Electisec EVK-periphery PR#58 and PR#233 - adaptive irm Review

Review Resources:

- EVK-periphery PR#58
- EVK-periphery PR#233

Auditors:

- HHK
- Panda
- Adriro

Table of Contents

Review Summary

Euler PRs review

This review covers the pull requests #58 and #233 to the EVK-periphery codebase.
They provide an adaptive IRM contract inspired by Morpho's IRM. This IRM updates the borrowing rates by shifting the curve over time depending on how far the vault is from the target utilization.

The initial review of the contracts was conducted in August 2024 as part of pull request #58.
Following the disclosure of findings, the Euler team placed the IRM codebase on standby.
A second review took place in February 2025 under pull request #233, incorporating fixes from the initial assessment and additional modifications. To differentiate between findings from the two reviews, each issue is labeled with its original date in the title.

Review Details

PR #58 of the EVK-periphery repository was reviewed over two days, from August 26th to August 27th, 2024. The review, conducted by two auditors, focused on the latest commit: 52a2572ed6f23fa4e0b293d0cc2d77d8ce7f1f8c.
PR #233 of the EVK-periphery repository was reviewed over two days, from February 4th to February 5th, 2025. The review, conducted by two auditors, focused on the latest commit: 21f0cfdad33774d76aac77083926f0381f1dbabe.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src/IRM/IRMAdaptiveCurve.sol
src/IRM/lib/ExpLib.sol
src/IRMFactory/EulerIRMAdaptiveCurveFactory.sol
```

After the findings were presented to the Euler team, fixes were made and included in the existing PRs or inside new PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

Electisec and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. Electisec and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Euler and users of the contracts agree to use the code at their own risk.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - 02-2025 - Interest Rate Manipulation Through Delayed Rate Updates

The rate applied after interaction with delay can be exploited to borrow most of the vault without paying the corresponding interest.

Technical Details

The interest rate adjustment in `IRMAdaptiveCurve` has a timing issue because it uses the previous utilization state to calculate rate changes. This creates a window for manipulation:

1. Borrow heavily (e.g. 99% utilization)
2. Wait for a transaction that would trigger an unfavorable rate update
3. Quickly repay everything after the high rate is applied to pay effectively none of the new interest rate
4. Immediately reborrow at 99% utilization with the new lower rate

The root cause is that rate updates use `lastUtilization` instead of current utilization, creating a one-transaction delay in rate adjustments.

```
// Uses lastUtilization from previous state
int256 lastUtilization = irState[vault].lastUpdate == 0 ? utilization :
irState[vault].lastUtilization;
(uint256 rate, uint256 rateAtTarget) = computeInterestRateInternal(vault,
lastUtilization);

// Updates state with current utilization for next interaction
irState[vault] = IRState(uint144(rateAtTarget), int64(utilization),
uint48(block.timestamp));
```

Impact

High - Allows systematic avoidance of interest rate mechanisms.

Recommendation

Consider keeping the old utilization rate for the curve shifting but using the new utilization rate when determining the position on the curve. If the user borrows 99% of the supply, he will experience the rate at 99% of the curve.

Similar to the suggestion from medium#1.

Developer Response

Fixed in `b279f6572ee0d4f21a899b0882b07ece231daaaa` and `53a192fd1d5eb206e2b3a285bd92a9dd8da94519`.

Medium Findings

1. Medium - 08-2024 - Difference in utilization rate leads to Incorrect curve shifting

The Morpho protocol and the EVK have different architectures and invoke the IRM at various stages in their lifecycles. As a result, when the EVK calls the IRM, it will apply an incorrect utilization rate when adjusting the curve, leading to discrepancies between the expected rate and the actual rate over the long term.

Technical Details

In the Morpho protocol, the IRM is called before the user action. It calculates the interest since the last call according to the utilization during that time and applies it to the debt before the user action.

In the EVK, the IRM is called at the end of the user action. It calculates the interest rate and saves it in storage. On the next call to the contract, it will apply the interest to the debt before the user's action.

As we can see, the IRM is not invoked at the same time. In the Morpho case, it will be invoked before the new action, while in the EVK cas,e it will be invoked after the new action.

The EVK calls the IRM with the new utilization rate to properly get the rate to save in storage, which differs from Morpho, which would use the old utilization rate.

This results in the curve shifting less or more than it should compared to when it's used with the Morpho protocol. Over time, the interest rates will differ.

POC:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import {Test, console} from "forge-std/Test.sol";
import {IRMAdaptiveCurve} from "../../../src/IRM/IRMAdaptiveCurve.sol";
import {ExpLib} from "../../../src/IRM/lib/ExpLib.sol";
import {
    AdaptiveCurveIrm,
    ConstantsLib,
    ExpLib as ExpLibRef,
    Id,
    Market,
    MarketParams,
    MarketParamsLib
} from "morpho-blue-irm/adaptive-curve-irm/AdaptiveCurveIrm.sol";

contract IRMAdaptiveCurveAuditTest is Test {
    using MarketParamsLib for MarketParams;

    address internal constant VAULT = address(0x1234);
    address internal constant MORPHO = address(0x2345);
    uint256 internal constant NUM_INTERACTIONS = 50;

    function test_Morpho() public {
        AdaptiveCurveIrm irmRef = new AdaptiveCurveIrm(MORPHO);

        uint256 rate;
        vm.startPrank(MORPHO);

        // t = 0
        MarketParams memory marketParams;
```

```
        Market memory market;
        market.lastUpdate = uint128(block.timestamp);
        market.totalSupplyAssets = 0;
        market.totalBorrowAssets = 0;

        rate = irmRef.borrowRate(marketParams, market);
        console.log("T=0:", rate);

        // t = 1 supply
        skip(1 days);

        rate = irmRef.borrowRate(marketParams, market);
        console.log("T=1:", rate);

        market.lastUpdate = uint128(block.timestamp);
        market.totalSupplyAssets = 100;
        market.totalBorrowAssets = 0;

        // t = 2 borrow
        skip(7 days);

        rate = irmRef.borrowRate(marketParams, market);
        console.log("T=2:", rate);

        market.lastUpdate = uint128(block.timestamp);
        market.totalSupplyAssets = 100;
        market.totalBorrowAssets = 95;

        // t = 3 accrue
        skip(1 days);

        rate = irmRef.borrowRate(marketParams, market);
        console.log("T=3:", rate);

        market.lastUpdate = uint128(block.timestamp);

        // t = 4 unwind
        skip(1 days);

        rate = irmRef.borrowRate(marketParams, market);
```

```
        console.log("T=4:", rate);

        market.lastUpdate = uint128(block.timestamp);
        market.totalSupplyAssets = 100;
        market.totalBorrowAssets = 0;

        // t = 5 accrue
        skip(1 days);

        rate = irmRef.borrowRate(marketParams, market);
        console.log("T=5:", rate);

        market.lastUpdate = uint128(block.timestamp);
    }

    function test_Euler() public {
        IRMAdaptiveCurve irm = new IRMAdaptiveCurve(
            ConstantsLib.TARGET_UTILIZATION,
            ConstantsLib.INITIAL_RATE_AT_TARGET,
            ConstantsLib.MIN_RATE_AT_TARGET,
            ConstantsLib.MAX_RATE_AT_TARGET,
            ConstantsLib.CURVE_STEEPNESS,
            ConstantsLib.ADJUSTMENT_SPEED
        );

        uint256 rate;
        vm.startPrank(VAULT);

        // t = 0 init
        rate = irm.computeInterestRate(VAULT, 0, 0);
        console.log("T=0:", rate);

        // t = 1 supply
        skip(1 days);

        rate = irm.computeInterestRate(VAULT, 100, 0);
        console.log("T=1:", rate);

        // t = 2 borrow
        skip(7 days);
```

```
        rate = irm.computeInterestRate(VAULT, 5, 95);
        console.log("T=2:", rate);

        // t = 3 accrue
        skip(1 days);

        rate = irm.computeInterestRate(VAULT, 5, 95);
        console.log("T=3:", rate);

        // t = 4 unwind
        skip(1 days);

        rate = irm.computeInterestRate(VAULT, 100, 0);
        console.log("T=4:", rate);

        // t = 5 accrue
        skip(1 days);

        rate = irm.computeInterestRate(VAULT, 100, 0);
        console.log("T=5:", rate);
    }
```

Impact

Medium.

Recommendation

Consider saving the previous utilization rate and using it to shift the curve. Keep using the new utilization rate to get the position on the curve and the corresponding interest rate to save in storage.

Developer Response

08-2024 - Partially fixed in `a4d7673eb0d854a72cde12fff8ccfbb8312125c6`.

02-2025 - Fully fixed in `b279f6572ee0d4f21a899b0882b07ece231daaaa` and `53a192fd1d5eb206e2b3a285bd92a9dd8da94519`.

Low Findings

1. Low - 08-2024 - Subsequent call to the IRM could clear the effect of the average rate

Unlike the Morpho case, where interest accrual is skipped if the elapsed time is zero, EVK vaults always call the IRM during checks at the end of each set of operations. A subsequent call to the IRM could unintentionally nullify the effect of the *average rate* and overwrite it with the stored *end rate*.

Technical Details

The implementation of IRMAdaptiveCurve.sol returns the *average rate* as the interest rate and stores the *end rate* for future use.

When the `linearAdaptation` parameter (a combination of speed and elapsed time) is zero, the function returns the start rate. This start rate is derived from the stored rate, the end rate from the previous call.

```
167:             int256 speed = ADJUSTMENT_SPEED * err / WAD;
168:
169:             // Calculate the adaptation parameter.
170:             int256 elapsed = int256(block.timestamp - state.lastUpdate);
171:             int256 linearAdaptation = speed * elapsed;
172:
173:             // If linearAdaptation == 0, avgRateAtTarget = endRateAtTarget =
startRateAtTarget.
174:             if (linearAdaptation == 0) {
175:                 avgRateAtTarget = startRateAtTarget;
176:                 endRateAtTarget = startRateAtTarget;
```

If the EVK vault calls the IRM a second time within the same block, the subtraction `block.timestamp - state.lastUpdate` will be zero. This causes the `linearAdaptation` parameter to be zero, resulting in the function returning the stored rate as the interest rate.

Impact

Low. The end rate could accidentally overwrite the average rate during a subsequent call to the IRM within the same block.

Recommendation

Early exit if the elapsed time is null.

Developer Response

Fixed in `a4d7673eb0d854a72cde12fff8ccfbb8312125c6`.

Gas Saving Findings

1. Gas - 08-2024 - Calculations that depend on immutables can be precomputed

Several intermediate calculations depend on a combination of immutable and constant variables and can be precomputed at construction time.

Technical Details

- IRMAdaptiveCurve.sol#L151: `WAD - TARGET_UTILIZATION`
- IRMAdaptiveCurve.sol#L198: `WAD - WAD * WAD / CURVE_STEEPNESS`
- IRMAdaptiveCurve.sol#L198: `CURVE_STEEPNESS - WAD`

Impact

Gas savings.

Recommendation

Precompute these calculations as immutable variables.

Developer Response

Acknowledged.

Informational Findings

1. Informational - 08-2024 - `TARGET_UTILIZATION` should be greater than zero

The current sanity checks allow a zero target utilization, but that would cause the IRM always to revert.

Technical Details

Configuring a zero value for the `TARGET_UTILIZATION` causes a revert due to a division by zero while calculating the error.

```
151:         int256 errNormFactor = utilization > TARGET_UTILIZATION ? WAD -
TARGET_UTILIZATION : TARGET_UTILIZATION;
152:         int256 err = (utilization - TARGET_UTILIZATION) * WAD / errNormFactor;
```

Impact

Informational.

Recommendation

Validate the target utilization parameter is above zero.

```
-   if (_TARGET_UTILIZATION < 0 || _TARGET_UTILIZATION > 1e18) {
+   if (_TARGET_UTILIZATION <= 0 || _TARGET_UTILIZATION > 1e18) {
        revert InvalidParams();
    }
```

Developer Response

Fixed in `d032cd90d5a23fe9a090895242175e4837db0967`.

2. Informational - 08-2024 - Useless `_MIN_RATE_AT_TARGET > _MAX_RATE_AT_TARGET` in the `constructor()`

Technical Details

In the constructor there is a check `_MIN_RATE_AT_TARGET > _MAX_RATE_AT_TARGET` on line 85.

This check is not needed as there is already a check on line 79 that checks that `_MIN_RATE_AT_TARGET >= _INITIAL_RATE_AT_TARGET` & `_INITIAL_RATE_AT_TARGET <= _MAX_RATE_AT_TARGET` so that means that `_MIN_RATE_AT_TARGET` cannot be greater than `_MAX_RATE_AT_TARGET`.

Impact

Informational.

Recommendation

Remove this check.

Developer Response

Fixed in `d032cd90d5a23fe9a090895242175e4837db0967`.

3. Informational - 02-2025 - Make the curve shifting start only after a certain time

Technical Details

The curve-shifting system needs regular interactions with the vault to become efficient and to reflect accurately the rate the market is willing to pay.

When a new liability vault is created, it might not have a lot of usage at the beginning until the TVL rises and users notice this new vault.

It could be interesting to add an immutable variable that would store the duration until the curve shifting shouldn't happen, and only the base curve should be used (e.g. `BOOTSTRAP_END = block.timestamp + delay`).

This value could be, for example 1 week. During that time, no matter the utilization, the curve would stay unchanged. After that time, the curve would finally start shifting up or down.

This would protect the IRM from being manipulated at vault creation by very low deposit/borrow, making it shift a lot and delaying the actual usage of the vault since users won't be willing to use it as the interest will be too low or too high.

Impact

Informational.

Recommendation

Consider adding a bootstrap period before the curve shifting starts.

Developer Response

Acknowledged.

Final remarks

The adaptive IRM is a valuable addition to the EVK. Battle-tested by Morpho, it will help determine the optimal rate the market is willing to pay. As noted by Euler, it is recommended to use this IRM only on vaults with high activity. Due to the architectural differences between Euler and Morpho, the curve shifting may not be as accurate if the vault is not frequently utilized.