` handle the payout. Keep event semantics consistent and ensure revert behavior is unchanged.

**Euler:** Fixed in commit 66475184.

**Cantina Managed:** Fix verified.

### 3.4.2 Fold exact-out fee into the `else` branch in `QuoteLib`

**Severity:** Gas Optimization

**Context:** QuoteLib.sol#L89-L98

**Description:** In `QuoteLib` the branch that handles `exactOut` first enforces limits and then inflates `quote` for fees using a separate `if (!exactIn)` after the branch. This is semantically identical to performing the inflation inside the `else` branch directly after the limits check, and doing so improves readability and can save a tiny amount of gas by avoiding an extra conditional.

Current:

```
if (exactIn) {
    require(amount <= inLimit && quote <= outLimit, SwapLimitExceeded());
} else {
    require(amount <= outLimit && quote <= inLimit, SwapLimitExceeded());
}
// exactOut: inflate required amountIn
if (!exactIn) quote = (quote * 1e18) / (1e18 - fee);
```

Equivalent and clearer:

```
if (exactIn) {
    require(amount <= inLimit && quote <= outLimit, SwapLimitExceeded());
} else {
    require(amount <= outLimit && quote <= inLimit, SwapLimitExceeded());
    // exactOut: inflate required amountIn
    quote = (quote * 1e18) / (1e18 - fee);
}
```

**Recommendation:** Consider to move the `quote` inflation for the exact-out path into the `else` branch directly after its limits check. This keeps all exact-out logic localized, makes the control flow self-evident, and may produce a marginal gas improvement from one fewer conditional.

```
   if (exactIn) {
              // if `exactIn`, `quote` is the amount of assets to buy from the AMM
              require(amount <= inLimit && quote <= outLimit, SwapLimitExceeded());
  } else {
              // if `!exactIn`, `amount` is the amount of assets to buy from the AMM
              require(amount <= outLimit && quote <= inLimit, SwapLimitExceeded());
```

```
+            // exactOut: inflate required amountIn
+            quote = (quote * 1e18) / (1e18 - fee);
  }

-   // exactOut: inflate required amountIn
-   if (!exactIn) quote = (quote * 1e18) / (1e18 - fee);
```

**Euler:** Fixed in commit c6a4b8f7.

**Cantina Managed:** Fix verified.

### 3.4.3 Duplicated reentrancy guard across `EulerSwap` and `UniswapHook`

**Severity:** Gas Optimization

**Context:** EulerSwap.sol#L45-L52

**Description:** `EulerSwap` defines a `nonReentrant` modifier that flips `CtxLib.State.status` between `1` and `2`. The same guard logic exists in `UniswapHook` (`nonReentrantHook` in the linked file).

**Recommendation:** Consider to centralize the guard in a single source of truth and have both `EulerSwap` and `UniswapHook` use it. Replace magic numbers with named constants (e.g., `STATUS_UNLOCKED = 1`, `STATUS_LOCKED = 2`) to avoid drift, and keep the revert selector consistent across both call sites.

**Euler:** Fixed in commit 95a0ec6c.

**Cantina Managed:** Fix verified.

## 3.5 Informational

### 3.5.1 Inconsistent Naming Convention for Internal Functions

**Severity:** Informational

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

**Description:** The `EulerSwapRegistry` contract uses inconsistent naming conventions for internal functions. While some internal functions like `_uninstall` and `_redeemValidityBond` follow the underscore prefix convention, others such as `getSlice` and `isValidVault` do not. This inconsistency can lead to confusion for developers and auditors, as it becomes unclear which functions are intended for internal use. Consistent naming conventions are critical for maintaining code readability and reducing the risk of unintended usage.

**Recommendation:** Update all internal function names to follow a consistent naming convention. For example, prepend an underscore (_) to all internal function names, including `getSlice` and `isValidVault`. This will align with the existing convention used for `_uninstall` and `_redeemValidityBond`, making the codebase more uniform and easier to understand.

**Euler:** Acknowledged. We usually don't do the leading underscore for internal functions. It might be a good habit to get into though. However, most of this code-base does not follow this convention. In particular, `_uninstall` and `_redeemValidityBond` don't seem to exist, and are without underscores.

**Cantina Managed:** Acknowledged.

### 3.5.2 Unused Errors in `CurveLib`

**Severity:** Informational

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

**Description:** Found by client. The `CurveLib` library defines the following errors that are not used anywhere in the code:

```
error Overflow();
error CurveViolation();
```

These unused errors increase the size of the compiled bytecode unnecessarily and may confuse developers or auditors by implying functionality that does not exist.

**Recommendation:** Remove the unused errors from the CurveLib library to improve code clarity. If these errors are intended for future use, document their purpose to avoid confusion.

**Euler:** Fixed in commit db4ecc92.

**Cantina Managed:** Fix verified.

### 3.5.3 Removal of `Ternary` Library Usage for Clarity

**Severity:** Informational

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

**Description:** The `CurveLib` library uses the `Ternary` library to simplify conditional expressions, such as:

```
shift = (shiftSquaredB < shiftFourAc).ternary(shiftFourAc, shiftSquaredB);
```

While the Ternary library improves contract size and runtime gas efficiency, the same logic can be expressed using the native Solidity ternary operator:

```
shift = shiftSquaredB < shiftFourAc ? shiftFourAc : shiftSquaredB;
```

This change improves code clarity and reduces dependency on the Ternary library. If the Ternary library is retained, unused overloads should be removed to reduce code complexity.

**Recommendation:**

- Replace the Ternary library usage with the native Solidity ternary operator for clarity.
- If the Ternary library is retained, remove any unused overloads to improve brevity and maintainability.

**Euler:** Fixed in commit 382d44f9.

**Cantina Managed:** Fix verified.

### 3.5.4 `beforeSwap` name for hook is misleading

**Severity:** Informational

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

**Description:** *Found by the Euler team*.

The hook named `beforeSwap` does not run at the very start of the swap. Amounts are first accounted (via `SwapLib.amounts` reading contract balances and fixing deltas) and other mid-swap steps occur, after which the hook logic is involved. This naming suggests a pre-entry "before swap starts" hook that can gate or parameterize the entire operation, but in practice it executes mid-flow. The mismatch can mislead integrators who expect a true pre-swap point for auctions, circuit-breakers, or fee setting aligned with quoting.

**Recommendation:** Consider to either rename the current hook to reflect its mid-swap placement, or introduce a new true pre-swap hook that runs immediately on swap entry (and during quoting) before any accounting or token movement. The pre-swap hook should be invoked under the lock, accept the `readOnly` signal for quote symmetry, be mandatory when enabled (non-zero address when the corresponding bit is set), and reject by returning the sentinel fee or reverting. Update the documentation to clarify the execution order and responsibilities of each hook to avoid misuse.

**Euler:** What used to be `beforeSwap` has been renamed to `getFee`, and a new `beforeSwap` hook was added that actually does run before any tokens have moved. Fixed in commit 22bb23aa.

**Cantina Managed:** Fix verified.

### 3.5.5 Missing events for curator and registry parameter updates

**Severity:** Informational

**Context:** EulerSwapRegistry.sol#L132-L144

**Description:** In `EulerSwapRegistry`, the following governance/operational setters do not emit events:

- `transferCurator(address newCurator)`.
- `setMinimumValidityBond(uint256 newMinimum)`.
- `setValidVaultPerspective(address newPerspective)`.

Without events, off-chain monitoring and indexers cannot reliably track changes to the curator, minimum bond requirement, or the vault verification perspective.

**Recommendation:** Consider to emit explicit events for each update, e.g., `CuratorTransferred(address indexed oldCurator, address indexed newCurator)`, `MinimumValidityBondUpdated(uint256 oldValue, uint256 newValue)`, and `ValidVaultPerspectiveUpdated(address indexed oldPerspective, address indexed newPerspective)`. Use `indexed` parameters where helpful for filtering, and emit after state changes to reflect final values.

**Euler:** Fixed in commit 232c184d.

**Cantina Managed:** Fix verified.

### 3.5.6 Outdated return-range comment in `CurveLib` conflicts with saturating behavior

**Severity:** Informational

**Context:** CurveLib.sol#L63

**Description:** *Found by the Euler team*.

In `CurveLib` NatSpec states that the function "returns `y` ... guaranteed to satisfy `y0    y    2^112 - 1`". The implementation no longer guarantees that range and instead uses a saturating behavior that returns `type(uint256).max` on overflow. This mismatch can mislead readers and integrators into assuming a bounded `uint112`-range output, potentially masking overflow conditions and complicating downstream checks that expect an in-range reserve value.

**Recommendation:** Consider to update the documentation to reflect the current saturating semantics and specify that `type(uint256).max` is a sentinel indicating overflow. If the sentinel is retained, ensure all call sites explicitly guard against it (treat as failure and revert) and add tests for the overflow path. Alternatively, replace saturation with a clear revert using a dedicated error to preserve the original "range-guaranteed" contract.

**Euler:** Fixed in commit c8e14d2e.

**Cantina Managed:** Fix verified.

### 3.5.7 Reconfigure above the curve enables zero-input dual-asset withdrawals

**Severity:** Informational

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

**Description:** `reconfigure` accepts an `InitialState` that is above/right of the curve (only "not below the curve" is enforced). From such a state, the pool contains excess, unclaimed value. Any caller can execute `swap(out0>0, out1>0, ...)` with zero input and withdraw both assets for free until the reserves land on the curve. This behavior is prevented at activation (initial state must be on-curve, with a strict boundary check), but it is allowed during reconfiguration, which enables accidental donation of tokens. Observed invariants still hold:

- Curve invariant: execution rejects only when a post-swap point would fall below the boundary.
- Other guards such as `minReserve*`, fees, and `expiration` still apply; they do not prevent the described withdrawal of the "above/right slack".

**Proof Of Concept:**

1. Deploy a pool whose activation uses on-curve reserves.
2. Call `reconfigure()` with `InitialState` above/right of `(x0, y0)` (e.g., `{reserve0: 60e18, reserve1: 60e18}` while `equilibriumReserve{0,1}=30e18`).
3. As any address, call:

```
pool.swap(out0 = 10e18, out1 = 10e18, receiver, "");
```

with no prior input transfers.

4. The call succeeds; `receiver` receives both assets; pool reserves reduce to `{50e18, 50e18}`. Repeating is possible until the point reaches the curve; the next such call then reverts with the curve violation.

```
function test_poc_direct_double_free_withdraw() public {
    // Start well above equilibrium: reserves (60,60) with eq (30,30)
    EulerSwap pool = _deployUpRightPool(60e18, 60e18, 30e18, 30e18);

    address attacker = makeAddr("attacker");

    // Request both outs without sending any input; keep new reserves >= equilibrium
    uint256 out0 = 10e18; // new reserve0 = 50e18
    uint256 out1 = 10e18; // new reserve1 = 50e18

    // No pre-transfers, empty callback
    vm.prank(attacker);
    pool.swap(out0, out1, attacker, "");

    // Attacker received both tokens for free
    assertEq(asset0.balanceOf(attacker), out0);
    assertEq(asset1.balanceOf(attacker), out1);

    // Reserves updated down but still above equilibrium; no input deposited
    (uint112 r0, uint112 r1,) = pool.getReserves();
    assertEq(r0, 50e18);
    assertEq(r1, 50e18);
}
```

**Recommendation:** Consider to:

- Enforce on-curve reconfigure: verify the provided `InitialState` lies on the curve (optionally allow a tiny epsilon, projecting to the curve internally when within tolerance).
- Or require explicit opt-in: add a boolean (e.g., `allowValueLeakage`) that must be `true` if `InitialState` is above/right; emit an event with the donated deltas.
- Or auto-snap: when `InitialState` is above/right, compute the curve boundary point and use it instead (emit an event).
- Policy via hook (if hooks are used): in a `beforeSwap` hook, reject zero-input withdrawals while `reserve0>x0 && reserve1>y0` (or restrict to `msg.sender == eulerAccount`).
- UX guardrail: in periphery/CLI, warn on reconfigure if `InitialState` is above/right and display the implied token deltas.

These changes preserve the existing invariant (no below-curve states) while preventing unintended donation of pool assets during reconfiguration.

**Euler:** This is by design. It's functionally the same as when somebody ”over-swaps” (ie provides more input token and/or takes less output token than necessary). The excess is available for next swapper. As mentioned, you can even get it with 0 input tokens.

We were on the fence about enforcing the strict on-curve boundary on a reconfigure. On one hand it prevents a reconfiguration from screwing up and losing value, but on the other hand getting it exactly precise after rounding and adds a large complexity overhead for some use-cases, such as pooled models that reconfigure on deposit.

This is documented in the developer guide, but we'll try to make it more clear that it is important to get this right. The recommendations are good ideas, but at this time we think it makes more sense to allow the reconfigurer to decide what to do.

**Cantina Managed:** Acknowledged.