CANTINA

Euler Swap
Security Review

Cantina Managed review by:
Christoph Michel, Lead Security Researcher
Mario.eth, Lead Security Researcher

March 29, 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 Mar 3rd to Mar 10th the Cantina team conducted a review of euler-swap on commit hash 54c1af6d. The team identified a total of **11** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 5
- Gas Optimizations: 1
- Informational: 5

3 Findings

3.1 Low Risk

3.1.1 Reserves and `eulerAccount` CDP can drift apart

Severity: Low Risk

Context: EulerSwap.sol#L24-L31

Description: At deployment time, the deployer defines the curve and sets the initial reserve point on it, defining the max swap sizes the AMM will support over its lifetime (the curve does not "grow" as no fees are reinvested to scale the curve). At the same time, the AMM's `eulerAccount` holds the initial collateral position(s). Note that a swap changes the collateralized-debt position of the `eulerAccount`.

Combined, these define an implicit initial max LTV for the AMM that can be attained by a single swap.

> For example, if the initial collateral position is $1000 in `vault0`, if the used curve is close to constant-sum and if the initial reserves are set to (6000, 5000), then the maximum LTV loan that the AMM will support will be `5000/6000 = 0.8333`. In order to leave a safety buffer, it is recommended to choose a maximum LTV that is below the borrowing LTV of the vault.

Notice that the net asset value (collateral value minus debt), and therefore the health factor, can change for several reasons:

1. Interest is accrued to collateral shares (increases NAV).
2. Debt is accrued to the borrow position (decreases NAV).
3. The position is (partially) liquidated (decreases NAV).
4. Swap fees (increases NAV).
5. The "slippage" earned. The NAV increases by `amountInValue - amountOutValue` (after fees). This depends on the curve parameters. (increases/decreases NAV).
6. The user account holding the position (`eulerAccount`) performs an action that changes the NAV (`withdraw`,`deposit`,`borrow`,`repay`).
7. Price changes for collateral or debt token.

While the AMM keeps "*its liquidity*" constant throughout its lifetime (the curve never changes), the NAV and therefore the health factor of the underlying `eulerAccount` can change and get out of sync - relative to the initial configuration. This leads to the following issues:

1. The AMM can fail to serve all swap sizes that it could serve previously. Swaps fail as the max swap output amount is overestimated, meaning, it's greater than the max amount the `eulerAccount` can still borrow while being healthy.
2. It enables swaps that end up in a CDP that goes **above the initial max LTV**. An attacker can swap past the max LTV, right up to the position becoming liquidatable in the next block. In the next block, the attacker liquidates the AMM for profit.

Example: Let C_i define the collateral asset value deposited into vault_i and D_i its debt value. Let R_i be the reserve for vault i's asset (corresponding to the `EulerSwap.reserve0/1` state variables). Given the example above with its initial configuration of $1000 in `vault0`, initial reserves of `(6000, 5000)`, the EulerCurve configured to be close to *constant-sum*, yields an implicit `maxLTV = 83.33%`.

1. A user performs a swap, and a CDP is created.
2. Borrow interest accrues, increasing the debt **but the reserves do not change**.
3. A user performs another swap and the contract ends up with an LTV of 89.29%, higher than the desired max LTV. The CDP could be close to liquidation at this point.

After action	R_0	C_0	D_0	R_1	C_1	D_1	LTV
deployment	6000	1000	0	5000	0	0	0
$\text{swap}_{0\to1}$ 2000	8000	3000	0	3000	0	2000	$\frac{2}{3}$
$400 debt accrues	8000	3000	0	3000	0	2400	80%

After action	R_0	C_0	D_0	R_1	C_1	D_1	LTV
$\text{swap}_{1\to0}$ 8000	0	0	5000	11000	5600	0	$\frac{5000}{5600} = 89.29\%$

Recommendation: LPs need to constantly be aware of their worst-case LTV that can be achieved by a swap in the current pool. Otherwise, they risk liquidation in addition to an AMM that can't serve its expected range anymore. They can manage this by repaying debt, adding new collateral, or simply migrating liquidity to a new pool with a (smaller) reserve configuration.

Alternatively, the AMM could dynamically scale the curve and adjust its reserves based on the current CDP. Special attention must be paid to ensure that any dynamic curve adjustment is not exploitable.

Euler: We acknowledge this risk and consider it by design. EulerSwap instances must be monitored over time and reinstalled if the drift becomes significant.

Spearbit: Acknowledged.

3.1.2 Swap limits returned by `EulerSwapPeriphery.sol` ignore expected health and hook states

Severity: Low Risk

Context: EulerSwapPeriphery.sol#L60

Description: Swaps can revert if the swap results in a collateral & borrow position of the `eulerAccount` that is unhealthy. The `EulerSwapPeriphery.calcLimits` does not take the `eulerAccount`'s health into account and can return limits (and quotes) that end up reverting.

Similarly, hooks on `deposit`, `withdraw`, etc... are ignored.

Recommendation: Restricting swap limits to the expected health range is complex and might not be necessary under normal circumstances as a swap should never be able to put the `eulerAccount` into health violation - assuming the NAV and reserves are chosen properly upon AMM deployment and any desync is monitored. Consider documenting this behavior for `getLimits` and `quote*`.

Euler: We acknowledge this. It is by design, since it would be too difficult to prevent in the general case. Operators should ensure their pools remain liquid, and not use vaults with problematic hooks installed, otherwise they may not receive order flow.

Spearbit: Acknowledged.

3.1.3 `quoteExactInput`/`quoteExactOutput` returns quotes that revert if fee pushes vault supply past cap

Severity: Low Risk

Context: EulerSwapPeriphery.sol#L111-L128

Description: The `EulerSwapPeriphery.computeQuote` function checks if the input amount is within the max-allowed swap input size (`inLimit`):

```
// exactIn: decrease received amountIn, rounding down
if (exactIn) amount = amount * feeMultiplier / 1e18;

bool asset0IsInput = checkTokens(eulerSwap, tokenIn, tokenOut);
(uint256 inLimit, uint256 outLimit) = calcLimits(eulerSwap, asset0IsInput);

uint256 quote = binarySearch(eulerSwap, reserve0, reserve1, amount, exactIn, asset0IsInput);

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: increase required quote(amountIn), rounding up
if (!exactIn) quote = (quote * 1e18 + (feeMultiplier - 1)) / feeMultiplier;
```

However, the input amount needs to be **inclusive of fees** as the `inLimit` might be restricted because of supply caps. The function can return wrong quotes that revert when executed.

Recommendation: For `exactIn`, the input amount (`amount`) check needs to be performed *before* it is reduced in L112. For `exactOut`, the input amount (`quote`) check needs to be performed *after* the fee is added to it in L128.

Proof of Concept: This test file contains several issues regarding limits. See `testBug_computeQuote_supplyCap_exactIn` and `testBug_computeQuote_supplyCap_exactOut`:

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

import {IEVault, IEulerSwap, EulerSwapTestBase, EulerSwap, EulerSwapPeriphery, TestERC20} from
↪  "./EulerSwapTestBase.t.sol";
import {console} from "forge-std/Test.sol";
import {Errors as EVKErrors} from "evk/EVault/shared/Errors.sol";

contract CantinaLimits is EulerSwapTestBase {
    EulerSwap public eulerSwap;

    error CurveViolation();
    error DepositFailure(bytes reason);

    function setUp() public virtual override {
        // EulerSwap creator is `holder`
        // `holder` deposited 10e18 in both vaults
        // `depositor` deposited 100e18 in both vaults
        super.setUp();
    }

    // reverts because reserves will be more than uint112.max given inLimit
    function testBug_limits_reserve() public {
        eulerSwap = createEulerSwap(120e18, 120e18, 0, 1e18, 1e18, 0.4e18, 0.85e18);

        (uint256 inLimit,) =
          periphery.getLimits(address(eulerSwap), address(assetTST), address(assetTST2));

        // assertEq(inLimit, type(uint112).max - 120e18);
        // assertEq(outLimit, 120e18);

        uint256 amountIn = inLimit;
        // revert is correct but as we use the value of getLimits, this revert is unexpected
        vm.expectRevert(EulerSwapPeriphery.SwapLimitExceeded.selector);
        uint256 amountOut =
            periphery.quoteExactInput(address(eulerSwap), address(assetTST), address(assetTST2), amountIn);

        assetTST.mint(address(this), amountIn);
        assetTST.transfer(address(eulerSwap), amountIn);
        // revert is correct but as we use the value of getLimits, this revert is unexpected
        vm.expectRevert(CurveViolation.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }

    function testBug_computeQuote_supplyCap_exactIn() public {
        // 10% fee
        eulerSwap = createEulerSwap(60e18, 60e18, 0.10e18, 1e18, 1e18, 0.4e18, 0.85e18);

        // vault1's supply cap is 120e18
        eTST.setCaps(uint16(120 << 6) | (18 + 2), 0);

        // 10e18 needed to hit the supply cap. depositing 1 more wei should revert quoteExactInput
        uint256 amountIn = 10e18 + 1;
        // SHOULD REVERT WITH SwapLimitExceeded
        uint256 amountOut =
            periphery.quoteExactInput(address(eulerSwap), address(assetTST), address(assetTST2), amountIn);
        assetTST.mint(address(this), amountIn);

        assetTST.transfer(address(eulerSwap), amountIn);
        vm.expectRevert(EVKErrors.E_SupplyCapExceeded.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }

    function testBug_computeQuote_supplyCap_exactOut() public {
        // 10% fee
        eulerSwap = createEulerSwap(60e18, 60e18, 0.10e18, 1e18, 1e18, 0.4e18, 0.85e18);
```

```
        // vault1's supply cap is 120e18
        eTST.setCaps(uint16(120 << 6) | (18 + 2), 0);

        uint256 amountOut = 8774547569435388591;
        // SHOULD REVERT WITH SwapLimitExceeded
        uint256 amountIn =
            periphery.quoteExactOutput(address(eulerSwap), address(assetTST), address(assetTST2), amountOut);
        // 10e18 needed to hit the supply cap. depositing 1 more wei should revert quoteExactInput
        assertEq(amountIn, 10e18 + 2);

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

        assetTST.transfer(address(eulerSwap), amountIn);
        vm.expectRevert(EVKErrors.E_SupplyCapExceeded.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }
}
```

Euler: Acknowledged, but not fixed as of this time. However, this one is important even if we don't attempt a strict `getLimits()` and I plan to resume working on it soon.

Spearbit: Acknowledged by the client.

3.1.4 `inLimit` does not respect the `reserve uint112` restriction

Severity: Low Risk

Context: EulerSwapPeriphery.sol#L211

Description: The `EulerSwap` reserves need to fit into `uint112`. Therefore, the maximum swap amount that can be traded in may not push the reserves over this limit.

The impact is that `getLimits` and `computeQuote` functions return invalid values that make the swap revert.

Recommendation: Consider initializing `inLimit` to `type(uint112).max - currentReserveIn`.

Proof of Concept:

This test file contains several issues regarding limits. See `testBug_limits_reserve`.


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

import {IEVault, IEulerSwap, EulerSwapTestBase, EulerSwap, EulerSwapPeriphery, TestERC20} from
↪  "./EulerSwapTestBase.t.sol";
import {console} from "forge-std/Test.sol";
import {Errors as EVKErrors} from "evk/EVault/shared/Errors.sol";

contract CantinaLimits is EulerSwapTestBase {
    EulerSwap public eulerSwap;

    error CurveViolation();
    error DepositFailure(bytes reason);

    function setUp() public virtual override {
        // EulerSwap creator is `holder`
        // `holder` deposited 10e18 in both vaults
        // `depositor` deposited 100e18 in both vaults
        super.setUp();
    }

    // reverts because reserves will be more than uint112.max given inLimit
    function testBug_limits_reserve() public {
        eulerSwap = createEulerSwap(120e18, 120e18, 0, 1e18, 1e18, 0.4e18, 0.85e18);

        (uint256 inLimit,) =
          periphery.getLimits(address(eulerSwap), address(assetTST), address(assetTST2));

        // assertEq(inLimit, type(uint112).max - 120e18);
        // assertEq(outLimit, 120e18);

        uint256 amountIn = inLimit;
        // revert is correct but as we use the value of getLimits, this revert is unexpected
```

```
        vm.expectRevert(EulerSwapPeriphery.SwapLimitExceeded.selector);
        uint256 amountOut =
            periphery.quoteExactInput(address(eulerSwap), address(assetTST), address(assetTST2), amountIn);

        assetTST.mint(address(this), amountIn);
        assetTST.transfer(address(eulerSwap), amountIn);
        // revert is correct but as we use the value of getLimits, this revert is unexpected
        vm.expectRevert(CurveViolation.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }

    function testBug_computeQuote_supplyCap_exactIn() public {
        // 10% fee
        eulerSwap = createEulerSwap(60e18, 60e18, 0.10e18, 1e18, 1e18, 0.4e18, 0.85e18);

        // vault1's supply cap is 120e18
        eTST.setCaps(uint16(120 << 6) | (18 + 2), 0);

        // 10e18 needed to hit the supply cap. depositing 1 more wei should revert quoteExactInput
        uint256 amountIn = 10e18 + 1;
        // SHOULD REVERT WITH SwapLimitExceeded
        uint256 amountOut =
            periphery.quoteExactInput(address(eulerSwap), address(assetTST), address(assetTST2), amountIn);
        assetTST.mint(address(this), amountIn);

        assetTST.transfer(address(eulerSwap), amountIn);
        vm.expectRevert(EVKErrors.E_SupplyCapExceeded.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }

    function testBug_computeQuote_supplyCap_exactOut() public {
        // 10% fee
        eulerSwap = createEulerSwap(60e18, 60e18, 0.10e18, 1e18, 1e18, 0.4e18, 0.85e18);

        // vault1's supply cap is 120e18
        eTST.setCaps(uint16(120 << 6) | (18 + 2), 0);

        uint256 amountOut = 8774547569435388591;
        // SHOULD REVERT WITH SwapLimitExceeded
        uint256 amountIn =
            periphery.quoteExactOutput(address(eulerSwap), address(assetTST), address(assetTST2), amountOut);
        // 10e18 needed to hit the supply cap. depositing 1 more wei should revert quoteExactInput
        assertEq(amountIn, 10e18 + 2);

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

        assetTST.transfer(address(eulerSwap), amountIn);
        vm.expectRevert(EVKErrors.E_SupplyCapExceeded.selector);
        eulerSwap.swap(0, amountOut, address(this), "");
    }
}
```


Euler: We acknowledge this issue and the related issue "`quoteExactInput/quoteExactOutput` returns quotes that revert if fee pushes vault supply past cap", but at this time have not comprehensively fixed them.

Currently `getLimits()` returns an upper-bound on the swap limits: In other words, any amounts beyond the returned values will for sure fail, but lesser amounts may or may not succeed. Making `getLimits()` precise is a bit more involved than I expected at first, since it requires finding these limits and then reducing them such that the corresponding implied amounts are also on the curve (and within the `uint112` limit of course).

Right now we're trying to figure out the actual use-case for an external `getLimits()` to determine if it is necessary to solve this precisely. I imagine this is mostly an optimisation to filter out obviously unsuitable pools, since a solver would need to actually perform a quote anyway to determine price impact etc, and would be able to observe a failure at that stage.

Spearbit: Acknowledged.

3.1.5 Non-strict monotonicity of `f` allows stealing dust for free

Severity: Low Risk

Context: EulerSwap.sol#L270-L278

Description: As the curve function `f` needs to be implemented with integer arithmetic in Solidity, it loses its strict monotonicity, and one can find values `x` and `x + dx` with `f(x) == f(x + dx)`. From the current reserve point `(x + dx, y)` one can always steal `dx` if `(x, y)` is also a valid point (similarly, one can steal `dy`). A swapper performs an `exactIn` swap that ends up with a new input reserve value `x` that is not optimal for the amount `y` they receive (meaning there is a `dx > 0  s.t. f(x - dx) = y`), the swapper is overpaying slightly.

Attached are fuzz tests that search for the `max dx` that can be stolen given curve parameters. With reasonable curve parameters for an 18-decimal and 6-decimal token pair, we could find that the most `token0` that can be stolen is around `1e12` tokens.

Recommendation: Before deployment, ensure that the value of the tokens that can be swapped out of the AMM for free is negligible.

Proof of Concept:

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

import {IEVault, IEulerSwap, EulerSwapTestBase, EulerSwap, EulerSwapPeriphery, TestERC20} from
↪  "./EulerSwapTestBase.t.sol";
import {console} from "forge-std/Test.sol";
import {Errors as EVKErrors} from "evk/EVault/shared/Errors.sol";
import {Math} from "openzeppelin-contracts/utils/math/Math.sol";

contract CantinaCurve is EulerSwapTestBase {
    EulerSwap public eulerSwap;

    error Overflow();

    function setUp() public virtual override {
        super.setUp();
    }

    // this test doesn't say too much as some curve parameters are unreasonable
    // the dust that can be stolen can become very large for some curve parameters.
    function test_f_monotonicity_any(uint256 xt, uint256 px, uint256 py, uint256 x0, uint256 y0, uint256 c)
    ↪  public {
        // increase dx and if the fuzzer finds a contradiction it means
        // there's an x s.t. f(x) == f(x+dx)
        // (sometimes errors in the Math.mulDiv with overflow if x0 is high and x is tiny)
        uint256 dx = 1e24;
        x0 = bound(x0, 1 + dx, type(uint112).max);
        y0 = bound(y0, 0, type(uint112).max);
        uint256 lower = 1 + x0 / 1e3;
        uint256 upper = Math.max(x0 - dx, lower);
        xt = bound(xt, lower, upper);
        px = bound(px, 1, 1e36);
        py = bound(py, 1, 1e36);
        c = bound(c, 1, 1e18);

        console.log("x0  : %e", x0);
        console.log("y0  : %e", y0);
        console.log("px  : %e", px);
        console.log("py  : %e", py);
        console.log("c   : %e", c);
        console.log("x   : %e", xt);

        uint256 f_x = f(xt, px, py, x0, y0, c);
        console.log("f(x): %e", f_x);
        uint256 f_xdx = f(xt + dx, px, py, x0, y0, c);
        assertNotEq(f_x, f_xdx, "found f(x) == f(x+dx)");
    }

    function test_f_monotonicity_reasonable_curve(uint256 xt) public {
        // increase dx and if the fuzzer finds a contradiction it means
        // there's an x s.t. f(x) == f(x+dx)
        uint256 dx = 1e12 - 100;
        uint256 x0 = 1e9 * 1e18;
        uint256 y0 = 1e9 * 1e6;
        uint256 px = 1e6;
        uint256 py = 1e18;
        uint256 c = 0.9e18;
```

```
        uint256 lower = 1;
        uint256 upper = Math.max(x0 - dx, lower);
        xt = bound(xt, lower, upper);

        console.log("x0  : %e", x0);
        console.log("y0  : %e", y0);
        console.log("px  : %e", px);
        console.log("py  : %e", py);
        console.log("c   : %e", c);
        console.log("x   : %e", xt);

        uint256 f_x = f(xt, px, py, x0, y0, c);
        console.log("f(x): %e", f_x);
        uint256 f_xdx = f(xt + dx, px, py, x0, y0, c);
        assertNotEq(f_x, f_xdx, "found f(x) == f(x+dx)");
    }

    function test_swap_steal() public {
        uint256 dx = 1e12 - 100;
        uint256 x0 = 1e9 * 1e18;
        uint256 y0 = 1e9 * 1e6;
        uint256 px = 1e6;
        uint256 py = 1e18;
        uint256 c = 0.9e18;
        uint256 xt = 9.999999999999990000000000098e26;

        eulerSwap = createEulerSwap(uint112(x0), uint112(y0), 0, px, py, c, c);

        // swap to xt+dx reserves naturally. ensures we always end up at reserve values from quoter
        // and we don't do bogus claims about states that the AMM would not naturally end up in
        // as it's easy to steal if previous swapper does a direct swap that ends up far above the curve
        {
            // need to swap left first, so we can exactIn swap right
            uint256 amountIn = 1e6;
            assetTST2.mint(address(this), amountIn);
            assetTST2.approve(address(periphery), type(uint256).max);
            periphery.swapExactIn(address(eulerSwap), address(assetTST2), address(assetTST), amountIn, 0);

            // we want to end up at xt + dx reserves
            amountIn = (xt + dx) - eulerSwap.reserve0();
            assetTST.mint(address(this), amountIn);
            assetTST.approve(address(periphery), type(uint256).max);
            periphery.swapExactIn(address(eulerSwap), address(assetTST), address(assetTST2), amountIn, 0);
        }

        // we can steal funds without transferring anything to the pool
        eulerSwap.swap(dx, 0, address(0x42), "");
        assertEq(assetTST.balanceOf(address(0x42)), dx);
    }

    /// @dev EulerSwap curve definition
    /// Pre-conditions: x <= x0, 1 <= {px,py} <= 1e36, {x0,y0} <= type(uint112).max, c <= 1e18
    function f(uint256 x, uint256 px, uint256 py, uint256 x0, uint256 y0, uint256 c) internal pure returns
    ↪  (uint256) {
        unchecked {
            uint256 v = Math.mulDiv(px * (x0 - x), c * x + (1e18 - c) * x0, x * 1e18, Math.Rounding.Ceil);
            require(v <= type(uint248).max, Overflow());
            return y0 + (v + (py - 1)) / py;
        }
    }
}
```

Euler: There are a few related issues here:

- Commit 0c6238c3 ensures that the initially installed reserves are not vulnerable to this, by independently making sure that neither x nor y direction can have a single wei removed.
- But in general, we acknowledge that a swapper can leave the pool in a state where some amount of excess assets was not claimed. If the quoting method determines that no additional funds can be removed in one direction, it does not necessarily imply that funds cannot be removed in the other direction according to the slope analysis in this issue. We consider this a type of "overswapping", and do not believe that it impacts the security of LP funds. A smarter periphery could find tight bounds in both directions (using something like `skimAll()` function in the tests), but this would likely cost

more in gas than it actually saves in realistic scenarios.

- The follow-up issue about incorrect exact output quoting when the point is above the curve was also noted by chainsecurity, and fixed in commit 54c1af6d.

Spearbit: Verified.

3.2 Gas Optimization

3.2.1 `EulerSwapPeriphery.computeQuote` can be optimized

Severity: Gas Optimization

Context: EulerSwapPeriphery.sol#L117

Description: The `EulerSwapPeriphery.computeQuote` function always performs a binary search to find the quote for the swap. This can be optimized to a single f function evaluation in half of the cases, and for the cases that require evaluating f^{-1} the binary search range can be reduced, roughly cutting the required iterations in half in practice.

Curve: The curve is defined by these two functions:

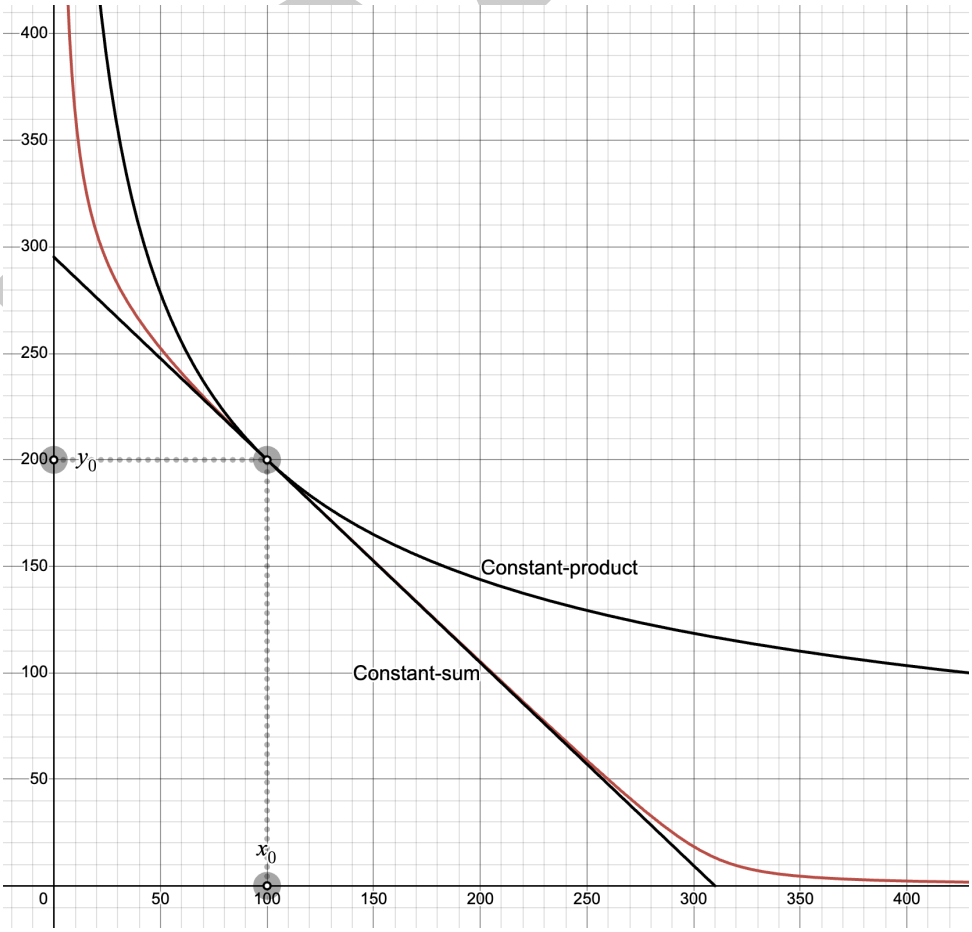
$$f_1 : (0, x_0] \to [y_0, \infty), \qquad x \mapsto y_0 + \frac{p_x}{p_y}(x_0 - x)\left(c_x + (1 - c_x)\left(\frac{x_0}{x}\right)\right)$$

$$f_2 : (0, y_0) \to (x_0, \infty), \quad y \mapsto x_0 + \frac{p_y}{p_x}(y_0 - y)\left(c_y + (1 - c_y)\left(\frac{y_0}{y}\right)\right)$$

The curve is the set of points:

$$\{(x, f_1(x)) \mid x \in (0, x_0]\} \cup \{(f_2(y), y) \mid y \in (0, y_0)\}$$

f_1 is the red curve part left of x_0 (to the top of y_0), f_2 is the red curve part right of x_0 (to the bottom of y_0).



Quotes: A swap quote can be one of four combinations, depending on `(exactIn, isAsset0In)`. Quoting works by computing the new specified reserve coordinate and finding the point on the curve for it. The resulting quote is then the difference between the current and new unspecified reserve amount.

1. For `exactIn == isAsset0In`, let `x` be the reserve updated by the specified amount; we need to find `(x,y)` on the curve:
    1. If $x \leq x_0$ compute $y = f_1(x)$.
    2. If xx_0 compute $y = f_2^{-1}(x)$, i.e., solve $f_2(y) = x$ with binary search in the range $y \in (0, y_0)$.
2. For `exactIn != isAsset0In`, let `y` be the reserve updated by the specified amount; we need to find `(x,y)` on the curve:
    1. If yy_0 compute $x = f_2(y)$.
    2. If $y \geq y_0$ compute $x = f_1^{-1}(y)$, i.e., solve $f_1(x) = y$ with binary search in the range $x \in (0, x_0]$.

Recommendation: Consider the following changes:

1. Extract the `f` function into a library that is imported by both `EulerSwap` and `EulerSwapPeriphery`.
2. When quoting, cache the curve parameters once and evaluate `f` with the imported library. This avoids a call overhead for each binary search iteration.
3. Implement the above algorithm.

Euler: Acknowledged. We do intend to improve the efficiency of quoting and appreciate this thoughtful analysis.

Spearbit: Acknowledged.

3.3 Informational

3.3.1 Documentation and minor issues

Severity: Informational

Context: See below

Description: We've identified various typos, minor issues, and documentation inconsistencies throughout the code:

- EulerSwap.sol#L131: `satisified` → `satisfied`.
- architecture.md#reserve-synchronisation:

    > "Reserve synchronisation": When this occurs, the `syncVirtualReserves()` should be invoked. This determines the actual balances (and debts) of the holder and adjusts them by the configured virtual reserve levels.

    This function doesn't exist, consider documenting maybe that you should move to a new EulerSwap instance if the actual balances (and debt) go out of sync and a swap would threaten ending up close liquidation.
- IEulerSwapFactory.sol: This interface has no NatSpec.
- EulerSwapPeriphery.sol#L76: The `swap` function should receive `IEulerSwap` as parameter to avoid doing `IEulerSwap(eulerSwap)` in the function same as you do in `computeQuote`.

Recommendation: Consider resolving the aforementioned documentation issues.

Euler: Fixed in commit c55ebc3f. Also:

- `src/interfaces/IEulerSwapFactory.sol` NatSpec was fixed in 9624a51a.
- `src/EulerSwapPeriphery.sol` `eulerSwap` type change in commit 6d35d872.

Spearbit: Fixes verified.

3.3.2 Read-only reentrancy on `getReserves`

Severity: Informational

Context: EulerSwap.sol#L116-L152

Description: The `swap` function follows this sequence of operations:

1. Calls `withdrawAssets`, which optimistically sends the tokens to `to`.
2. If `data.length > 0`, it invokes `uniswapV2Call` on `to`.
3. Calls `depositAssets` to deposit all available funds in the contract.
4. Applies the invariant check.
5. Updates the reserves.

As we can see, the reserves are updated at the final step. This means that until that point, the reserves reflect stale values. If `getReserves` is called during the callback or within a vault hook, it will return outdated reserve values. This can lead to a read-only reentrancy issue, as external contracts might read an inconsistent state.

Recommendation: Consider documenting this behavior explicitly. Additionally, consider adding a view `nonReentrant` modifier to `getReserves()`, which would revert if reentered during a swap.

Euler: Fixed in commit e44a49ac.

Spearbit: Verified.

3.3.3 Factory allows deployments with malicious vaults

Severity: Informational

Context: EulerSwap.sol#L73

Description: Even though there's an `EulerSwapFactory` and the `EulerSwap` constructor performs basic checks on the vault, the vaults can still be malicious. An attacker can deploy an `EulerSwap` deployment through the factory with a malicious vault that, for example, performs a noop on `withdraw`.

Users swapping directly in the pool or through the `EulerSwapPeriphery` contract deposit their funds to the attacker's vault but will not receive any output in return as the token transfers are solely at the `vault`'s discretion (through the `withdraw` and `borrow` actions).

Recommendation: Users, aggregators and a potential `EulerSwap` frontend need to detect malicious `EulerSwap` contracts. Currently, it's not obvious how to check for pool legitimacy. Consider only allowing EVK vaults in the `EulerSwap` factory to legitimize all of its deployments. (EVK vaults are deployed by their EVK factory).

Euler: Fixed in commit 79158fe5.

Spearbit: Verified.

3.3.4 `EulerSwap` does not support "borrow-only" vaults

Severity: Informational

Context: EulerSwapPeriphery.sol#L60

Description: A swap in `EulerSwap` always performs a `deposit` followed by a `repayWithShares`. Vaults that only allow `borrow`/`repay` are not supported (even if the curve is parametrized in such a way to only hold debts) as the repay flow reverts in the `deposit` part due to the hook configuration.

Recommendation: If these vaults are supposed to be supported, consider repaying the debt first before depositing.

Euler: Fixed in commit 62bdbcea.

Spearbit: Verified.

3.3.5 Equilibrium point verification is unnecessary

Severity: Informational

Context: EulerSwap.sol#L99

Description: The `verify` function checks that the given point's coordinates are at most `uint112.max` and check that the point is on the curve. As the equilibrium point is on the curve parameter by definition and `Params.equilibriumReserve0/1` are `uint112`, the equilibrium point must not be verified.

Recommendation: Consider removing the check.

```
- require(verify(equilibriumReserve0, equilibriumReserve1), CurveViolation());
```

Euler: Fixed in commit 0c6238c3.

Spearbit: Verified.