Euler Swap
Security Review

Cantina Managed review by:

MiloTruck, Lead Security Researcher
Cryptara, Security Researcher

May 3, 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	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

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 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 Apr 25th to Apr 29th the Cantina team conducted a review of euler-swap on commit hash a8cf4966. The team identified a total of **15** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	3	0
Low Risk	6	6	0
Gas Optimizations	0	0	0
Informational	6	5	1
Total	**15**	**14**	**1**

3 Findings

3.1 Medium Risk

3.1.1 Unsafe Token Transfer

Severity: Medium Risk

Context: FundsLib.sol#L82

Description: In the `FundsLib` contract, the transfer of tokens to the protocol fee recipient is performed using the standard ERC-20 `transfer()` function:

```
IERC20(asset).transfer(p.protocolFeeRecipient, protocolFeeAmount);
```

This approach assumes compliance with the ERC-20 specification, including the return of a boolean success value. However, many tokens in the Ethereum ecosystem, such as USDT and others, do not strictly follow the ERC-20 standard and either do not return a value or behave inconsistently.

Recommendation: Replace the use of `IERC20(asset).transfer(...)` with OpenZeppelin's `SafeERC20.safeTransfer(...)`, which safely handles non-compliant tokens by suppressing return value decoding and inferring success from the absence of reverts. This wrapper ensures maximum compatibility across token implementations.

Euler: Fixed in PR 74.

Cantina Managed: Verified.

3.1.2 Incorrect Parameter Ordering in Curve Evaluation

Severity: Medium Risk

Context: QuoteLib.sol#L213

Description: In the `EulerSwap` swap logic, the call to `CurveLib.f` is made with an incorrect ordering of parameters during the evaluation of the post-swap state. The function `f()` is defined to compute a new reserve value based on the curve. However, the `EulerSwap` contract incorrectly inverts the order of `px` and `py` in some branches of its logic:

```diff
- yNew = CurveLib.f(xNew, py, px, y0, x0, cx);
+ yNew = CurveLib.f(xNew, px, py, x0, y0, cx);
```

This misordering distorts the price weight applied in the curve's internal calculations, leading to incorrect reserve updates and swap results that violate the intended AMM curve. Specifically, this causes inconsistencies in the swap symmetry: when executing an *exact-out* swap followed by an *exact-in* reversal using the same token pair and path, the system fails to return to the original state — demonstrating loss or unintended gain.

This error was clearly observed in simulated outputs. For instance, under the following parameter set:

```
priceX = 1
priceY = 2
x0 = 1000
y0 = 1000
cX = 0.8
cY = 0.8
reserve0 = 1000
reserve1 = 1000
amount = 500
asset0IsInput = False
exactIn = False
```

The uncorrected logic produced:

```
Starting reserves: (1000, 1000)
New Reserve: (500.00, 2200.00)
Output: 1200.00
```

Attempting to reverse this swap using exact input `1200` with `exactIn = True` should have brought the reserves back to `(1000,1000)`. Instead, the incorrect computation yielded:

```
New Reserve: (106.11, 2200.00)
Output: 893.89
```

This asymmetry stems from misaligned price parameters in the curve formula. After correcting the line the outputs aligned as expected. The output of the exact-out leg becomes:

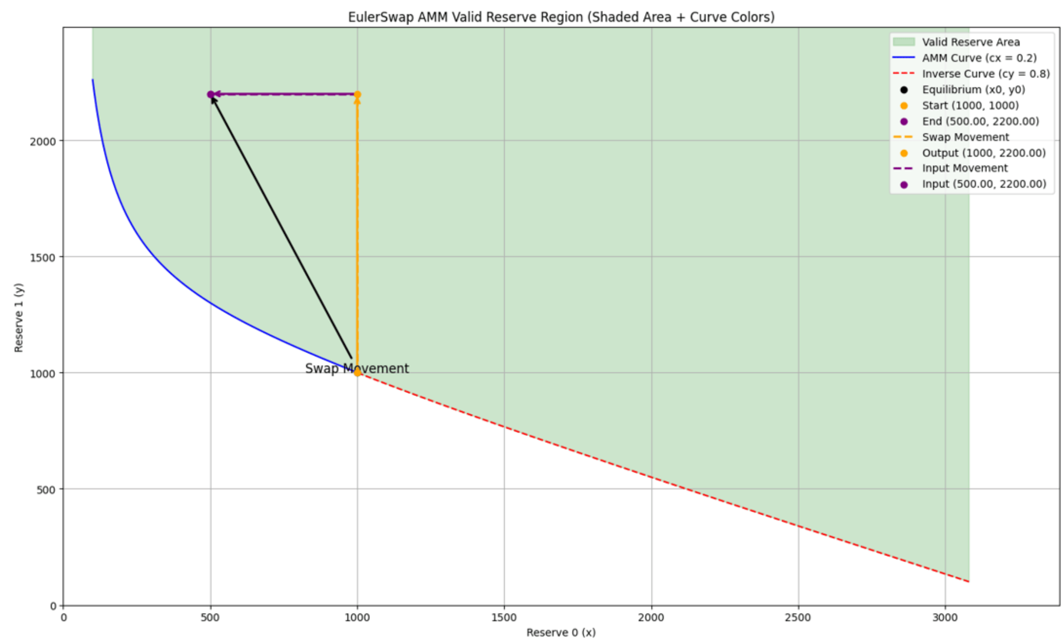
```
New Reserve: (500.00, 1300.50)
Output: 300.50
```

and the exact-in reversal with `amount = 300` restores the reserves to:

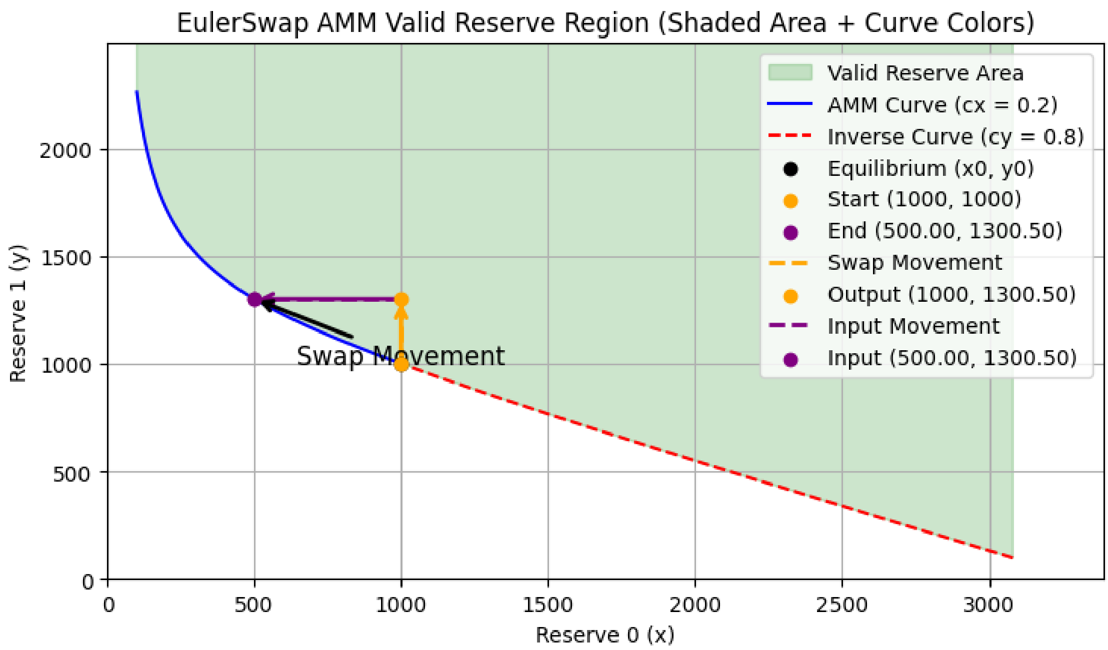
```
New Reserve: (500.00, 1300.00)
Output: 500.00
```

This confirms that the pricing logic and reserve transitions now obey the intended curve dynamics and yield symmetry across swap paths.

Visual validation using the provided plot script (see below) further illustrates the discrepancy. The invalid version of the logic plots a destination point far outside the feasible curve region:



Whereas the corrected computation yields a valid, curve-consistent transition:



Recommendation: All calls to `CurveLib.f()` and `CurveLib.fInverse()` must be carefully reviewed to ensure that parameters are passed in the correct order, specifically:

```diff
- yNew = CurveLib.f(xNew, py, px, y0, x0, cx);
+ yNew = CurveLib.f(xNew, px, py, x0, y0, cx);
```

In particular, care must be taken when switching between x-based and y-based flows to ensure that price tokens and reserve coordinates are not accidentally flipped. Consider isolating the curve application logic into well-named wrapper functions that enforce correct parameter order depending on swap direction, reducing the surface area for future human error.

Further, add test cases and simulation plots that validate the invertibility of swaps under various exact-in/exact-out scenarios across curve segments. Plotting the reserve trajectory with respect to the invariant is highly recommended for debugging and internal QA.

Euler: Fixed in PR 67.

Cantina Managed: Verified.

3.1.3 Removing pools with `swapAndPop()` in `EulerSwapFactory.uninstall()` corrupts stored indexes

Severity: Medium Risk

Context: EulerSwapFactory.sol#L152-L156, EulerSwapFactory.sol#L189-L192, EulerSwapFactory.sol#L178-L179

Description: When deploying a new pool, the index of the pool's address in the `allPools` and `poolMap` arrays are stored:

```
eulerAccountState[eulerAccount] = EulerAccountState({
    pool: newOperator,
    allPoolsIndex: uint48(allPools.length),
    poolMapIndex: uint48(poolMapArray.length)
});
```

When uninstalling a pool with `uninstall()`, `swapAndPop()` is used to remove pool addresses from the `allPools`/`poolMap` array:

```
function swapAndPop(address[] storage arr, uint256 index) internal {
    arr[index] = arr[arr.length - 1];
    arr.pop();
}
```

However, this causes the index stored in `eulerAccountState` for the last pool address to become incorrect. For example:

- Assume two pools are installed, so `allPools = [0x11, 0x22]`.
- For the second pool, `allPoolsIndex = 1`.
- The first pool is uninstalled, so `allPools = [0x22]`.
- Since `allPoolsIndex` is not updated, it is now invalid.

When the second pool is uninstalled, it will remove the wrong index in the `allPools` array. As a result, it will be impossible for accounts to uninstall pools. The following proof of concept demonstrates how the second pool cannot be uninstalled due to an OOB access:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {EulerSwapTestBase, IEulerSwap} from "test/EulerSwapTestBase.t.sol";

contract EulerSwapFactoryTest is EulerSwapTestBase {
    function test_multipleUninstalls() public {
        // Users
        address alice = makeAddr("alice");
        address bob = makeAddr("bob");

        // Parameters for deployPool()
        IEulerSwap.Params memory params = getEulerSwapParams(1e18, 1e18, 1e18, 1e18, 0, 0, 0, 0, address(0));
        IEulerSwap.InitialState memory initialState = IEulerSwap.InitialState(1e18, 1e18);
        bytes32 salt = bytes32(0);

        // Deploy pool for Alice
        params.eulerAccount = alice;
        address alicePool = eulerSwapFactory.computePoolAddress(params, salt);
        vm.startPrank(alice);
        evc.setAccountOperator(alice, alicePool, true);
        eulerSwapFactory.deployPool(params, initialState, salt);

        // Deploy pool for Bob
        params.eulerAccount = bob;
        address bobPool = eulerSwapFactory.computePoolAddress(params, salt);
        vm.startPrank(bob);
        evc.setAccountOperator(bob, bobPool, true);
        eulerSwapFactory.deployPool(params, initialState, salt);

        // Uninstall pool for Alice
        vm.startPrank(alice);
        evc.setAccountOperator(alice, alicePool, false);
        eulerSwapFactory.uninstallPool();

        // Uninstalling pool for Bob reverts due to an OOB access of the allPools array
        vm.startPrank(bob);
        evc.setAccountOperator(bob, bobPool, false);
        vm.expectRevert(stdError.indexOOBError);
        eulerSwapFactory.uninstallPool();
    }
}
```

Recommendation: Consider storing pools with OpenZeppelin's `EnumerableSet` instead of arrays. The pools can be removed from the set based on their address instead of index.

Euler: Fixed in PR 66.

Cantina Managed: Verified, the recommendation has been implemented.

3.2 Low Risk

3.2.1 Potential Re-Entrancy via `uniswapV2Call` Hook in `_beforeSwap`

Severity: Low Risk

Context: UniswapHook.sol#L68

Description: In the `UniswapHook` contract, the `_beforeSwap()` function is invoked as a pre-swap hook in the `EulerSwap` architecture. Within this function, important mutable state — specifically `s.reserve0` and `s.reserve1` — is accessed and potentially acted upon. Simultaneously, the main `EulerSwap.swap()` function does not follow a strict checks-effects-interactions pattern: it invokes the `uniswapV2Call` callback before updating the reserve values.

This ordering opens a subtle but dangerous surface for re-entrancy attacks via Uniswap-style flash loan callbacks. Since `s.reserve0` and `s.reserve1` are only written after the callback returns, a malicious contract could re-enter `swap()` from within `uniswapV2Call()` and trigger `_beforeSwap()` again with the stale reserves — possibly manipulating them further, creating inconsistencies, or even circumventing pricing invariants.

While no concrete exploit has been identified in the current implementation, this behavior creates an attack surface for reserve desynchronization or double-usage, especially when interacting with other composable DeFi primitives (e.g., nested swaps, lending hooks, callbacks with `delegatecall`, etc.).

Recommendation: Protect `_beforeSwap` by applying the existing `nonReentrantHook` modifier using `CtxLib.Storage` and the `s.status` field. This ensures the hook cannot be re-entered mid-swap and preserves reserve integrity.

Euler: Fixed in PR 77.

Cantina Managed: Verified. The re-entrancy lock was added.

3.2.2 Unrestricted Donations via Uniswap V4 Integration with `EulerSwap` Hook

Severity: Low Risk

Context: UniswapHook.sol#L157

Description: In the current design, `EulerSwap` acts as a Uniswap V4-style hook, but it does not override or restrict the `_beforeDonate` function. While these donations do **not** affect `EulerSwap`'s internal reserve tracking, they **do** affect the underlying Uniswap V4 pool contract, which receives and accounts for these tokens.

This introduces a mismatch: tokens can be donated directly to the Uniswap V4 pool contract during swap flows or hook invocations without passing through `EulerSwap`'s intended control path. These donations can cause offsets in Uniswap V4's internal balance tracking, leading to incorrect tick updates, mispriced liquidity ranges, and potentially broken accounting. Since `EulerSwap` does not mirror or react to these changes, the system ends up with inconsistent reserve views between the hook and the core pool.

The issue is made more dangerous by the flexibility of the V4 architecture — for instance, a flash swap callback or misconfigured hook integration could inadvertently or maliciously trigger a donation, causing state divergence that is not recoverable.

Recommendation: Override `_beforeDonate` in `EulerSwap` and revert unconditionally, matching the behavior of `_beforeAddLiquidity`. This ensures that any attempt to donate tokens directly to the pool while `EulerSwap` is the active hook is rejected, preserving alignment between the hook logic and the Uniswap V4 core pool state.

Euler: Fixed in PR 83.

Cantina Managed: Verified, the `beforeDonate` hook now reverts. This is implemented by setting `beforeDonate` in `getHookPermissions()` to `true`, which would revert when `BaseHook.beforeDonate()` is called.

3.2.3 Fee collection in `FundsLib.depositAssets()` reverts when the protocol fee recipient is the zero address

Severity: Low Risk

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

Context: FundsLib.sol#L78-L86.

Description: In `FundsLib.depositAssets()`, the protocol fee is calculated and sent to `p.protocolFeeRecipient` without checking if the recipient is the zero address:

```
{
    uint256 protocolFeeAmount = feeAmount * p.protocolFee / 1e18;

    if (protocolFeeAmount != 0) {
        IERC20(asset).transfer(p.protocolFeeRecipient, protocolFeeAmount);
        amount -= protocolFeeAmount;
        feeAmount -= protocolFeeAmount;
    }
}
```

As such, if the protocol fee recipient is ever configured to be the zero address and `protocolFeeAmount != 0`, a transfer to the zero address is performed, which reverts for many tokens. An example would be any token inheriting OpenZeppelin's `ERC20`, since it explicitly checks for this case. This would make it impossible to swap through the protocol as `FundsLib.depositAssets()` will always revert.

Recommendation: Consider only taking a fee when the protocol fee recipient is not the zero address:

```
- {
+ if (p.protocolFeeRecipient != address(0)) {
      uint256 protocolFeeAmount = feeAmount * p.protocolFee / 1e18;

      if (protocolFeeAmount != 0) {
          IERC20(asset).transfer(p.protocolFeeRecipient, protocolFeeAmount);
          amount -= protocolFeeAmount;
          feeAmount -= protocolFeeAmount;
      }
  }
```

Euler: Fixed in PR 76.

Cantina Managed: Verified, the recommendation has been implemented.

3.2.4 Checks in `EulerSwap.activate()` prevent deploying pools with a one-sided curve

Severity: Low Risk

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

Context: EulerSwap.sol#L87-L89.

Description: `EulerSwap.activate()` performs the following checks to ensure the pool's reserves are on the curve:

```
require(CurveLib.verify(p, s.reserve0, s.reserve1), CurveLib.CurveViolation());
require(!CurveLib.verify(p, s.reserve0 > 0 ? s.reserve0 - 1 : 0, s.reserve1), CurveLib.CurveViolation());
require(!CurveLib.verify(p, s.reserve0, s.reserve1 > 0 ? s.reserve1 - 1 : 0), CurveLib.CurveViolation());
```

However, these checks make it impossible to deploy a pool with either reserve as `0`, which is a valid input if `x0` or `y0` is also `0` (i.e. a one-sided curve). For example, if `x = 0` and `x0 = 0`, the first two checks would simplify to:

```
require(CurveLib.verify(p, s.reserve0, s.reserve1), CurveLib.CurveViolation());
require(!CurveLib.verify(p, 0, s.reserve1), CurveLib.CurveViolation());
```

Both checks are called with the same arguments as `reserve0 = 0`. However, assuming `y` and `y0` are valid values, `CurveLib.verify(p, reserve0, reserve1)` would pass but `!CurveLib.verify(p, 0, reserve1)` would fail.

The same problem exists if `f()` is called directly to check if the current reserves are on the curve, since `x = 0` is not a valid input for `f()`, but is technically on the curve since `x0 = 0`.

Recommendation: Consider modifying the checks to:

```
require(CurveLib.verify(p, s.reserve0, s.reserve1), CurveLib.CurveViolation());
if (s.reserve0 != 0) require(!CurveLib.verify(p, s.reserve0 - 1, s.reserve1), CurveLib.CurveViolation());
if (s.reserve1 != 0) require(!CurveLib.verify(p, s.reserve0, s.reserve1 - 1), CurveLib.CurveViolation());
```

Calling `f()` directly is avoided as `activate()` would need to determine which curve (i.e. `f(x)` or `f(y)`) to use, which would be re-implementing the logic in `verify()`.

Euler: Fixed in PR 75.

Cantina Managed: Verified, the recommendation has been implemented.

3.2.5 Division by zero occurs for `CurveLib.fInverse()` due to `c == 0` edge case

Severity: Low Risk

Context: CurveLib.sol#L85-L88, QuoteLib.sol#L181-L192

Description: In `CurveLib.fInverse()`, `x` is determined as follows when `B <= 0`:

```
if (B <= 0) {
    // use the regular quadratic formula solution (-b + sqrt(b^2 - 4ac)) / 2a
    x = Math.mulDiv(absB + sqrt, 1e18, 2 * c, Math.Rounding.Ceil) + 1;
} else {
```

However, there is an edge case here when `c = 0` as a division-by-zero occurs, causing `fInverse()` to revert. It occurs when `y = y0` and `x0 = 0`, which causes `B == 0`.

This edge case is reachable if the pool is deployed with a one-sided curve (i.e. `x0 = 0` or `y0 = 0`) and `c` for the other curve is `0`. For example, assume a pool is deployed with:

- `x0 = 0`, `y0 != 0`.
- `cx = 0`.

Consider a swap from `asset1` to `asset0` where the input amount is specified:

```
// swap Y in and X out
yNew = reserve1 + amount;
if (yNew < y0) {
    // remain on g()
    xNew = CurveLib.f(yNew, py, px, y0, x0, cy);
} else {
    // move to f()
    xNew = CurveLib.fInverse(yNew, px, py, x0, y0, cx);
}
```

If `yNew == y0`, `fInverse()` is called with `y == y0`, `x0 = 0` and `c = 0`, which fulfils all the listed conditions above for the edge case to occur.

Recommendation: There are two ways to handle this edge case:

1. In `QuoteLib.findCurvePoint()`, modify all the conditions that determine which curve to use to `x <= x0/y <= y0` (i.e. call `CurveLib.f()` when `x == x0/y == y0`). Alternatively, implement a short-circuit for `x == x0/y == y0` which returns `(x0, y0)`.
2. Handle the `c == 0` case in `CurveLib.fInverse()` by adding this branch to the top of the function:

    ```
    if (c == 0) {
        return Math.mulDiv(x0 * x0, px, x0 * px + py * (y - y0), Math.Rounding.Ceil);
    }
    ```

    This is equal to:

    $$\frac{x_0^2}{x_0 + \frac{p_y}{p_x}(y - y_0)}.$$

    Which is obtained by substituting $c_x = 0$ into equation 20 in the whitepaper and simplifying.

Euler: Fixed in PR 86.

Cantina Managed: Verified, `QuoteLib.findCurvePoint()` now calls `CurveLib.f()` when `x == x0/y == y0`.

3.2.6 Calculation of `term1` in `CurveLib.fInverse()` could overflow `int256.max` with extreme prices

Severity: Low Risk

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

Context:: CurveLib.sol#L55.

Description:: `CurveLib.fInverse()` performs the following calculation when calculating `B`:

```
int256 term1 = int256(Math.mulDiv(py * 1e18, y - y0, px, Math.Rounding.Ceil)); // scale: 1e36
```

However, it is possible for `term1` to exceed `uint256.max` (and `int256.max` by extension) within the given bounds when `px` is extremely small and `py` is extremely large.

For example:

- `y = uint112.max`.
- `px = 1e10`, `py = 1e36`.
- `y0 = 0`.

The price in this example would be `py / px = 1e26`, which extreme, but not entirely unreachable. For reference, WBTC / PEPE has a price of `1e21` (assuming WBTC = 100,000 USD and PEPE = 0.000001 USD).

Additionally, the following inputs result in `int256.max < term1 <= uint256.max`, causing the `int256` cast to overflow:

```
y: 4601578453167855735395833887649913
px: 23200824491
py: 758264634474574188751969252527909293
y0: 1302464407997906980507415513033026
```

As such, if a pool is deployed with an extreme price, functions involving quotes could unexpectedly revert since they call `CurveLib.fInverse()`.

Recommendation:: A suitable upper bound for `px` and `py` can be found by solving the following inequality:

```
(py * 1e18) * (y - y0) / px <= int256.max
```

Assuming the worst-case of `y - y0 = uint112.max`, the inequality simplifies to:

```
py / px <= int256.max / uint112.max / 1e18
```

Which is roughly equal to `1.1e25`. As such, consider restricting the upper bound of `px` and `py` to `1e25` to ensure `py / px` (or `px / py`) does not exceed `1.1e25`.

While this reduces the maximum price which can be configured, it is still a reasonable price range. Realistically, the largest possible price range is GUSD / PEPE, which has a price of:

```
price of 1 wei of GUSD / price of 1 wei of PEPE = (1 / 1e2) / (0.000001 / 1e18) = 1e22
```

Euler: Fixed in PR 85.

Cantina Managed: Verified, `px` and `py` are now checked to not be greater than `1e25` in `EulerSwap.activate()`.

3.3 Informational

3.3.1 Missing Internal Function Naming Convention

Severity: Informational

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

Description: The `EulerSwapFactory` contract and other related contracts (such as `EulerSwap`, `EulerSwapPeriphery`, and `MetaProxyDeployer`) do not consistently follow a naming convention to clearly distinguish between public/external and internal/private functions. Specifically, internal helper functions like `uninstall`, `swapAndPop`, and similarly scoped utility functions are defined without a leading underscore (_), which makes it difficult to differentiate at a glance between callable external APIs and internal logic. This is especially problematic in contracts with complex state transitions or logic branching, where traceability of function calls plays a critical role in auditing and maintaining correctness.

This inconsistent naming can cause confusion during review and debugging phases, and it opens the possibility for accidental misuse or misidentification of function purpose. It also breaks with widely adopted

Solidity conventions, which favor using a prefixed underscore (_) for non-external functions that should not be exposed or used directly by consumers of the contract.

Recommendation: It is recommended to refactor all internal and private functions across `EulerSwapFactory` and associated contracts (`EulerSwap`, `FundsLib`, `MetaProxyDeployer`, etc...) to include a leading underscore (_) in their names. For example, `swapAndPop` should be renamed to `_swapAndPop`, and similarly for `uninstall` and other auxiliary logic handlers. This will improve code readability, clarify intent, and align the codebase with Solidity best practices.

If the team prefers to maintain the current naming scheme for legacy or readability reasons, an acceptable alternative is to clearly document the visibility and intended usage of such functions with NatSpec annotations and visibility modifiers, though this is still considered a weaker practice than using naming convention alone.

Euler: Pending remediation.

Cantina Managed: Pending remediation.

3.3.2 Incompatibility with `uniswapV2Call` Interface Expectations

Severity: Informational

Context: EulerSwap.sol#L163

Description: The `EulerSwap` contract is designed to behave similarly to a Uniswap V2 pair, but it does not implement all interface expectations of the UniswapV2 ecosystem. Specifically, it omits the `token0()` and `token1()` public getters, which are critical for integration with external contracts using the `uniswapV2Call()` flash swap pattern.

As described in the Uniswap V2 documentation, integrators typically write callback handlers under the assumption that:

```
function uniswapV2Call(address sender, uint amount0, uint amount1, bytes calldata data) external {
    address token0 = IUniswapV2Pair(msg.sender).token0();
    address token1 = IUniswapV2Pair(msg.sender).token1();
    assert(msg.sender == IUniswapV2Factory(factoryV2).getPair(token0, token1));
    ...
}
```

This pattern assumes that:

1. The pair contract exposes `token0()` and `token1()` as public view functions.
2. The caller (`msg.sender`) is a contract created by a known `UniswapV2Factory`, registered via `getPair(token0, token1)`.

The issue is that `EulerSwap` pairs are not deployed or tracked by a canonical UniswapV2Factory instance. Therefore, any standard check like:

```
assert(msg.sender == IUniswapV2Factory(factoryV2).getPair(token0, token1));
```

will fail, unless integrators explicitly whitelist `EulerSwap` addresses or bypass the factory check entirely. Even though `EulerSwapFactory` exists, it is **not** designed to conform to `UniswapV2Factory`'s interface, and integrators will continue targeting Uniswap's version.

Moreover, without `token0()` and `token1()` getters, the very first step in `uniswapV2Call` reverts, as `msg.sender` does not expose these functions. This breaks compatibility with flash swap routers, arbitrage bots, and legacy DeFi integrations that assume drop-in V2 pair behavior.

Recommendation: If compatibility with existing `uniswapV2Call`-based contracts is a goal (e.g., allowing integrators to plug EulerSwap into existing arbitrage or flash swap flows), the following steps should be taken:

1. Add `token0()` and `token1()` public view functions to the `EulerSwap` pair contracts. These should return the respective token addresses in fixed canonical order. This change alone will enable basic integration and unlock compatibility with a broad range of `uniswapV2Call` consumers.
2. Document factory expectations clearly. While `EulerSwapFactory` is not meant to mimic `UniswapV2Factory`, it's important to communicate to integrators that the factory validation step

(i.e., using `getPair(token0, token1)`) must be either removed or replaced by a manual whitelisting process.

3. Consider renaming the callback (e.g., `eulerSwapCall()`) or explicitly documenting that while the mechanism is similar to UniswapV2, it is **not** drop-in compatible unless these caveats are handled.

If compatibility is *not* a design goal, the documentation should strongly emphasize that integrators **must not** assume this contract adheres to the V2 pair interface, and should instead follow custom integration guidelines.

Euler: Fixed in PR 80.

Cantina Managed: Verified, the callback has been renamed to `IEulerSwapCallee.eulerSwapCall()`.

3.3.3 `maxWithdraw` in `QuoteLib.calcLimits()` double-counts deposited assets

Severity: Informational

Context: QuoteLib.sol#L98-L103.

Description: In `QuoteLib.calcLimits()`, `outLimit` is restricted by the remaining cash and borrow caps in the vault as such:

```
(, uint16 borrowCap) = vault.caps();
uint256 maxWithdraw = decodeCap(uint256(borrowCap));
maxWithdraw = vault.totalBorrows() > maxWithdraw ? 0 : maxWithdraw - vault.totalBorrows();
if (maxWithdraw > cash) maxWithdraw = cash;
maxWithdraw += vault.convertToAssets(vault.balanceOf(eulerAccount));
if (maxWithdraw < outLimit) outLimit = maxWithdraw;
```

However, the following line:

```
if (maxWithdraw > cash) maxWithdraw = cash;
```

should be removed for two reasons:

1. Assets deposited by `eulerAccount` are part of `cash`, so `maxWithdraw` is double-counting deposited assets.

2. The case where `maxWithdraw > cash` is implicitly covered by the `cash < outLimit` check above.

For example, assume the following numbers:

- `cash = 100`.
- `borrowCap - totalBorrows = 120`.
- `depositedAssets = vault.convertToAssets(vault.balanceOf(eulerAccount)) = 20`.

For point (1), `maxWithdraw` is calculated as:

```
min(borrowCap - totalBorrows, cash) + depositedAssets = min(120, 100) + 20 = 120
```

`120` is clearly wrong as the vault only has a total of `100` assets that can be withdrawn/borrowed.

Due to point (2), the inflated `maxWithdraw` does not cause any issues as `outLimit = cash = 100` from the `cash < outLimit` check above. In general, `borrowCap - totalBorrows > cash` implies `maxWithdraw > cash`, so it will always be covered by the `cash < outLimit` check.

Recommendation: The `maxWithdraw > cash` check prevents `maxWithdraw + depositedAssets` from overflowing, since `maxWithdraw` could be up to `uint256.max`. Consider short-circuiting if the intermediate `maxWithdraw` is greater than `outLimit`:

```
- if (maxWithdraw > cash) maxWithdraw = cash;
- maxWithdraw += vault.convertToAssets(vault.balanceOf(eulerAccount));
- if (maxWithdraw < outLimit) outLimit = maxWithdraw;
+ if (maxWithdraw < outLimit) {
+     maxWithdraw += vault.convertToAssets(vault.balanceOf(eulerAccount));
+     if (maxWithdraw < outLimit) outLimit = maxWithdraw;
+ }
```

This implementation should save some gas since further calculations are skipped when `maxWithdraw > outLimit` can be determined early.

Euler: Fixed in PR 81.

Cantina Managed: Verified, the recommendation has been implemented.

3.3.4 Minor improvements to code and comments

Severity: Informational

Context: *(See each case below)*

Description/Recommendation:

1. CurveLib.sol#L71-L72, CurveLib.sol#L79-L80 - Consider using OZ's `sqrt()` with rounding to round up. For example:

```
- sqrt = Math.sqrt(discriminant); // drop back to 1e18 scale
- sqrt = (sqrt * sqrt < discriminant) ? sqrt + 1 : sqrt;
+ sqrt = Math.sqrt(discriminant, Math.Rounding.Ceil);
```

2. CurveLib.sol#L38, CurveLib.sol#L90 - Consider using `Math.ceilDiv()` for rounding up instead, it improves readability and avoids the case where `n + (n - 1)` overflows `uint256.max`.

```
- return y0 + (v + (py - 1)) / py;
+ return y0 + Math.ceilDiv(v, py);
```

```
- x = (2 * C + (absB + sqrt - 1)) / (absB + sqrt) + 1;
+ x = Math.ceilDiv(2 * C, absB + sqrt) + 1;
```

3. CurveLib.sol#L71 - Typo, double comment slashes.
4. CurveLib.sol#L58 - Unncessary brackets can be removed:

```
- C = Math.mulDiv((1e18 - c), x0 * x0, 1e18, Math.Rounding.Ceil); // scale: 1e36
+ C = Math.mulDiv(1e18 - c, x0 * x0, 1e18, Math.Rounding.Ceil); // scale: 1e36
```

5. UniswapHook.sol#L82-L88 - The code here can be simplified:

```
  if (isExactInput) {
      amountIn = uint256(-params.amountSpecified);
-     amountOut = QuoteLib.computeQuote(evc, p, params.zeroForOne, uint256(-params.amountSpecified),
↪   true);
+     amountOut = QuoteLib.computeQuote(evc, p, params.zeroForOne, amountIn, true);
  } else {
-     amountIn = QuoteLib.computeQuote(evc, p, params.zeroForOne, uint256(params.amountSpecified),
↪   false);
      amountOut = uint256(params.amountSpecified);
+     amountIn = QuoteLib.computeQuote(evc, p, params.zeroForOne, amountOut, false);
  }
```

6. UniswapHook.sol#L119 - This check is not needed as `CurveLib.verify()` has the same check. Consider removing it.
7. EulerSwapFactory.sol#L63-L67 - `params.eulerAccount` does not need to be included in the salt as it in the creation code as part of `params`. Therefore, a different `eulerAccount` would result in a different pool address.
8. EulerSwapFactory.sol#L94 - `poolParams.eulerAccount` doesn't need to be cast to an address as it is already one:

```
- keccak256(abi.encode(address(poolParams.eulerAccount), salt)),
+ keccak256(abi.encode(poolParams.eulerAccount, salt)),
```

9. CtxLib.sol#L24-L40 - The code here can be simplified to avoid using assembly:

```
return abi.decode(msg.data[msg.data.length - 384:], (IEulerSwap.Params));
```

10. EulerSwap.sol#L61 - `activate()` should be declared external.

Euler: Fixed in the following PRs:

1. PR 78.
2. PR 84.
3. PR 70.
4. PR 70.
5. PR 82.
6. PR 82.
7. PR 71.
8. PR 70.
9. PR 69.
10. PR 70.

Cantina Managed: Verified.

3.3.5 `UniswapHook._beforeInitialize()` is never reached during normal initialization

Severity: Informational

Context: UniswapHook.sol#L129-L136, Hooks.sol#L170-L175

Description: `UniswapHook._beforeInitialize()` is overridden to check that `_poolKey.tickSpacing` has not been set. This is meant to ensure the pool can only be initialized through `UniswapHook.activateHook()`:

```
function _beforeInitialize(address, PoolKey calldata, uint160) internal view override returns (bytes4) {
    // when the hook is deployed for the first time, the internal _poolKey is empty
    // upon activation, the internal _poolKey is initialized and set
    // once the hook contract is activated, do not allow subsequent initializations
    require(_poolKey.tickSpacing == 0, AlreadyInitialized());
    return BaseHook.beforeInitialize.selector;
}
```

However, Uniswap V4 calls hooks with the `noSelfCall` modifier, which skips calling the hook if the caller is the hook address itself:

```
/// @notice modifier to prevent calling a hook if they initiated the action
modifier noSelfCall(IHooks self) {
    if (msg.sender != address(self)) {
        _;
    }
}
```

Therefore, when `poolManager.initialize()` is called in `activateHook()`, the `beforeInitialize` hook is actually never called and `_beforeInitialize()` is never reached. In fact, checking `_poolKey.tickSpacing == 0` is incorrect as `_poolKey` is set before calling `poolManager.initialize()` in `activateHook()`. This behavior can be observed by replacing the require statement with `revert AlreadyInitialized()` and running the tests; they will all still pass.

Recommendation: Consider removing `_beforeInitialize()` entirely. Initializations outside of `activateHook()` would revert with `HookNotImplemented` in `BaseHook`, while normal initialization works as intended. `_beforeAddLiquidity()` can also be removed, since the implementation in `BaseHook` also reverts. Alternatively, remove the check and simply revert with `AlreadyInitialized`.

Euler: Fixed in PR 83.

Cantina Managed: Verified, `_beforeInitialize()` and `_beforeAddLiquidity()` have been removed.

3.3.6 Additional fuzz tests for `CurveLib`

Severity: Informational

Context: CurveLib.sol.

Description: The following tests may be helpful for future changes or fixes:

1. Fuzz tests for `f()` and `fInverse()` to check for reverts/overflows.
2. Differential tests for `f()` and `fInverse()` against the equations from the whitepaper translated into python.

Some things to note for (2):

- The tests require enabling ffi in your Foundry config.
- The tests skip any inputs where `f()`/`fInverse()` revert, or the case where casting `term1` to `int256` overflows in `fInverse()`.
- `y` is bounded to `[y0, uint112.max]` for `fInverse()` although it isn't listed in the pre-conditions.

Euler: Improved NatSpec for `[y0, uint112.max]` for `fInverse()` in PR 85 and PR 87.

Cantina Managed: Fix verified.