EulerSwap Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

BladeSec

ChainDefenders (0x539 & PeterSR)

May 26, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

EulerSwap is a novel AMM design developed by the Euler team. It allows users of Euler vaults to utilize their vault positions as liquidity providers for the EulerSwap AMM. The AMM supports a customizable curve that can behave like either a constant product or a constant sum AMM. Each deployed instance functions both as a standalone Uniswap V2-style AMM and as a hook compatible with Uniswap V4.

4.1 Actors and Roles

- **1. Actors:**
  - **Euler team:** Deploys and maintains the `EulerSwapFactory`, and configures the protocol fee and its recipient.
  - **Euler vault users:** Enable their vault positions to act as LPs for new EulerSwap instances.
  - **Traders:** Interact with the protocol by trading via EulerSwap.
- **2. Roles:**
  - `EulerSwapFactory` **Owner:** Can set a new protocol fee and designate a new protocol fee recipient. Each `EulerSwap` instance is deployed with an immutable fee, so changes only affect newly deployed instances.
  - `EulerSwap` **deployer:** Uses their Euler vault position to deploy a new `EulerSwap` instance with a chosen curve configuration. Earns fees as shares in the vault.

4.2 Key Components

- `EulerSwapFactory`: A permissionless factory that allows any Euler vault holder to deploy an `EulerSwap` AMM as an operator on their vault.

- `EulerSwap`: The AMM instance. Trades using a customizable curve and liquidity sourced from Euler vaults. Supports borrowing, depositing, and withdrawing from the vault. Operates both as a standalone Uniswap V2-style AMM and as a Uniswap V4 hook with a custom curve.

5 Audit Scope

```
EulerSwap.sol
EulerSwapFactory.sol
EulerSwapPeriphery.sol
UniswapHook.sol
CtxLib.sol
CurveLib.sol
FundsLib.sol
QuoteLib.sol
MetaProxyDeployer.sol
ProtocolFee.sol
```

6 Executive Summary

Over the course of 16 days, the Cyfrin team conducted an audit on the EulerSwap smart contracts provided by Euler. In this period, a total of 7 issues were found.

During the audit, two low-severity findings were identified. One concerned the risk of the protocol fee recipient address becoming unavailable or compromised. The other related to the potential for liquidity providers to lose collateral value due to external factors, which could in turn place their vault positions at risk.

The codebase was well written and thoughtfully structured. Each component was thoroughly tested and accompanied by clear documentation.

An additional commit, `c1c375f`, was reviewed after the conclusion of the engagement, and no issues were found.

Summary

Project Name	EulerSwap
Repository	euler-swap
Commit	1022c0bb3c03...
Audit Timeline	May 5th - May 9h, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	2
Informational	3
Gas Optimizations	2
Total Issues	7

Summary of Findings

[L-1] Protocol Fee Recipient Updates Are Not Enforced On Old EulerSwap Instances	Acknowledged
[L-2] Configured reserves may not guarantee protection against liquidation	Acknowledged
[I-1] Unused imports and errors	Resolved
[I-2] Lack of events emitted on state changes	Resolved
[I-3] Lack of input validation	Acknowledged
[G-1] Unnecessary extra storage reads when swapping	Acknowledged
[G-2] Vault calls can be done directly to EVC	Acknowledged

7 Findings

7.1 Low Risk

7.1.1 Protocol Fee Recipient Updates Are Not Enforced On Old EulerSwap Instances

Description: The `protocolFeeRecipient` value is stored in the parameters of each `EulerSwap` instance at the time of its deployment. When the `ProtocolFee::setProtocolFeeRecipient` function in the `ProtocolFee` contract is called to update the `protocolFeeRecipient`, the change only affects new instances of `EulerSwap` deployed after the update. Existing `EulerSwap` instances retain the old `protocolFeeRecipient` value since it is embedded in their parameters during deployment and is not dynamically referenced.

Impact: This creates a discrepancy where older `EulerSwap` instances continue to send protocol fees to the outdated recipient address, potentially leading to financial losses or mismanagement of funds. It also introduces operational complexity, as the protocol owner must manually update or redeploy affected `EulerSwap` instances to align them with the new recipient address.

Proof of Concept: Add the following test to the `HookFees.t.sol` file:

```
function test_protocolFeeRecipientChange() public {
    // Define two protocol fee recipients
    address recipient1 = makeAddr("recipient1");
    address recipient2 = makeAddr("recipient2");

    // Set initial protocol fee and recipient in factory
    uint256 protocolFee = 0.1e18; // 10% of LP fee
    eulerSwapFactory.setProtocolFee(protocolFee);
    eulerSwapFactory.setProtocolFeeRecipient(recipient1);

    // Deploy pool1 with recipient1
    EulerSwap pool1 = createEulerSwapHookFull(
        60e18,
        60e18,
        0.001e18,
        1e18,
        1e18,
        0.4e18,
        0.85e18,
        protocolFee,
        recipient1
    );

    // Verify pool1's protocolFeeRecipient
    IEulerSwap.Params memory params1 = pool1.getParams();
    assertEq(params1.protocolFeeRecipient, recipient1);

    // Perform a swap in pool1
    uint256 amountIn = 1e18;
    assetTST.mint(anyone, amountIn);
    vm.startPrank(anyone);
    assetTST.approve(address(minimalRouter), amountIn);
    bool zeroForOne = address(assetTST) < address(assetTST2);
    minimalRouter.swap(pool1.poolKey(), zeroForOne, amountIn, 0, "");
    vm.stopPrank();

    // Check that fees were sent to recipient1
    uint256 feeCollected1 = assetTST.balanceOf(recipient1);
    assertGt(feeCollected1, 0);
    assertEq(assetTST.balanceOf(recipient2), 0);

    // Change protocolFeeRecipient to recipient2 in factory
    eulerSwapFactory.setProtocolFeeRecipient(recipient2);
    // Perform another swap in pool1
    assetTST.mint(anyone, amountIn);
    vm.startPrank(anyone);
    assetTST.approve(address(minimalRouter), amountIn);
    minimalRouter.swap(pool1.poolKey(), zeroForOne, amountIn, 0, "");
    vm.stopPrank();

    // Check that additional fees were sent to recipient1, not recipient2
    uint256 newFeeCollected1 = assetTST.balanceOf(recipient1);
    assertGt(newFeeCollected1, feeCollected1); // recipient1 received more fees
    assertEq(assetTST.balanceOf(recipient2), 0); // recipient2's balance unchanged
}
```

Recommended Mitigation: Refactor the `EulerSwap` contract to reference the `protocolFeeRecipient` dynamically from the `EulerSwapFactory` contract instead of storing it in the deployment parameters. This ensures that any updates to the `protocolFeeRecipient` are immediately reflected across all `EulerSwap` instances, both old and new.

Euler: Acknowledged.

7.1.2 Configured reserves may not guarantee protection against liquidation

Description: According to the EulerSwap whitepaper:

> The space of possible reserves is determined by how much real liquidity an LP has and how much debt their operator is allowed to hold. Since EulerSwap AMMs do not always hold the assets used to service swaps at all times, they perform calculations based on virtual reserves and debt limits, rather than on strictly real reserves. Each EulerSwap LP can configure independent virtual reserve levels. These reserves define the maximum debt exposure an AMM will take on. Note that the effective LTV must always remain below the borrowing LTV of the lending vault to prevent liquidation.

This implies that, under proper configuration, an AMM should not be at risk of liquidation due to excessive loan-to-value (LTV). However, external factors can still undermine this guarantee.

For example, if another position in the same collateral vault is liquidated and leaves behind bad debt, the value of the shared collateral used for liquidity could drop. This would affect the effective LTV of all positions using that vault, including those managed by the EulerSwap AMM.

An attacker could exploit this situation by initiating a swap that pushes the AMM's position right up to the liquidation threshold, leveraging the degraded collateral value caused by unrelated bad debt.

Recommended Mitigation: Consider enforcing an explicit LTV check after each borrowing operation, and introduce a configurable maximum LTV parameter in `EulerSwapParams`. Additionally, clarify in the documentation that Euler account owners are responsible for monitoring the health of their vaults and should take proactive steps if the collateral accrues bad debt or drops in value—since this can happen independently of swap activity.

Euler: Acknowledged. Doing a health computation at the end of a swap would cost too much gas.

Cyfrin: The Euler team added several points in their documentation regarding liquidation concerns in PR#93

7.2 Informational

7.2.1 Unused imports and errors

Description: The following imports are unused:

- `IEVC` and `IEVault`, `EulerSwapPeriphery.sol#L6-L7`

And the following error is unused:

- `InvalidQuery`, `EulerSwapFactory.sol#L36`

Consider removing them.

Euler: Fixed in commit `6109f53`

Cyfrin: Verified.

7.2.2 Lack of events emitted on state changes

Description: Both `ProtocolFee::setProtocolFee` and `ProtocolFee::setProtocolFeeRecipient` changes state for the `EulerSwapFactory` however no events are emitted.

Consider emitting events from these functions for better off-chain tracking and transparency.

Euler: Fixed in commit `05a9148`

Cyfrin: Verified.

7.2.3 Lack of input validation

Description: In the constructor for `EulerSwapFactory` there's no sanity check that the addresses for `evkFactory_`, `eulerSwapImpl_`, and `feeOwner_` aren't `address(0)` which is a simple mistake to make.

Consider validating that these are not `address(0)`.

For example:

```
require(evkFactory_ != address(0), "Zero address");
require(eulerSwapImpl_ != address(0), "Zero address");
require(feeOwner_ != address(0), "Zero address");
```

Euler: Acknowledged.

7.3 Gas Optimization

7.3.1 Unnecessary extra storage reads when swapping

Description: When a swap is executed, whether via EulerSwap or Uniswap V4, a reentrant hook is employed:

UniswapHook::nonReentrantHook:

```
modifier nonReentrantHook() {
    {
        CtxLib.Storage storage s = CtxLib.getStorage();
        require(s.status == 1, LockedHook());
        s.status = 2;
    }

    _;

    {
        CtxLib.Storage storage s = CtxLib.getStorage();
        s.status = 1;
    }
}
```

Here, `CtxLib.getStorage()` is called once to load the storage struct and set `status` to `2`, then again afterward to restore `status` back to `1`.

Later in the swap flow, the same storage slot is reloaded multiple times:

1. In QuoteLib::calcLimits:

```
function calcLimits(IEulerSwap.Params memory p, bool asset0IsInput) internal view returns
↪  (uint256, uint256) {
    CtxLib.Storage storage s = CtxLib.getStorage();
    ...
}
```

2. In QuoteLib::findCurvePoint:

```
function findCurvePoint(IEulerSwap.Params memory p, uint256 amount, bool exactIn, bool
↪  asset0IsInput)
    internal
    view
    returns (uint256 output)
{
    CtxLib.Storage storage s = CtxLib.getStorage();
    ...
}
```

3. And again in UniswapHook::_beforeSwap just before updating reserves:

```
CtxLib.Storage storage s = CtxLib.getStorage();
```

Each `getStorage()` call emits an extra SLOAD, increasing the overall gas cost of the swap.

Consider embedding the non-reentrant logic directly within `_beforeSwap`, fetch storage only once, and cache the reserve values for use in the CurveLib calls. For example:

```
function _beforeSwap(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams calldata params,
    bytes calldata
)
    internal
    override
    returns (bytes4, BeforeSwapDelta, uint24)
{
    IEulerSwap.Params memory p = CtxLib.getParams();

    // Single storage load and reentrancy guard
    CtxLib.Storage storage s = CtxLib.getStorage();
    require(s.status == 1, LockedHook());
    s.status = 2;

    // Cache reserves locally
    uint112 reserve0 = s.reserve0;
    uint112 reserve1 = s.reserve1;

    // ... perform limit and curve-point calculations using reserve0/reserve1 ...

    // Update reserves and release guard
    s.reserve0 = uint112(newReserve0);
    s.reserve1 = uint112(newReserve1);
    s.status = 1;

    return (BaseHook.beforeSwap.selector, returnDelta, 0);
}
```

By doing so, you eliminate redundant storage reads, reducing gas consumption on every swap. Same goes for the `EulerSwap::swap` flow as well.

Euler: Acknowledged.

7.3.2 Vault calls can be done directly to EVC

Description: In `FundsLib::depositAssets`, two calls are made to the Euler Vault:

- Line 92:

```
uint256 repaid = IEVault(vault).repay(amount > debt ? debt : amount, p.eulerAccount);
```

- Line 104:

```
try IEVault(vault).deposit(amount, p.eulerAccount) {}
```

Both of these function calls are routed through the Ethereum Vault Connector (EVC) via the `callThroughEVC` modifier defined in the Vault:

- `EVault::repay`:

```
function repay(uint256 amount, address receiver) public virtual override callThroughEVC
↪   use(MODULE_BORROWING) returns (uint256) {}
```

- `EVault::deposit`:

```
function deposit(uint256 amount, address receiver) public virtual override callThroughEVC
↪   use(MODULE_VAULT) returns (uint256) {}
```

Each call incurs the cost of a contract jump due to the indirection through the Vault contract. To reduce this overhead, these operations can instead be invoked directly on the EVC, as is already done for other vault interactions within `FundsLib`.

Euler: Acknowledged.