yAudit Euler Yield Aggregator Review

Review Resources:

- Whitepaper
- Low-Level Spec

Auditors:

- HHK
- Adriro

Table of Contents

Review Summary

Euler Yield Aggregator

The Yield Aggregator is an open-source protocol for permissionless risk curation on top of ERC4626 vaults(strategies). Although it is initially designed to be integrated with Euler V2 vaults, technically, it supports any other vault as long as it is ERC4626 compliant.

The Yield Aggregator is an ERC4626 vault; any risk curator can deploy one through the factory. Each vault has one loan asset and can allocate deposits to multiple strategies. The aggregator vaults are noncustodial and immutable instances that offer users an easy way to provide liquidity and passively earn yield.

The contracts of the Yield Aggregator repository were reviewed over eight days. Two auditors performed the code review between August 26th and September 4th, 2024. The repository was under active development during the review, but the review was limited to the latest commit at the start. This was commit a00602c3429cdddaa1cbfe3c741705823cd2923e for the Yield Aggregator repository.

Scope

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

```
src
├── Dispatch.sol
├── YieldAggregator.sol
├── YieldAggregatorFactory.sol
├── common
│   └── Shared.sol
├── interface
│   ├── IBalanceForwarder.sol
│   └── IYieldAggregator.sol
├── lib
│   ├── AmountCapLib.sol
│   ├── ConstantsLib.sol
│   ├── ErrorsLib.sol
│   ├── EventsLib.sol
│   ├── HooksLib.sol
│   └── StorageLib.sol
└── module
    ├── Fee.sol
    ├── Hooks.sol
    ├── Rebalance.sol
    ├── Rewards.sol
    ├── Strategy.sol
    ├── WithdrawalQueue.sol
    └── YieldAggregatorVault.sol
```

After the findings were presented to the Euler team, fixes were made and included in several 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.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit 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.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Proper access control is enforced using roles (RBAC).
Mathematics	Good	The implementation doesn't involve complex math operations.
Complexity	Average	We identified several issues stemming from the logic in the harvest and rebalance functions, which could impact general accounting and adherence to the ERC-4626 standard.
Libraries	Good	The protocol uses an up-to-date version of the OpenZeppelin library.
Decentralization	Average	Centralized functionalities are limited to the vault and strategies configuration. Wrong configuration could affect user's safety.
Code stability	Good	The codebase remained stable during the review.
Documentation	Good	The contracts are documented using NatSpec. Additional resources include the whitepaper and a specification document.
Monitoring	Good	Multiple events are emitted through the protocol life-cycle.
Testing and verification	Good	The codebase has an extensive testing suite, including unit, integration, fuzz, and invariant tests.

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

None.

Medium Findings

None.

Low Findings

1. Low - Missing `_harvest()` in `rebalance()`

The function `rebalance()` doesn't call `_harvest()`, which can lead to incorrect allocation of some strategies and loss of yield when rebalancing to 0.

Technical Details

The function `rebalance()` allows updating a list of strategies allocation according to their allocation points. If a strategy has too much funds, it will be withdrawn; if it doesn't have enough funds, it will try to move some funds from cash to the strategy.

The issue is that there is no prior call to the `_harvest()` function. This function is in charge of updating the amount allocated to each strategy, determining whether there was a positive or negative yield, and applying it.

By calling rebalancing without a prior update to the allocated amount saved in storage, the contract will assume that X amount is currently deposited, while in fact, it could be more or less depending on whether yield or bad debt was recently generated. This will lead to the wrong amount being rebalanced for each strategy.

Additionally, if a strategy allocation point were just lowered to 0, the rebalancing would result in the yield being lost as future call to `harvest()` will ignore that strategy since its allocated amount has been set to 0 which cause to return early inside `_executeHarvest()`.

Impact

Low.

Recommendation

Replace the `_gulp()` call in `rebalance()` by a `_harvest()` call.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/60 & https://github.com/euler-xyz/yield-aggregator/pull/74.

2. Low - Unable to remove strategies in emergency mode

Technical Details

The function `removeStrategy()` allows to remove strategies from the `withdrawalQueue` and reset their storage. However, it can only remove 'active' strategies, not those with the `Emergency` status.

The `Emergency` status is set when a `GUARDIAN` calls the `toggleStrategyEmergencyStatus()` function. This is only to be called when a strategy suffers a big issue and cannot be safely withdrawn from.

If a strategy is in `Emergency` status for a while, it is likely that the `STRATEGY_OPERATOR` may want to remove it permanently; however, because of the `strategyStorage.status != IYieldAggregator.StrategyStatus.Active` inside the `removeStrategy()` function he won't be able to unless he toggles it back to `active` first and rebalance it to 0 amount allocated which could be very unsafe and impossible in some case if withdrawals revert.

Impact

Low.

Recommendation

Consider allowing the removal of strategies in `Emergency` status even when they still have funds. A timelock could be implemented to allow this removal only if, for example, one month has passed.

Developer Response

Acknowledged. To give more context regarding the `Emergency` status and why we came up with it, in the initial design, a strategy operator can add/remove any strategy, even ones with an allocated amount > 0. We felt that it's not perfect from a user perspective to allow that, as it makes it easier to rug-pull and mess with funds, but we still want the yield aggregator mechanism to work as expected if there is an issue in a specific strategy. And that's why we came up with the `Emergency` status and the `toggleStrategyEmergencyStatus()` function.

Assuming a strategy is in `Emergency` status and withdrawing from it is reverting(so can't set allocation points to 0 and rebalance), the operator may only really want to remove that strategy unless they want to add a different one, but there is no more available slot in the withdrawal queue (max strategies is 10) which is a highly unlikely scenario...plus giving the other options available, like you mentioned a timelock for removing such a strategy, we see the upside of such an implementation is low compared with the code that need to be added and the complexity of the function. We will make sure to document this scenario and make yield aggregator's curators aware of it.

3. Low - Preview functions could return different results than `withdraw()` or `redeem()`

Withdrawals may experience a decrease in the share price due to debt socialization while harvesting, causing a conflict with the preview functions, which don't factor in unrealised losses.

Technical Details

Both `withdraw()` and `redeem()` execute a harvest of the strategies as part of their implementation.

An eventual drop in the share price due to losses while harvesting the strategies would result in a different outcome than what would have been returned by the preview functions. This behavior may cause a conflict with the expectations set by the ERC-4626 standard:

> previewWithdraw
> [...]
> MUST return as close to and no fewer than the exact amount of Vault shares that would be burned in a withdraw call in the same transaction. I.e. withdraw should return the same or fewer shares as previewWithdraw if called in the same transaction.
>
> previewRedeem
> [...]
> MUST return as close to and no more than the exact amount of assets that would be withdrawn in a redeem call in the same transaction. I.e. redeem should return the same or more assets as previewRedeem if called in the same transaction.

Impact

Low. The `previewWithdraw()` and `previewRedeem()` functions don't strictly follow the standard, potentially affecting integrators.

Recommendation

The preview functions would need to factor in any potential unrealised loss. Alternatively, ensure this is properly documented as it deviates from the expected standard behavior.

Developer Response

Fix in https://github.com/euler-xyz/yield-aggregator/pull/61 & https://github.com/euler-xyz/yield-aggregator/pull/73.

4. Low - Optimistic max withdrawable amount

The max withdrawable amount for the aggregator vault is based on the account's shares balance. However, the effective withdrawal amount could be limited downstream by caps implemented in the underlying strategies.

Technical Details

In the YieldAggregator vault, the `maxWithdraw()` and `maxRedeem()` functions base their result solely on the share balance held by the given account.

However, the actual implementation of `_withdraw()` limits the effective withdrawable amount to the available cash held by the vault and sum of each of the underlying strategies' max withdrawable amount.

```
323:                 uint256 underlyingBalance = strategy.maxWithdraw(address(this));
324:                 uint256 desiredAssets = _assets - assetsRetrieved;
325:                 uint256 withdrawAmount = (underlyingBalance >= desiredAssets) ?
desiredAssets : underlyingBalance;
```

Impact

Low. The `maxWithdraw()` and `maxRedeem()` functions could potentially return an amount of assets that exceeds what is currently available for withdrawal.

Recommendation

Consider applying the same logic used in the `withdraw()` and `redeem()` functions to the `maxWithdraw()` and `maxRedeem()` implementations to ensure consistency with the expected withdrawal behavior. Alternatively, ensure this is properly documented as it deviates from the expected standard behavior.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/62.

5. Low - Conditional `_harvest()` on withdrawals could lead to unexpected issues

The vault only harvests the yield when enough time has passed since the last harvesting. A positive or negative yield can result in unexpected issues.

Technical Details

When calling `withdraw()` or `redeem()` the vault will call the internal function `harvest()` with `_checkCooldown = true`.

When `_checkCooldown = true`, the `_harvest()` function will check when the last harvesting happened and will return early if it was less than `HARVEST_COOLDOWN`, currently set to 1 day. This means the user may withdraw without the harvesting executing if executed less than 24 hours ago.

The issue is that the `_harvest()` function applies any positive or negative yield. If a strategy's balance increases or decreases because of a positive or negative yield, it will update the `allocated` variable as well as the share value.

In the case of a positive yield, if the withdrawal amount is bigger than the cash amount, the contract will try to withdraw from strategies. The contract will call `strategy.maxWithdraw(address(this))` which might return a higher amount than the `allocated` variable. This could lead to the call reverting because of an underflow on line 334.

In the case of a negative yield, users will be able to withdraw at a higher price per share than they should. In the case of a very important negative yield, this could lead to a race to exit, with the last users to exit potentially unable to withdraw and more impacted by the negative yield than they should have been.

Impact
Low.

Recommendation

- Could consider always calling `_harvest()`.
- If the conditional harvesting is kept, then:
  - Consider only withdrawing the `allocated` amount or checking if the withdrawn amount is bigger than the `allocated` amount.
  - Consider checking for negative yield when looping through the strategies and applying the percentage to the amount withdrawn by the user, similar to how the Yearn v3 vault does.

Developer Response
Fixed in https://github.com/euler-xyz/yield-aggregator/pull/73

6. Low - Interests might be distributed even when `_totalSupply() < MIN_SHARES_FOR_GULP`

Technical Details

The function `gulp()` checks if the current supply is bigger than the constant `MIN_SHARES_FOR_GULP`. This is to prevent the distribution of interest if there is little to no deposit in the vault. However, the check might not fully protect against such a case.

If the supply is high enough when `_gulp()` is called but then reduced by users withdrawing from the vault, the interest will keep being distributed. These interests might be lost to the dead share resulting from `_decimalsOffset()` being 0, which would increase the share ratio by a lot.

A malicious user could donate to the vault and try to accrue the share value over time with little to no deposit, potentially resulting in future users receiving 0 shares for their deposits. This would be hard to execute and likely result in a loss for the attacker, but it is not impossible.

Impact

Low.

Recommendation

Consider moving the check `if (_totalSupply() < Constants.MIN_SHARES_FOR_GULP) return;` to the function `_interestAccruedFromCache()` and return 0.

This way, interests will not be distributed when the supply is too low.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/70.

Gas Saving Findings

1. Gas - External calls to self

There are several instances where the aggregator vault makes external calls to the `asset()` function. These external calls are unnecessary, as the `asset()` function can be safely routed internally since it has no reentrancy protection.

Technical Details

- Shared.sol#L144
- Rebalance.sol#L101
- Strategy.sol#L123

Impact

Gas savings.

Recommendation

Change the external call to use an internal jump.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/63.

2. Gas - Use unchecked math if there is no overflow risk

There are mathematical operations that can be performed using unchecked arithmetic for gas savings.

Technical Details

- Shared.sol#L51
- Rebalance.sol#L67-L68
- Rebalance.sol#L80
- Rebalance.sol#L82
- YieldAggregatorVault.sol#L433
- YieldAggregatorVault.sol#L435
- YieldAggregatorVault.sol#L469
- YieldAggregatorVault.sol#L471

Impact

Gas savings.

Recommendation

Use an unchecked block if there is no overflow or underflow risk for gas savings.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/64.

3. Gas - Use direct approval instead of increasing the allowance

Increasing the token allowance amount is not strictly necessary and requires an extra call to read the current allowance amount first.

Technical Details

- Rebalance.sol#L101

Impact

Gas savings.

Recommendation

Use `forceApprove()` instead of `safeIncreaseAllowance()`.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/65.

4. Gas - Max checks can be omitted in deposit and redeem

Given that the YieldAggregatorVault.sol contract overrides the `deposit()` and `redeem()` functions, and there are no practical limits on deposits, the maximum deposit amount checks can be safely omitted.

Technical Details

YieldAggregatorVault.sol#L54-L60

```
54:     function deposit(uint256 _assets, address _receiver) public virtual override
nonReentrant returns (uint256) {
55:         _callHooksTarget(Constants.DEPOSIT, _msgSender());
56:
57:         uint256 maxAssets = _maxDeposit();
58:         if (_assets > maxAssets) {
59:             revert Errors.ERC4626ExceededMaxDeposit(_receiver, _assets, maxAssets);
60:         }
```

YieldAggregatorVault.sol#L72-L78

```
72:     function mint(uint256 _shares, address _receiver) public virtual override
nonReentrant returns (uint256) {
73:         _callHooksTarget(Constants.MINT, _msgSender());
74:
75:         uint256 maxShares = _maxMint();
76:         if (_shares > maxShares) {
77:             revert Errors.ERC4626ExceededMaxMint(_receiver, _shares, maxShares);
78:         }
```

Impact

Gas savings.

Recommendation

Remove the checks related to `_maxDeposit()` and `_maxMint()`.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/67.

5. Gas - Cache storage variables to prevent multiple reads from storage

Cache variables read from storage to prevent multiple SLOAD operations.

Technical Details

- `interestLeft` in `_gulp()`. Lines 84, 87, 92 and 94.
- `interestLeft` in `_updateInterestAccrued()`. Lines 104 and 110.
- `asset()` in `_withdraw()`. Lines 314 and 331.

Impact

Gas saving.

Recommendation

Cache state in local variables instead of reading again from storage.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/68.

Informational Findings

1. Informational - Missing EIP-712 initialization

The YieldAggregator.sol contract fails to initialize the base EIP-712 library, which is part of ERC20Votes.

Technical Details

The initializer in YieldAggregator.sol never calls the `__EIP712_init_unchained()` function to initialize the EIP-712 functionality included as part of the ERC20Votes library.

Impact

Informational.

Recommendation

Call `__EIP712_init_unchained()` to set the name and the version for EIP-712.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/59.

2. Informational - Missing `_harvest()` in `setPerformanceFee()`

Technical Details

The function `setPerformanceFee()` should be calling `_harvest()` before updating the `performanceFee` so the previous fee applies to any existing positive yield.

Impact

Informational.

Recommendation

Call `_harvest()` at the beginning of the function.

Developer Response

Acknowledged. The `setPerformanceFee()` function can only be called by an address that has the `YIELD_AGGREGATOR_MANAGER` role, so we will make sure to document and make it clear that if they want to harvest any previous fee before changing, they should be calling the `harvest()` before changing fee.

3. Informational - `STRATEGY_OPERATOR` is unable to update allocation points

Technical Details

The function `adjustAllocationPoints()` allows to update the allocation points for a strategy. It can be called only by the `GUARDIAN` role.

The `STRATEGY_OPERATOR` is in charge of adding and removing strategies. By not being able to call `adjustAllocationPoints(),` he won't be able to correct allocation points for a strategy that was added with the wrong amount, or that needs to be updated. He also won't be able to remove a strategy, as it can only be removed if there are 0 funds allocated to it.

This limits the `STRATEGY_OPERATOR` role and will require the `GUARDIAN` for "safe" operations, which might not be ideal.

Impact

Informational.

Recommendation

Allow the `STRATEGY_OPERATOR` to call `adjustAllocationPoints()`.

Developer Response

Acknowledged. The roles system is designed in a granular way to ensure maximum safety for the users and to decrease the chance of any rug-pull like scenario.

For that reason, strategy allocation points update is left outside of the `STRATEGY_OPERATOR` privileges.

4. Informational - Unused field in FactoryParams struct

The implementation does not use the `owner` field in the FactoryParams structure.

Technical Details

- YieldAggregatorFactory.sol#L33

Impact

Informational.

Recommendation

Remove the `owner` field from the FactoryParams structure in YieldAggregatorFactory.sol.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/66.

5. Informational - Leaked functions from base contracts

The interface of YieldAggregator.sol omits several functions inherited from base contracts.

Technical Details

All functions explicitly implemented in modules are present in the YieldAggregator.sol contract. However, the following functions, inherited from ERC20Upgradeable.sol, ERC20VotesUpgradeable.sol, and VotesUpgradeable.sol, are missing from the list:

- `asset()`
- `name()`
- `symbol()`
- `transfer()`
- `allowance()`
- `approve()`
- `transferFrom()`
- `numCheckpoints()`
- `checkpoints()`
- `clock()`
- `CLOCK_MODE()`
- `getVotes()`
- `getPastVotes()`
- `getPastTotalSupply()`
- `delegates()`
- `delegate()`
- `delegateBySig()`

Impact

Informational.

Recommendation

Consider whether these inherited functions should also be listed in YieldAggregator.sol for consistency. Note that mutable functions are not being dispatched to the corresponding YieldAggregatorVaultModule, as they are unaffected by the `use()` modifier.

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/71.

6. Informational - `toggleStrategyEmergencyStatus()` should accrue interest before deducting losses

Accrued interest pending to be flushed will be considered as balance not yet distributed when deactivating a strategy.

Technical Details

When losses are deducted through `_deductLoss()`, the implementation first reduces the undistributed quota and then decrements the share's value if needed.

```
45:         uint256 totalAssetsDepositedCache = $.totalAssetsDeposited;
46:         uint256 totalNotDistributed = _totalAssetsAllocatable() -
totalAssetsDepositedCache;
47:
48:         // set interestLeft to zero, will be updated to the right value during
_gulp()
49:         $.interestLeft = 0;
50:         if (_lossAmount > totalNotDistributed) {
51:             _lossAmount -= totalNotDistributed;
52:
53:             // socialize the loss
54:             $.totalAssetsDeposited = totalAssetsDepositedCache - _lossAmount;
55:
56:             emit Events.DeductLoss(_lossAmount);
57:         }
```

Here, `totalAssetsDeposited` is taken as the amount of assets already owned by the shareholders. However, when `_deductLoss()` is called from `toggleStrategyEmergencyStatus()`, the function doesn't update the accrued interest, which might be necessary to adjust the current value of `totalAssetsDeposited`.

The issue has no major impact since `totalAssetsDeposited` would otherwise be decremented when socializing the debt. However, there are a couple of small differences:

- If the unflushed interest covers the losses, it won't be semantically interpreted as debt socialization.
- If the losses are smaller than the unflushed interest, the difference will be re-gulped as the interest left is reset to zero.

Impact

Informational.

Recommendation

Call `_updateInterestAccrued()` when deactivating a strategy in `toggleStrategyEmergencyStatus()`.

```
    } else if (strategyCached.status == IYieldAggregator.StrategyStatus.Active) {
        $.strategies[_strategy].status = IYieldAggregator.StrategyStatus.Emergency;

+       _updateInterestAccrued();

        // we should deduct loss before decrease totalAllocated to not underflow
        _deductLoss(strategyCached.allocated);

        $.totalAllocationPoints -= strategyCached.allocationPoints;
        $.totalAllocated -= strategyCached.allocated;

        _gulp();

        emit Events.ToggleStrategyEmergencyStatus(_strategy, true);
    } else {
```

Developer Response

Fixed in https://github.com/euler-xyz/yield-aggregator/pull/69.

Final remarks

The Euler Yield Aggregator Vault is a powerful addition to the Euler Vault Kit, allowing users to benefit from multiple isolated lending markets simultaneously. The vault implements the ERC4626 standard, enhancing compatibility with the broader ecosystem. During the audit, certain aspects of the standard were found to be partially incompatible. These issues were resolved by the Euler team but increased the complexity of the implementation. The codebase features an extensive test suite, and the Euler team responded promptly to questions and addressed fixes. Euler has indicated that the vault will undergo another audit following this one, which we strongly encourage, given the added complexity of the latest fixes.