Euler Money Market Review

August 5, 2024

Fixes reviewed September 8, 2024

Prepared for Contango

Conducted by Richie Humphrey (devtooligan)

About the Contango Euler Money Market Review

Contango builds perps by automating a looping strategy, also known as recursive borrowing and lending. This is achieved using spot and money markets.

This review focused on the `EulerMoneyMarket` contract, a new adapter for Contango which enables depositing, borrowing, and claiming rewards with Euler.

About Offbeat Security

Offbeat Security is a boutique security company providing custom solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Summary & Scope

The euler (https://github.com/contango-xyz/core-v2-private/tree/fa6841a4d5e602c7f7ae3aecadd31fb7b9122290/src/moneymarkets/euler) folder of the `core-v2-private` repo was reviewed at commit fa6841a4d5e602c7f7ae3aecadd31fb7b9122290.

The following three contracts were in scope:

- /src/moneymarkets/euler/EulerMoneyMarket.sol
- /src/moneymarkets/euler/EulerReverseLookup.sol
- /src/moneymarkets/euler/EulerRewardsOperator.sol

Summary of Findings

Identifier	Title	Severity	Fixed
H-01	Unchecked return value allows vault calls with amount of type(uint256).max	High	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): All return values checked and properly returned.
L-01	Removed tokens may have rewards claimable	Low	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): While technically this is still the case, `enabled` tokens are now decoupled from `live` tokens and tokens may now be disabled without being `live`
L-02	Removed tokens may be left enabled on the reward stream	Low	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): `enabled` tokens decoupled from `live` tokens
I-01	Claim allowed on zero amount	Info	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): Not allowed
I-02	Spurious events emitted for enable/disable reward	Info	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): Events emitted correctly.
I-03	MAX_REWARDS_ENABLED does not need to be checked during `addReward`	Info	Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5): No longer applicable since `enabled` tokens are now decoupled from `live` tokens

Code Quality

- In `RewardsOperator` , the internal storage variable `_enabledTokens` does not represent tokens that are enabled on the reward stream. It represents tokens that have been added to the set that may or may not be enabled on the reward stream. Consider changing the name of this variable to something else such as `allowedTokens` .

Centralization

- As a result of a discussion with the team about privileged roles, a decision was made to remove the ability for a `DELEGATE` to claim rewards since there is no strong use case for this workflow. This reduces the centralization risk.

Code Assumptions

Assumptions noted:

1. Native tokens will not be used to interact with Euler.

2. Euler vaults are trusted with uncanceled, maximum approvals.

~~**3.** There is a use case for storing added tokens that may or may not be enabled on the reward stream.~~ Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5)

~~**4.** It is not problematic that events are emitted spuriously for certain actions such as claim, enable reward, and disable reward.~~ Fixed in PR#5 (https://github.com/contango-xyz/core-v2/pull/5)

Detailed Findings

High Findings

[H-01] Unchecked return value allows vault borrows with amount of type(uint256).max

Euler uses the `type(uint256).max` value as an indication that the user wants to use the maximum amount possible for the transaction.

When `type(uint256).max` is passed as the `amount` in the borrow function (https://github.com/euler-xyz/euler-vault-kit/blob/20973e1dd2037d26e8dea2f4ab2849e53a77855e/src/EVault/modules/Borrowing.sol#L65-L78) on a Euler vault, the amount will be adjusted to use the value of the `vault.cash` .

```
    function borrow(uint256 amount, address receiver) public virtual nonReentran
        (VaultCache memory vaultCache, address account) = initOperation(OP_BORRO

====>> Assets assets = amount == type(uint256).max ? vaultCache.cash : amount.t
        if (assets.isZero()) return 0;

        if (assets > vaultCache.cash) revert E_InsufficientCash();

        increaseBorrow(vaultCache, account, assets);

        pushAssets(vaultCache, receiver, assets);

        return assets.toUint();
    }
```

If the vault.cash is 0, then the `borrow` function will return 0.

However the `EulerMoneyMarket._borrow` function ignores the return value and returns the original amount passed in by the user.

```
    function _borrow(PositionId positionId, IERC20, uint256 amount, address to)
        reverseLookup.quote(positionId).borrow(actualAmount = amount, to);
    }
```

This means that if `_borrow` is called with an `amount` of uint max, the function will not revert and will instead process a borrow for whatever the vault's `cash` is.

Note: The `_deposit` and `_withdraw` functions also have similar issue, but `_deposit` would revert if uint.max was passed in.

Any calls made to `_borrow` or `_withdraw` using uint max as the `amount` will result in the function succeeding and returning uint max to the caller which is problematic when the caller is depending on the return value.

Recommendation

These functions should return the return values of the vault calls, including `_deposit` which doesn't represent a current security risk but should still be updated for consistency and best practices.

```diff
     function _lend(PositionId positionId, IERC20 asset, uint256 amount, address
         actualAmount = asset.transferOut(payer, address(this), amount);
-        reverseLookup.base(positionId).deposit(actualAmount, address(this));
+        return reverseLookup.base(positionId).deposit(actualAmount, address(thi
     }

     function _borrow(PositionId positionId, IERC20, uint256 amount, address to)
-        reverseLookup.quote(positionId).borrow(actualAmount = amount, to);
+        return reverseLookup.quote(positionId).borrow(actualAmount = amount, to
     }

     function _repay(PositionId positionId, IERC20 asset, uint256 amount, addres
@@ -84,7 +84,9 @@ contract EulerMoneyMarket is BaseMoneyMarket {
     }

     function _withdraw(PositionId positionId, IERC20, uint256 amount, address t
-        reverseLookup.base(positionId).withdraw(actualAmount = amount, to, addr
+        IEulerVault vault = reverseLookup.base(positionId);
+        uint256 sharesWithdrawn = vault.withdraw(amount, to, address(this));
+        actualAmount = vault.convertToAssets(sharesWithdrawn);
     }
```

Note: For `_withdraw`, it's not as simple as returning the return value from Euler because the returned value is in shares which must be converted to assets.

Low Findings

[L-01] Removed tokens may have rewards claimable

When `removeReward` is called, the reward is removed from the `_enabledRewards` set and that reward cannot be interacted with on the rewards stream. This means that `claimReward` cannot be called on the reward even when there are claimable rewards.

NOTE: The claimable rewards are not lost. They can still be claimed if the Timelock were to add back the reward. However, the chance of overlooking the claimable rewards greatly increases since there is no longer an on-chain record of rewards that may have claimable amounts.

Recommendation

Consider calling claim on the reward before removing it.

[L-02] Removed tokens may be left enabled on the reward stream

When `removeReward` is called, the reward is removed from the `_enabledRewards` set but it is not disabled on the reward stream.

The reward stream enforces a MAXIMUM_ENABLED_REWARDS of 5 rewards.

If a reward is removed from `_enabledRewards` but left enabled on the reward stream, it will still count towards that maximum. This means that if 5 new rewards were added to the EulerRewardOperator contract, only 4 of them could be enabled on the reward stream.

Recommendation

Consider calling `disableReward` on the reward stream before removing it from the `_enabledReward` set.

Another option to consider is combining the concept of adding/removing and enabling/disabling rewards, however this comes with additional requirements such as handling the claiming of rewards for disabled tokens.

Informational Findings

[I-01] Claim allowed on zero amount

When `_claim` is called, there is no check whether that reward has a claimable amount and always emits the event RewardClaimed even if no rewards were claimed.

```
    function _claimReward(PositionId positionId, AuthorisedProxy proxy, IEulerVa
        // solhint-disable-next-line avoid-low-level-calls
        evc.call(
            address(rewardStreams),
            AuthorisedProxy.unwrap(proxy),
            0,
            abi.encodeWithSelector(IRewardStreams.claimReward.selector, vault, r
        );
        emit RewardClaimed(positionId, reward, to);
    }
```

Recommendation

Consider checking the earned rewards before calling claim on the rewards stream.

```
     function _claimReward(PositionId positionId, AuthorisedProxy proxy, IEulerV
+        if (rewardStreams.earnedReward(address(this), vault, reward, false) ==
         // solhint-disable-next-line avoid-low-level-calls
         evc.call(
             address(rewardStreams),
```

Alternatively, if no code change is made, clearly documentation to inform integrators and future development.

[I-02] Spurious events emitted for enable/disable reward

When `_enableReward` is called the `RewardEnabled` is always emitted, even if it was previously enabled on the reward stream.

In the [Euler BaseRewardStreams.enableReward function (https://github.com/euler-xyz/reward-streams/blob/0139e238ee04fcab579cd6ab7e4edfe06d27c461/src/BaseRewardStreams.sol#L262-L283)] (https://) if the reward is already contained in the `accountEnabledRewards` set, the `insert` will return false which skips the enabling logic and creates a noop since it never reverts.

~~~
    function enableReward(address rewarded, address reward) external virtual ove
        address msgSender = _msgSender();
        AccountStorage storage accountStorage = accounts[msgSender][rewarded];
        SetStorage storage accountEnabledRewards = accountStorage.enabledRewards

        if (accountEnabledRewards.insert(reward)) {
            if (accountEnabledRewards.numElements > MAX_REWARDS_ENABLED) {
                revert TooManyRewardsEnabled();
            }

            DistributionStorage storage distributionStorage = distributions[rewa
            uint256 currentAccountBalance = accountStorage.balance;

            // We pass zero as `currentAccountBalance` to not distribute rewards
            // enabled them.
            updateRewardInternal(distributionStorage, accountStorage.earned[rewa

            distributionStorage.totalEligible += currentAccountBalance;

            emit RewardEnabled(msgSender, rewarded, reward);
        }
    }```

  Upon returning control flow to the `_enableReward` function, the `RewardEnabled`

  ```solidity
    function _enableReward(PositionId positionId, AuthorisedProxy proxy, IEulerV
        // solhint-disable-next-line avoid-low-level-calls
        evc.call(
            address(rewardStreams),
            AuthorisedProxy.unwrap(proxy),
            0,
            abi.encodeWithSelector(IRewardStreams.enableReward.selector, vault,
        );
        emit RewardEnabled(positionId, reward);
    }
~~~

The same is true for `disableReward`, an event is always emitted even it it was not enabled previously.

Recommendation

Assuming this is not a significant issue, be sure to clearly document this issue to inform integrators and future development.

Alternatively, this issue could be addressed by checking if the reward is already enabled prior to making the call.

Another option to consider is combining the concept of adding/removing and enabling/disabling rewards, however this comes with additional requirements such as handling the claiming of rewards for disabled tokens.

[I-03] MAX_REWARDS_ENABLED does not need to be checked during `addReward`

Since the concept of adding a reward to the `RewardsOperator` is decoupled from the enabled rewards on the reward stream, it is not necessary to enforce a max on the unfortunately named `_enabledRewards` set.

The `_enabledRewards` set is better named `_allowedRewards` since it represents the set of reward that the contract is able to interact with the reward stream using `enableReward`, `disableReward`, and `claimReward`.

By enforcing this limit, allowed rewards are unnecessarily limited to 5, the value of the MAXIMUM_REWARDS_ENABLED constant. It may be desirable to have additional rewards allowed so that additional rewards can be claimed or so that rewards can be switched on and off without having to remove any.

Recommendation

Remove the check against `MAX_REWARDS_ENABLED` and the associated `TooManyRewards` error. Doing this will also save gas.

```
     function addReward(IEulerVault vault, address reward) external onlyRole(OPE
         EnumerableSet.AddressSet storage rewards = _enabledRewards[vault];
-        if (rewards.length() == rewardStreams.MAX_REWARDS_ENABLED()) revert Too
```