PUBLIC

Code Assessment

# of the EulerSwap Smart Contracts

Mar 31, 2025

Produced for

Euler

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Euler with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of EulerSwap according to Scope to support you in forming an opinion on their security risks.

Euler implements EulerSwap, an AMM that uses a custom bonding curve, and enhances its liquidity by borrowing additional funds from Euler vaults.

The most critical subjects covered in our audit are functional correctness, precision of arithmetic operations, and front-running. Security regarding all the aforementioned subjects is good but improvable, see Unaccounted roundings when depositing and withdrawing and Balance of Euler account counted twice in calcLimits(). Notice that Euler decided not to fix some of the issues for the time being; these issues have been marked Acknowledged.

The general subjects covered are code complexity, gas efficiency, and trust relationships. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a good level of security, which could be improved if all the outstanding issues were to be addressed.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	9
• Code Corrected	5
• Specification Changed	1
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the EulerSwap repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 Feb 2025	ec46a582d7666f0c2cb333005917fdb353e52c78	Initial Version
2	5 Mar 2025	54c1af6d7891583ddc0df332002bcbc6e06bb4cd	Version with Fixes

For the solidity smart contracts, the compiler version `0.8.27` was chosen.

This review covers the EulerSwap smart contracts:

- src/EulerSwap.sol
- src/EulerSwapPeriphery.sol
- src/interfaces/IEulerSwap.sol
- src/interfaces/IEulerSwapPeriphery.sol
- src/interfaces/IUniswapV2Callee.sol

2.1.1 Excluded from scope

Any files not explicilty mentioned in the `Scope` are not part of this review. This includes third party libraries, as well as integrated external protocols (Euler Vault Connector, Euler Vaults, Permit2), which are assumed to always work correctly and according to specification.
Tests and deployment scripts are also out of scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Euler offers EulerSwap, an Automated Market Maker where swap liquidity is enhanced by borrowing from Euler Vaults. EulerSwap is a simple AMM with a single liquidity provider, the owner. The AMM parameters, including the liquidity, are set at deployment time, and are immutable. EulerSwap is a non-custodial AMM, in the sense that it does not hold the funds that it swaps. The funds are held by the owner in Euler Vaults, and EulerSwap is configured as an *Ethereum Vault Connector* (EVC) **operator** for the owner. The AMM is based on a custom bonding curve, which is defined piece-wise by two branches around an initial point. Each of the two branches is a weighted average between a constant sum and a constant product curve, the concentration parameter (how much the weight is toward the constant sum term) can be different on the two branches, allowing asymmetric liquidity concentration around the initial reserves. The AMM earns fees for the account owner, which are levied from the swaps inputs and deposited in the owner's account, without modifying the shape of the curve. The AMM borrows tokens for swap outputs on behalf of the owner, collateralizing the loans with the swap inputs, as well as the initial "liquid" reserves, if present. The debt accrues interest. It is the intention of EulerSwap to earn more fees from swaps than the interest that needs to be repaid.

2.2.1 The EulerSwap Curve

EulerSwap implements a custom bonding curve that defines the minimum amount of `asset0` reserves as a function of `asset1` reserves and vice-versa. It is defined piece-wise around the point (x_0, y_0), which are the initial reserves, and is parametrized by p_x, p_y, the initial prices of the two assets, and c_x, c_y, the liquidity concentrations of the two branches of the function.

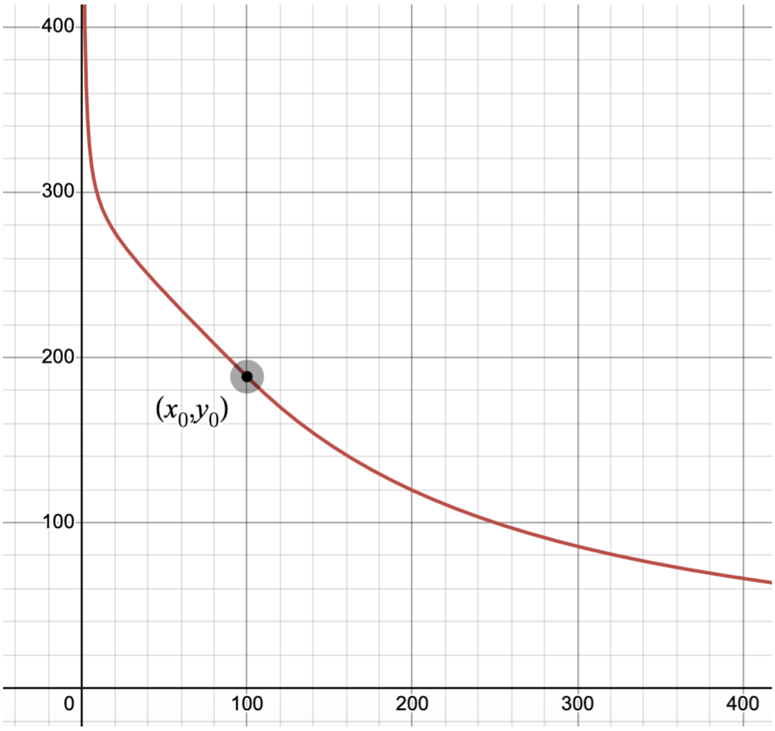
The curve is continuous and differentiable in (x_0, y_0) and its slope is $-\frac{px}{py}$. When $x \leq x_0$, it is defined as:

$$y = y_0 + \frac{p_x}{p_y}(x_0 - x)\left(c_x + (1 - c_x)\left(\frac{x_0}{x}\right)\right)$$

When $x > x_0$, it is defined as a function of y as:

$$x = x_0 + \frac{p_y}{p_x}(y_0 - y)\left(c_y + (1 - c_y)\left(\frac{y_0}{y}\right)\right)$$

Parameters c_x and c_y act as weighting factors between a constant product curve and a constant sum curve, with $c = 1$ yielding constant sum, and $c = 0$ constant product. Inbetween values allow to concentrate liquidity more or less around the central point (x_0, y_0).



In the above image we can see the curve, configured with $c_x \sim 1$ and $c_y \sim 0$.

Since the curve concentrates liquidity around the (x_0, y_0) point, where the price is fixed to $\frac{p_x}{p_y}$, the curve is more suitable for use in pegged assets such that `asset0` and `asset1` have a *true* exchange rate which is fixed, or evolves slowly. Such pairs are for example wstETH and WETH, or USDC and USDT.

2.2.2 Virtual balances and borrowing

Differently to conventional AMMs, EulerSwap does not hold the reserve amounts that it is ready to transfer out through swapping. EulerSwap uses virtual reserves, accessible through public view functions `reserve0()` and `reserve1()`. These represent amounts of `asset0` and `asset1` that characterize the state of the AMM with respect to the bonding curve, but they are not backed by actual token balances in the contract. They are partially backed by deposits in the Euler vaults `vault0` and `vault1`, and the remaining part is backed by "potential" debt, that is, EulerSwap is willing to borrow up to `reserve0 - myBalance(vault0)` of `asset0` to perform a trade, where `myBalance()` is a function that returns how much `asset0` deposits the EulerSwap contract can withdraw from the vault. The vaults are lending vaults which are mutually collateralized: `vault0` accepts `vault1` shares as collateral to back loans, and conversely.

Swapping out an asset therefore first decreases deposits, and when deposits in a vault are exhausted, it resorts to borrowing from the vault. While swapping out one asset, the vault receives the other asset, which it uses as collateral for borrowing. EulerSwap is initialized with a certain amount of deposits in the vaults, and *debt limits* for the two assets. The amounts of initial deposits plus debt limits for each token make up the initial virtual reserves (x_0, y_0).

As a result of swapping out `asset1`, if the price stays fixed to $\frac{p_x}{p_y}$, the vault can take at most `debtLimit1` debt. The amount of the `asset0` tokens it holds on the other hand will be at least:

```
initialBalance0 + initialBalance1 * p_y / p_x + debtLimit1 * p_y / p_x
```

or, expressed in the `asset1` denomination:

```
initialBalance0 * p_x/p_y + initialBalance1 + debtLimit1
```

This amount comes from the curve being convex, and the least price that the swap receiver is paying for `asset1` is therefore $\geq \frac{p_y}{p_x}$ (at least the price in the initial position). The previous formula shows that the value of the collateral the pool holds after swapping out all available `asset1` (initial balance plus debt limit) is at least `initialValue + debtLimit1`. So an upper bound on the loan-to-value (ltv) achievable by users swapping to a given asset while the price stays pegged is:

$$\frac{debtLimit}{initialValue + debtLimit}$$

This approximation is conservative and assumes constant-sum curve, and more convex curves can allow higher debt limits while keeping the same max ltv, since the collateral will be received at a more and more favorable price because of slippage. Owners of EulerSwap should consider this formula, or more refined versions, when setting initial parameters such that the maximum ltv achievable by swapping is below the borrow ltv (configured maximum ltv to borrow) and liquidation ltv (maximum ltv before liquidation) of the Euler vault, such that swapping cannot revert because of ltv checks (borrow ltv), and does not bring the EulerSwap loan on the edge of liquidation (liquidation ltv checks, if liquidation ltv is equal to borrow ltv). See the note on LTV checks for more details.

If the real market prices of the two assets change (as returned by the oracles in the Euler lending vaults), then the configured max ltv can be exceeded. Asset pairs should therefore be strongly correlated with low risk of depeg, and in the event of depegs, EulerSwap can incur liquidations.

2.2.3 Reserves (de)synchronization

The EulerSwap contract has its own internal accounting to keep track of its current (virtual) reserves: these are *only* updated upon swaps (and *only* according to the input and output amounts), and always respect the invariant defined by the immutable curve parameters set at pool-creation time. In other words, the pool never readjusts in response to events that modify the "actual" credit/debit situation of the Euler account, and will keep serving swaps indefinitely, always "blindly" honoring the same bonding curve.

At the beginning, the virtual reserve is set to the configured `debtLimit` plus the initial "real" balance (or minus the initial debt). This, together with the fact that the bonding curve has asymptotes at the (virtual) axes, is a "soft" guarantee that the Euler account will "never" incur a debt higher than `debtLimit`; or, slightly more accurately, the pool will never be the "direct cause" for more than `debtLimit` in debt.

There are, however, several sources of "desynchronization", that throw this initial equation off balance. These include:

- Swap fees. They are levied on the input tokens, and deposited into the Euler vault, but they are not "reinvested" in the strict sense, meaning that the curve does not "expand" to account for the new liquidity.
- Interest paid and received on debts and deposits.
- Bad debt socialization, leading to a depreciation of the vault shares, and thus to a reduction of the "token balance" as reported by `myBalance()`
- Liquidation of the Euler account, leading to a reduction in the number of shares in possession, and thus to a reduction of the "token balance".
- Concurrent action on the same Euler account, either by the same owner, or by other operators.

All these events lead to the "actual" balances (as reported by `myBalance()` and `myDebt()`) to change without the pool noticing, thus the equation `reserve = debtLimit + myBalance()` (or `-myDebt()`) no longer holds true after some time. Depending on "which side" the equation has become unbalanced, the "effective debt limit" (i.e. the debt of the Euler account, in case the entire virtual reserve were to be swapped out) can become higher or lower than the configured `debtLimit`. Should it become significantly higher, this could lead to swaps failing, due to the maximum borrow ltv being reached; also the debt interests and the risk of liquidation would increase, feeding back into this "vicious circle". Should it become significantly lower, this would lead to a "reduced capital efficiency", as the pool would be trading with less leverage than initially configured; in the limit case where the swap fees and the interests received exceed the configured `debtLimit`, this would lead the pool to never borrow anything.

Euler acknowledges that, when this desynchronization has become significant enough, admins should "decommission" the pool by removing it as an EVC operator, and redeploy.

2.2.4 EVC and Euler Vault Kit

The EVC (*Ethereum Vault Connector*) is a singleton contract that enables call batching, deferred liquidity checks, and account management on Euler vaults. EulerSwap acts as an EVC operator on behalf of its owner (`myAccount`). An operator is an EVC role that is approved by an account to act on their behalf. Euler vaults (vaults implemented with the Euler Vault Kit) allow the EVC to operate on behalf of any account, so operators can interact with Euler vaults with spoofed message sender, acting as their account owner.

Calling an EVC enabled contract on behalf of another account can be done through calls to the `EVC.call()` function, which performs a call to an EVC-enabled target contract on behalf of a target account. The EVC performs the authentication necessary to ensure that the caller of `EVC.call()` is authorized to act on behalf of the target account, meaning it is the account owner or an operator. In practice, this allows EulerSwap to borrow, withdraw, enable collaterals, enable controllers, on behalf of `myAccount`.

EulerSwap borrows from Euler Vault Kit (EVK) vaults, through the EVC, by acting as an operator for `myAccount`. To borrow from EVK vaults, `myAccount` must own EVC-enabled collateral (in practice shares of other EVK vaults), and must grant right to the lending vault to grab the collateral in case of liquidation. This happens by setting the lending vault as a controller for `myAccount`, through `EVC.enableController()`. This operation guarantees that every operation that changes the collateral balance will generate a liquidity check (check that collateral covers the debt), and it also allows the controller vault to call the collateral vaults on behalf of the borrower (to appropriate collateral during liquidations). The collateral must be enabled in the EVC with `enableCollateral()`, such that the controller can interact with it on behalf of `myAccount`.

The EVC defers liquidity checks to the end of the EVC execution, and multiple operations can execute in a single execution context, such that flash-borrowing is possible. In practice, loans needs not to be collateralized when they are opened, as long as they are collateralized at the end of EVC execution. An account can only have a single controller enabled at the end of EVC execution, but can have multiple controllers enabled transiently. The `swap()` function of EulerSwap is decorated with the `callThroughEVC` decorator, which wraps all the operations performed in `swap()` in a single EVC execution. This allows multiple loans to be open simultaneously during `swap()`, and to borrow the assets before the collateral is received. At the end of the execution of `swap()`, however a single loan must exist, and it must be fully collateralized.

2.2.5 Contract creation and activation

The EulerSwap contract is used as an EVC operator for account `myAccount`. It is only configured once, at creation time, by specifying a set of immutable parameters:

- `vault0` and `vault1`: the addresses of the two vaults which hold the swapped assets. They should allow borrowing and accept each other as collateral.
- `myAccount`: the account which holds the balances of `vault0` and `vault1` shares, and the debt to fund the swaps.
- `debtLimit0` and `debtLimit1`: maximum amounts of debt for `asset0` and `asset1` that `myAccount` can incur from swaps.
- `fee`: the portion of the input amount that is deduced for the benefit of `myAccount`.
- `priceX` and `priceY`: the price of `asset0` and `asset1` with respect to a reference asset. The prices are quoted "per wei", such that two tokens with the "same" price but different decimals such as USDC (6 decimals) and LUSD (18 decimals) will have a price ratio that's scaled appropriately.
- `concentrationX` and `concentrationY`: the *liquidity concentration* parameters for the two sides of the curve. Concentration closer to 10**18 means the weighted average between constant sum and constant product is more towards constant sum, while concentration towards 0 means more constant product.

The newly created EulerSwap instance has to be configured by the owner of `myAccount` as an operator for `myAccount` in the EVC.

Finally the EulerSwap instance has to be activated, by calling `activate()`. Activation sets the approvals of EulerSwap to the vaults such that the vaults can pull assets on deposits, and enables the vaults as collaterals for `myAccount` in the EVC, such that one of the two assets can be borrowed with the other as collateral.

2.2.6 EulerSwap swaps interface

The EulerSwap contract offers a low level swap interface similar to Uniswap v2.

The `swap` function accepts four arguments:

- `amount0Out`: amount of `asset0` to be sent to the `to` address.
- `amount1Out`: amount of `asset1` to be sent to the `to` address.
- `to`: the receiver of the out amounts, and the target of the callback is `data` is not empty.
- `data`: data for the optional callback.

At the beginning of the swap, `amount0Out` and `amount1Out` are sent to the `to` address respectively from `vault0` and `vault1`. If `myAccount` has enough deposits in the vaults, the deposits are withdrawn to the `to` address. If `myAccount` does not have enough deposits in `vault0` and `vault1` for the amounts, the difference between the deposits and the requested amount is borrowed from the vaults.

As the next step of the swap, an optional callback is executed, if `data` is not empty, calling the `uniswapV2Call()` method of the `to` address. The same callback interface as uniswap v2 is implemented to facilitate integration. After the callback returns, the input funds are deposited to `vault0` and `vault1`, and any debt with those vaults is (partially) repaid. Any amount of `asset0` or `asset1` present in the contract after the callback is considered as input funds, and they could have been received by the EulerSwap contract either before the `swap()` invocation, or during the callback. A fee is deducted from the input funds. Fees are deposited into the vaults on behalf of `myAccount`. Finally, the new reserves are verified to satisfy the invariant. Typically a swap will involve `asset0` as input and `asset1` as output or vice-versa, but not necessarily. A swap can specify both tokens as outputs, and provide both tokens as inputs, as long as the invariant is honored after the swap.

Two internal functions, `withdrawAssets()` and `depositAssets()`, control the transfer of assets between the vaults, the EulerSwap contract, and the swap receiver (`to` address).

The `withdrawAssets()` function transfers a requested `amount` of an underlying asset from a `vault` to a given address. If `myAccount` has enough asset balance in `vault`, the asset is entirely withdrawn of the vault (burning shares owned by `myAccount` through `vault.withdraw()`). If the asset balance of `myAccount` in the vault is not sufficient, the remaining amount is borrowed by `myAccount` from the vault. The calls to `borrow()` and `withdraw()` are performed by the EulerSwap contract through the EVC, acting on behalf of `myAccount` since it is an operator.

The `depositAssets()` function deposits a given asset amount to `vault` on behalf of `myAccount`, minting vault shares through `deposit()`. If `myAccount` has outstanding debt with the vault, the debt is (maybe partially) repaid by calling the `repayWithShares()` method of the vault through the EVC on behalf of `myAccount`.

2.2.7 EulerSwapPeriphery contract

The `EulerSwapPeriphery` contract implements methods to compute price quotes for EulerSwap instances. It is designed to work with contracts implementing the `IEulerSwap` interface, allowing future use of different curves.

The contract exposes two external methods: `quoteExactOutput()` and `quoteExactInput()`.

`quoteExactInput()` allows specifying a given `amountIn` of `tokenIn`, and returns the amount of `tokenOut` that can be received from the given EulerSwap instance while still satisfying the invariant.

Conversely, `quoteExactOutput()` allows specifying an `amountOut` of `tokenOut` to receive from an `eulerSwap` instance, and returns how much `tokenIn` must be transferred to satisfy the invariant.

Internally these methods find the least amount of input or output tokens that solve the invariant by bisection search between the minum value of `0` and the maximum value of `type(uint112).max`.

2.2.8 Changes in Version 2

The deployer of an `EulerSwap` instance no longer needs to provide an explicit `debtLimit`; instead, he directly provides the initial (virtual) reserves. Furthermore, the initial reserves are now allowed to be different from the equilibrium point. The `EulerSwapPeriphery` now contains additional refined logic to estimate the maximum input and output amounts for swaps, and exports it as a view function. Moreover, slippage-protected wrappers around the low-level swap have been added.

2.2.9 Trust Model and Caveats

- The account owner (`myAccount`) is assumed to not modify the configuration with respect to balances, debts, enabled collaterals and enabled controllers while EulerSwap is in operation, as this could desynchronize EulerSwap with respect to `myAccount` and cause reverts for swappers.
- Vaults do not need to be trusted by the swapper, however the swapper should be aware that untrusted vaults could execute arbitrary code (for example through hooks). Untrusted vaults could

perform re-entrancies against the caller. The `to` callback receiver should be protected against re-entrancies when interacting with EulerSwap that use untrusted vaults.

- `token0` and `token1` are assumed to be non rebasing, incur no fees on transfer, and are trusted to not execute malicious code.
- `myAccount` fully trusts `vault0` and `vault1`. Badly configured vaults (malicious oracles for example) could liquidate `myAccount` at will.
- The caller of `swap()` trusts that `vault0` and `vault1` are valid instances of EVK vaults. If they are malicious contracts, the swapper could not receive the `amount0Out` and `amount1Out` requested (`vault.withdraw()` does not send the tokens).
- The functions `quoteExactInput()` and `quoteExactOutput()` of the periphery are not used to compute prices on-chain without an additional slippage protection mechanism (which queries an oracle such as Chainlink or uses an off-chain quote), as the on-chain quoting can be sandwiched to return arbitrary prices.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

- Balance of Euler Account Counted Twice in calcLimits() Acknowledged
- Comparing amountIn "After Taxes" Acknowledged
- Unaccounted Roundings When Depositing and Withdrawing Acknowledged

5.1 Balance of Euler Account Counted Twice in `calcLimits()`

Correctness Low Version 2 Acknowledged

CS-EULSWP-015

The function `EulerSwapPeriphery.calcLimits()` takes the available cash in the vault as a limiting factor for the output amount. However, the balance of the Euler account is then added to such a limit, even though it's already part of the cash that was taken into account before. This leads to the balance being counted twice, if `maxWithdraw > cash - balance`.

Acknowledged:

Euler said:

> Acknowledged. I will look into fixes for these.

5.2 Comparing `amountIn` "After Taxes"

Correctness Low Version 2 Acknowledged

CS-EULSWP-016

The function `EulerSwapPeriphery.computeQuote()` checks the input amount (either `amount` or `quote`) against the computed `inLimit`. However, it does so on the "reduced" input amount, after deducting the swap fee, even though the full amount will be deposited in the vault.

Acknowledged:

Euler said:

> Acknowledged. I will look into fixes for these.

5.3 Unaccounted Roundings When Depositing and Withdrawing

Correctness Low Version 1 Acknowledged

CS-EULSWP-018

The functions `depositAssets()` and `withdrawAssets()` of `EulerSwap` do not take into account the rounding errors that happen internally in the Euler vault.

The function `depositAssets()` effectively lets the vault pull exactly the specified amount of tokens, but mints a number of shares that is worth slightly less. The function `withdrawAssets()` also does transfer the specified amount of tokens, but by burning a number of shares that is worth slightly more.

The difference is not a direct profit to the swapper, because the specified amounts are still correctly transferred; instead, the pool incurs a small loss (comparable to the price of 1 wei of shares) at the advantage of the vault. The swapper can then extract some of that profit, in case he is a major contributor to the vault. Moreover, this is a further source of desynchronization between `myBalance()` and `reserve0/1`, which builds up over time.

Acknowledged:

Euler said:

> This is technically a good point, but I don't really see this causing an issue in practice.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- Donation Attack Results in a DoS Code Corrected

Low-Severity Findings	6

- Bad Rounding Direction Code Corrected
- Hardcoded Precision for Equilibrium Price May Be Low Code Corrected
- Potential Overflows in Periphery Contract Code Corrected
- Rounding Error Gets Amplified Code Corrected
- Simultaneous Debt and Balance Lead to Incorrect Parametrization Specification Changed
- high Assumed to Always Verify in Binary Search Code Corrected

Informational Findings	3

- Clamping of Initial Reserves Preserves Excessive Debt Specification Changed
- Redundant address(this) in Event Code Corrected
- Requesting type(uint256).max to the Swap Fails With Incorrect Error Code Corrected

6.1 Donation Attack Results in a DoS

Correctness Medium Version 1 Code Corrected

CS-EULSWP-001

The function `EulerSwap.swap()` treats any liquid balance of both tokens as an input amount, and tries to deposit it. However, EVK vaults revert with `E_ZeroShares` if the deposited amount is so low that it would result in zero shares being minted (given the downward direction of rounding).
Therefore, an attacker can front-run any swap with a small donation to the pool, equal to 1 wei of the out-token. This will be treated as an input amount by the pool and, as explained, will lead to the deposit failing, and the whole swap reverting.

Code corrected:

The abovementioned error in Euler vaults is now explicitly caught through a `try/catch` statement, and ignored, leading to the liquid balance staying in the pool until the next swap.

6.2 Bad Rounding Direction

Correctness Low Version 1 Code Corrected

CS-EULSWP-002

The function `EulerSwap.f()` should round up the result, given that it is only used as a sufficiency check in `verify()`. Rounding down results in fewer input tokens being accepted by the pool for the same swap.

Code corrected:

The code has been revisited in Version 2 to consistently round upwards

6.3 Hardcoded Precision for Equilibrium Price May Be Low

Design Low Version 1 Code Corrected

CS-EULSWP-003

The equilibrium price `px/py` is effectively used, in function `f()`, with a hardcoded precision of 18 decimal places. The (worst-case) absolute error of `1e-18` may be too big for certain pairs, in case the tokens differ a lot in value and/or number of decimals.

Code corrected:

In Version 2 of the codebase, the variables `px` and `py` are now treated as scale-free in the computation of `f()`, and their precision has been increased to `1e36`.

6.4 Potential Overflows in Periphery Contract

Design Low Version 1 Code Corrected

CS-EULSWP-004

In the `binarySearch()` internal function of `EulerSwapPeriphery`, several unsafe casts and arithmetic overflows and underflows may be potentially problematic.

Version 1:

1. At lines 101,102,104,105, unsafe casts from `uint256` to `int256` are performed for the variable `amount`. If `amount` is outside the range of `int256`, the unsafe cast will fail silently.
2. At lines 109 and 110, unchecked additions are performed that could potentially overflow, since `dx` and `dy` can be as small as `type(int256).min` causing underflows, or as high as `type(int256).max` causing overflows.
3. At lines 133 and 134, potential unsafe casts are performed on the variables `dx` and `dy`. These signed integers are assumed to be postive, but they could be negative, in case when the current reserves are above the curve (`low - reserve <= 0`). This can happen if `computeQuote()` is called for an exact output of token A when the current reserves are above the curve, such that getting the token A output does not require paying token B, but actually can allow withdrawing token B also. The assumption that the point (`reserve0`, `reserve1`) lay on the curve is in general incorrect.

Version 2:

1. At the beginning of the `unchecked` block, `reserve0/1` is added to `dx/dy`. Even though both are within the `uint112` range, their sum might not.

The potentially incorrect arithmetic operations could cause issues depending on how `EulerSwapPeriphery` is used on-chain and off-chain. Addressing them prevents unforeseen negative scenarios.

Code corrected

The points present in Version 1 have been addressed. The issue present in Version 2 is not problematic, because a subsequent `require()` statement indirectly prevents the function from succeeding in this case.

6.5 Rounding Error Gets Amplified

Correctness Low Version 1 Code Corrected

CS-EULSWP-005

The definition of function `f()` in `EulerSwap` comprises a parenthesized addend, namely `(1e18 - c) * x0 / 1e18 * x0 / xt`, whose final rounding error might be large compared to its "true" value. This is because the rounding error of an integer division is amplified by a subsequent multiplication: the first division incurs an error of at most 1, which then gets multiplied by the factor `x0/xt`. This rounding error, "denominated" in weis of "token `x`", gets then multiplied by `px/py` (with precision), and ends up in an equivalent underevaluation of `y`. The difference (the underestimation of the curve) is a profit to the swapper.

Code corrected:

The code has been revisited in Version 2 to avoid this amplification

6.6 Simultaneous Debt and Balance Lead to Incorrect Parametrization

Correctness Low Version 1 Specification Changed

CS-EULSWP-006

In `EulerSwap`, the functions `myDebt()` and `myBalance()` are not "mutually exclusive", in that they can both be non-zero at any given time. If this happens at pool-creation time, this will lead to an incorrect calculation of the initial (virtual) reserves of the corresponding token.
Specifically, the function `offsetReserve()` will compute the virtual reserve as `debtLimit - myDebt()`, whereas it should really return `debtLimit - myDebt() + myBalance()`. This effectively decreases the debt limit by `myBalance()`. If `myBalance() > debtLimit`, the pool will never borrow anything from the vault, and will never even fully utilise the liquid "real" reserves.

Specification changed:

Euler said:

> The offsetReserve() function was removed. Instead, this can be computed externally when creating the EulerSwap instance. An incorrect parameterisation may still occur, but an operator can decide whether this is acceptable.

6.7 `high` Assumed to Always Verify in Binary Search

Design Low Version 1 Code Corrected

CS-EULSWP-007

The `high` bound of the binary search is never tested with `eulerSwap.verify()`, and is assumed to always verify since it has the highest value a reserve could take. This assumption is incorrect, as the curve value, returned by `EulerSwap.f()`, can be above `type(uint112).max`, and is typically above it in proximity of the asymptotes. If the curve is above `type(uint112).max`, it defines regions which are not reachable by swaps, since the reserves can at most be `uint112.max`, and can't be below the curve.

When asking an exact output quote which would move the reserves to a region where the curve is above `type(uint112).max` (therefore inaccessible), the function `quoteExactOutput()` will return input quotes which are too low for the desired output.

Code corrected:

The code has been revisited in Version 2, and now a check is present *after* the `while` loop, ensuring that at least one iteration had a value that verified.

6.8 Clamping of Initial Reserves Preserves Excessive Debt

Informational Version 1 Specification Changed

CS-EULSWP-008

The function `EulerSwap.offsetReserve()` clamps the result to `0`, in case the subtraction `debtLimit - myDebt()` would be negative, that is, in case the current debt is already larger than the desired cap, at pool-creation time. The result is that one of the two "arms" of the curve is unusable (collapsed to a single point). The "other arm" will be fully functional, and it will as usual pass through the point `(x0,y0)` (with either of them being 0): the initial over-indebtedness situation will then forever be a reachable pool state, and it will even be its ideal "equilibrium state", where the most liquidity is concentrated.

Specification changed:

In Version 2 of the codebase, `debtLimit` is no longer an explicit parameter supplied by the admin at construction time. Euler said:

> The offsetReserve() function was removed. At creation time the operator can decide if this behaviour is allowed or not.

6.9 Redundant `address(this)` in Event

Informational Version 1 Code Corrected

CS-EULSWP-012

The event `EulerSwapCreated` includes a field `address indexed eulerSwap` which is set to `address(this)` when emitting the event at the end of the `EulerSwap` constructor. This field is redundant, as event logs already automatically include the address of the emitter.

Code corrected:

The redundant field was removed

6.10 Requesting `type(uint256).max` to the Swap Fails With Incorrect Error

Informational Version 1 Code Corrected

CS-EULSWP-013

Calling swap with `type(uint256).max` as `amount0Out` should revert, because the amount is necessarily too high. However, calling `vault.borrow()` with an amount of `2**256` succeeds and borrows all the cash in the vault. The `withdrawAssets()` in swap therefore succeeds, and sends to the `to` address `vault.cash` instead of `amount0Out == 2**256`.

The function execution eventually reverts because of the line:

```
uint256 newReserve0 = reserve0 + amount0In - amount0Out;
```

subtracting `amount0Out`, which has value `2**256` will necessarily underflow. However the function execution should have reverted earlier with a more informative error.

Code corrected:

The function now explicitly check that the amounts lie within the `uint112` range.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Redundant Checks

Informational Version 2

CS-EULSWP-019

In the `EulerSwap` constructor, the requirement that the equilibrium point lie on the curve is unnecessary, since those immutable variables are what define the curve itself. The function `verify()` will always return true when called at the equilibrium point.

7.2 Reentrancy Not Checked

Informational Version 2

CS-EULSWP-020

The functions `computeQuote()` and `calcLimits()` of `EulerSwapPeriphery` call `EulerSwap.getReserves()` ignoring its `status` return value, signaling whether a swap is already happening (i.e. we are in the callback of a swap).

7.3 Gas Optimizations

Informational Version 1 Acknowledged

CS-EULSWP-009

The following opportunities exist to reduce the gas consumption:

1. The function `EulerSwapPeriphery.binarySearch()` starts off with a low-to-high range of `2^112`, therefore performing `112` iterations before converging. Each iteration comprises a `STATICCALL` to the `EulerSwap` instance. The number of iterations could be brought down by refining the initial `low` and `high` guesses.

Acknowledged:

Euler said:

> We have several ideas on how to improve the quoting efficiency, but are not going to make the change at this stage

7.4 Periphery Quoting Functions Can Be Frontrun

Informational Version 1 Risk Accepted

CS-EULSWP-010

On-chain execution of functions `quoteExactInput()` and `quoteExactOutput()` of `EulerSwapPeriphery` can be front-run, such that the functions return arbitrary amounts. If used on-chain, this functions should be used together with an additional slippage protection mechanism (on-chain oracle querying, or passing an off-chain quote as parameter).

The documentation of the functions does not mention the slippage risk of using on-chain quotes.

Risk accepted:

Euler said:

> This seems to be a general property common to many different AMMs. Since we expect our users to be primarily sophisticated actors such as aggregators, we assume they are already familiar with on-chain slippage checks and other best practices

7.5 Pool Can't Be Emptied

Informational **Version 1** **Risk Accepted**

CS-EULSWP-011

Due to the way the function `f()` is computed — with `xt` being at the denominator of the second parenthesised addend — the pool can never be emptied, even if `c = 1` (resulting in the bonding curve being a straight line), because `xt = 0` would make the arithmetic division revert.

Risk accepted:

Euler said:

> While an interesting observation, we don't believe this will have any real-world impact

7.6 Swaps Can Be Censored at No Cost

Informational **Version 1** **Risk Accepted**

CS-EULSWP-014

Transactions containing swaps can be censored (forced to revert) by removing all the `cash` from the output token vault, such that `vault.withdraw()` and `vault.borrow()` fail. An attacker can frontrun a transaction containing a swap with their own transaction that borrows all the cash, and then backrun the target transaction with a transaction repaying the debt. Since this all happens in a block, the attacker pays no interest on the loan. However, since this happens in separate transactions, the attacker needs a large capital to collateralize the loan (can't flashloan).

Risk accepted:

Euler said:

> This is an interesting observation, but we don't expect this to be a serious issue due to the lack of direct incentives to perform the censorship.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Bonding Curves Should Not Pass by the "Third Quadrant"

Note Version 1

The EVC mandates that an account have at most one active controller at the end of a batch: this means that, "at rest", one cannot be in debt in more than one vault. However, for some special pool configurations, it is possible for a swap to leave the pool indebted in both vaults (and thus actually revert in the end), leading to a section of the bonding curve being completely unreachable. The `EulerSwap` constructor does not check against such misconfigurations.

For this to happen, the bonding curve should pass by the third quadrant of the "real reserves" axes; by the convexity of the curve, this means that the tangent at `(x0,y0)` (which has slope equal to `-px/py`), also intersects those axes at some negative value; this, in turn, means that the pool starts off with a debt of one of the two tokens, and the LTV of that debt is greater than 1.

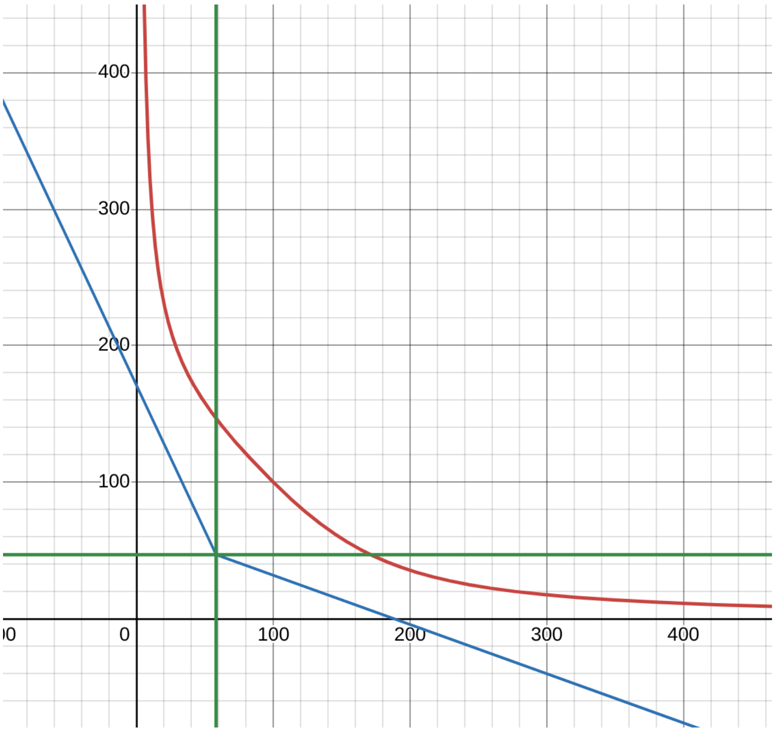
In the case where the Euler swap account exclusively enables the two pool tokens as collateral, this is impossible. However, should the owner decide to enable a third collateral, the seemingly "uncollateralized" initial debt could actually be covered, thus enabling the creation of a pool whose bonding curve has inaccessible regions.

8.2 LTV Checks

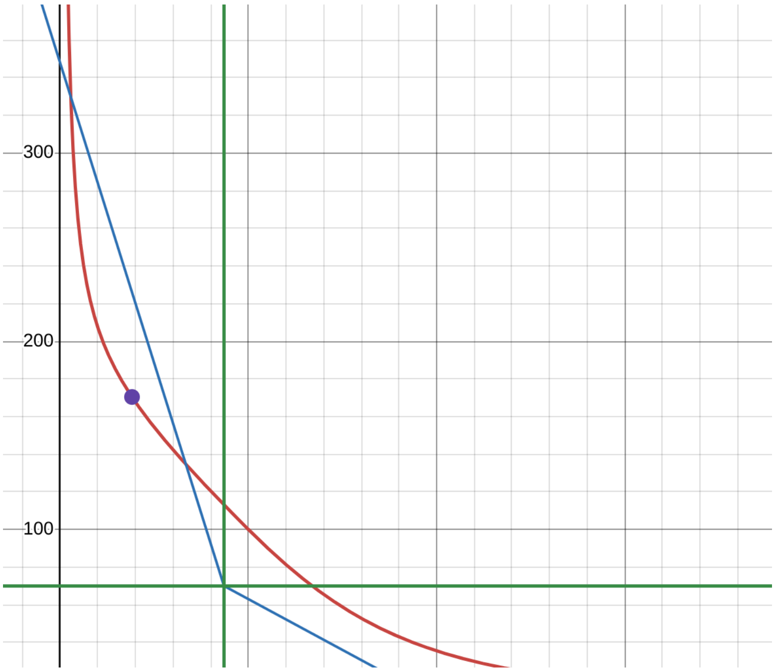
Note Version 1

When serving swaps by borrowing, the pool might hit an LTV limit, making the whole swap revert. At a certain "true price" p_r, the region of accessible points, in the frame of reference of the "real reserves" axes, is given by $y > -l_1 * p_r * x, x > 0$ and $x > -l_2/p_r * y, y > 0$, with l_1 and l_2 being the two maximum borrow LTVs. If these delimiting lines intercept the bonding curve, they identify the section of the curve that is inaccessible.

At pool-creation time, one can choose parameters such that the curve is entirely enclosed within the permissible region, for $p_r = p_x/p_y$. Graphically, this would look like the following:



However, no matter the parametrization, there always exist prices p_r, sufficiently far from the initial price p_x/p_y, such that the "delimiting lines" do intersect one of the two arms of the bonding curve. Furthermore, after a certain threshold, the "inaccessible" section of the bonding curve contains the (unique) point whose spot price is p_r. Therefore, the pool cannot effectively track arbitrary price movements. This is depicted in the following picture (the purple point being the point at price p_r):



8.3 Operations Within EVC Batch

Note Version 1

It is disadvised for admins to create their pool in the middle of an existing EVC batch involving the two vaults of that pool, as this could lead to its parametrization being based on incorrect, "temporary" values reported by `myBalance()` and, especially, `myDebt()`.

For the same reason, it is disadvised to request a quote to `EulerSwapPeriphery` in the middle of a batch, because the quoting functions rely on the current debt of the Euler account.

8.4 Quoting Functions Don'T Check LTV Limits

Note Version 1

The function `EulerSwapPeriphery.computeQuote()` checks for several cases where the amounts would be incompatible with the parameters of the pool and the vault. However, it does not check that the provided swap wouldn't fail due to LTV checks.