Zellic

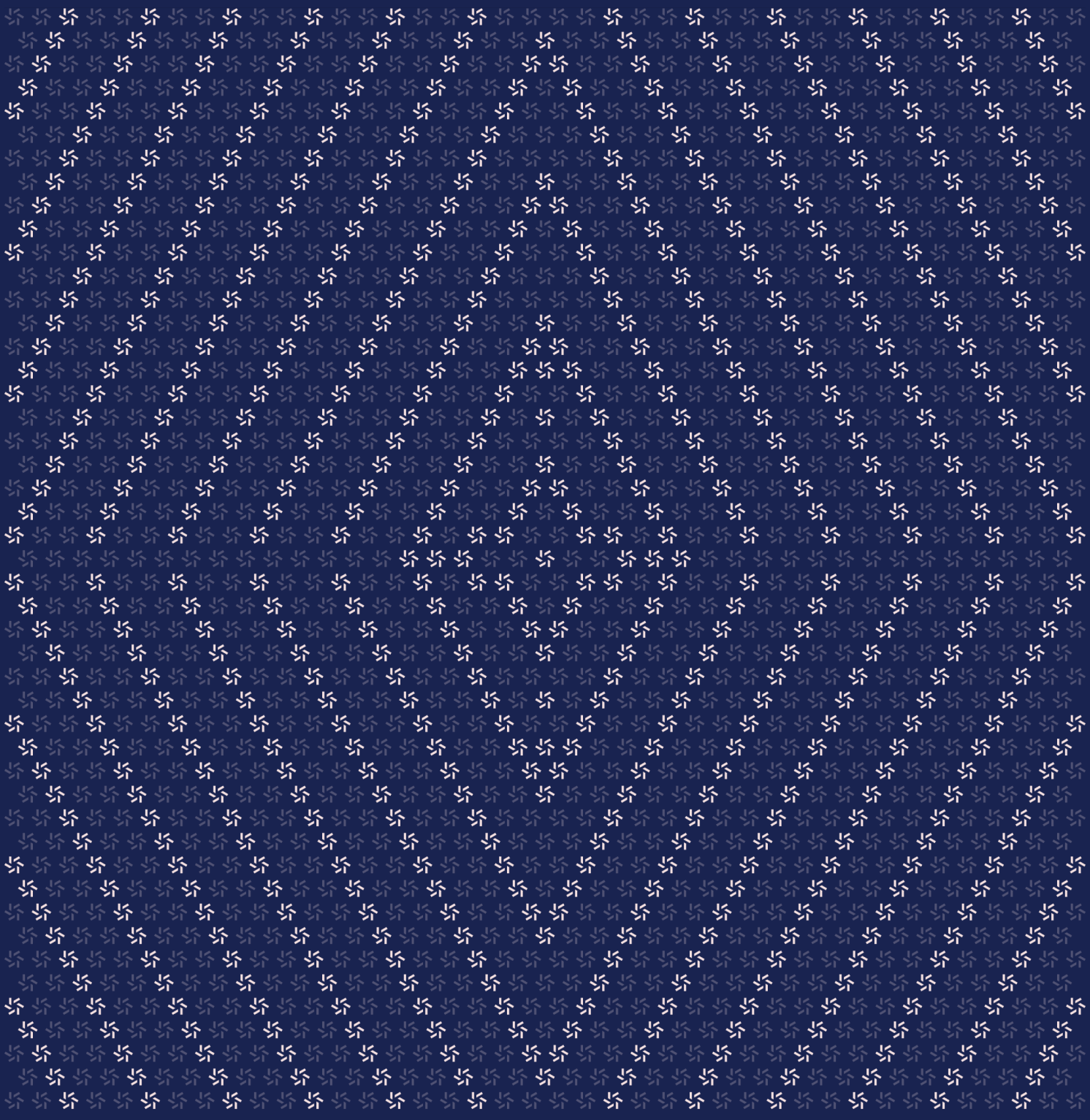
Prepared for
Sunil Srivatsa
Storm Labs

Prepared by
Ethan Lee
Vlad Toie
Zellic

December 23, 2024

Cove

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Storm Labs from November 27th to December 19th, 2024. During this engagement, Zellic reviewed Cove's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How are the baskets organized and managed?
- How is the ERC-7540 vault implemented? What are the general invariants that need to be maintained? Does the implementation satisfy the necessary invariants and EIP specifications?
- Are there any specific considerations when using CoW Swap in the context of the vault?
- How does the rebalancing mechanism work? What are the possible misconfigurations that could lead to a loss of funds?
- What possible attack vectors exist in between the rebalancing periods?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Off-chain components that deal with handling CoW Swap operations
- Off-chain components that should use the correct parameters when calling any of the on-chain components

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Cove contracts, we discovered nine findings. Two critical issues were found. Three were of high impact, two were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Storm Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	3
Medium	2
Low	1
Informational	1

2. Introduction

2.1. About Cove

Storm Labs contributed the following description of Cove:

> Cove is an asset management protocol designed to maximize returns through intelligent automation. Cove simplifies complex strategies into accessible, yield-bearing products.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Cove Contracts

Type	Solidity
Platform	EVM-compatible
Target	Cove Finance Core Contracts
Repository	https://github.com/Storm-Labs-Inc/cove-contracts-core ↗
Version	`fad59b375e8176a6753d33668af2216aee9a45ad`

Programs	`src/AnchoredOracle.sol`
	`src/AssetRegistry.sol`
	`src/BasketManager.sol`
	`src/BasketToken.sol`
	`src/FeeCollector.sol`
	`src/libraries/BasketManagerUtils.sol`
	`src/libraries/BitFlag.sol`
	`src/strategies/AutomaticWeightStrategy.sol`
	`src/strategies/ManagedWeightStrategy.sol`
	`src/strategies/StrategyRegistry.sol`
	`src/strategies/WeightStrategy.sol`
	`src/swap_adapters/CoWSwapAdapter.sol`
	`src/swap_adapters/CoWSwapClone.sol`
	`src/swap_adapters/TokenSwapAdapter.sol`

Target	Token Plugins Update, diff between ‘fbeedec4‘ and ‘439806ce‘
Repository	https://github.com/Storm-Labs-Inc/token-plugins-upgradeable ↗
Version	`fbeedec4e8f52182d4cf769e4a57755cb6d16074`
Programs	`contracts/ERC20PluginsUpgradeable.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.7 person-weeks. The assessment was conducted by two consultants over the course of two and a half calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Ethan Lee
Engineer
ethl@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 27, 2024	Kick-off call
November 27, 2024	Start of primary review period
December 19, 2024	End of primary review period
December 23, 2024	Closing call

3. Detailed Findings

3.1. Lack of access control in `requestDeposit`

Target	BasketToken		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The BasketToken contract manages users' deposits and redemptions. It follows the ERC-7540 standard, which is an ERC-4626 vault with asynchronous operations. The `requestDeposit` function facilitates deposits of the base assets into the vault. The function stores the deposit request in the `_depositRequests` mapping and transfers the assets to the contract.

```
function requestDeposit(uint256 assets, address controller, address owner)
    public returns (uint256 requestId) {

    // Checks
    if (assets == 0) {
        revert Errors.ZeroAmount();
    }
    requestId = nextDepositRequestId;
    uint256 userLastDepositRequestId = lastDepositRequestId[controller];
    // If the user has a pending deposit request in the past, they must wait
    for it to be fulfilled before making a
    // new one
    if (userLastDepositRequestId != requestId) {
        if (pendingDepositRequest(userLastDepositRequestId, controller) > 0) {
            revert MustClaimOutstandingDeposit();
        }
    }
    // If the user has a claimable deposit request, they must claim it before
    making a new one
    if (claimableDepositRequest(userLastDepositRequestId, controller) > 0) {
        revert MustClaimOutstandingDeposit();
    }
    if (AssetRegistry(assetRegistry).hasPausedAssets(bitFlag)) {
        revert AssetPaused();
    }
    // Effects
    DepositRequestStruct storage depositRequest = _depositRequests[requestId];
    // update controllers balance of assets pending deposit
    depositRequest.depositAssets[controller] += assets;
```

```
        // update total pending deposits for the current requestId
        depositRequest.totalDepositAssets += assets;
        // update controllers latest deposit request id
        lastDepositRequestId[controller] = requestId;
        emit DepositRequest(controller, owner, requestId, msg.sender, assets);

        // Interactions
        // Assets are immediately transferrred to here to await the basketManager
        to pull them
        // slither-disable-next-line arbitrary-send-erc20
        IERC20(asset()).safeTransferFrom(owner, address(this), assets);
    }
```

We note that the function does not implement any sort of access control with regards to the `owner` address. This means that any address can deposit assets on behalf of any other address, as long as the `owner` has previously approved the contract to spend their assets. This is generally the case in such contracts, as the approve function is called with `type(uint256).max` to allow the contract to spend any amount of the asset, improving the UX.

Impact

This would lead to total loss of base-asset funds for the users that have approved the contract to spend their assets. An attacker can deposit the assets of any user into the contract and specify themselves as the `controller`, effectively stealing the assets of the users.

Recommendations

We recommend ensuring that adequate approval checks are in place to prevent unauthorized deposits on behalf of users. Alternatively, only allow the `owner` to deposit assets on their own behalf.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `f2d21812` ↗.

3.2. Underflow when calculating basket balances

Target	BasketManagerUtils		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	Critical

Description

The `_processInternalTrades` function handles the internal trades between baskets. It calculates the initial buy amount and the net buy amount for each trade. The function then checks if the net buy amount is within the min and max amount limits. It then checks whether the calculated net buy and trade sell amounts are within the current basket balances' limits. It then updates the balances for each basket that requires a trade.

```
function _processInternalTrades(
    BasketManagerStorage storage self,
    InternalTrade[] calldata internalTrades,
    address[] calldata baskets,
    uint256[][] memory basketBalances
)
    private
{
    uint256 swapFee = self.swapFee; // Fetch swapFee once for gas optimization
    uint256 internalTradesLength = internalTrades.length;
    for (uint256 i = 0; i < internalTradesLength;) {
        InternalTrade memory trade = internalTrades[i];

        // ...

        uint256 initialBuyAmount = self.eulerRouter.getQuote(
            self.eulerRouter.getQuote(info.netSellAmount, trade.sellToken,
    _USD_ISO_4217_CODE),
            _USD_ISO_4217_CODE,
            trade.buyToken
        );

        if (swapFee > 0) {
            info.feeOnBuy = FixedPointMathLib.fullMulDiv(initialBuyAmount,
    swapFee, 20_000);
            self.collectedSwapFees[trade.buyToken] += info.feeOnBuy;
            emit SwapFeeCharged(trade.buyToken, info.feeOnBuy);
        }
```

```
            info.netBuyAmount = initialBuyAmount - info.feeOnBuy;

            if (info.netBuyAmount < trade.minAmount || trade.maxAmount <
        info.netBuyAmount) {
                revert InternalTradeMinMaxAmountNotReached();
            }

            if (trade.sellAmount >
        basketBalances[info.fromBasketIndex][info.sellTokenAssetIndex]) {
                revert IncorrectTradeTokenAmount();
            }

            if (info.netBuyAmount >
        basketBalances[info.toBasketIndex][info.toBasketBuyTokenIndex]) {
                revert IncorrectTradeTokenAmount();
            }

            unchecked {
                self.basketBalanceOf[trade.fromBasket][trade.sellToken] =
                    basketBalances[info.fromBasketIndex][info.sellTokenAssetIndex]
        -= trade.sellAmount; // nosemgrep

                self.basketBalanceOf[trade.fromBasket][trade.buyToken] =
                    basketBalances[info.fromBasketIndex][info.buyTokenAssetIndex]
        += info.netBuyAmount; // nosemgrep

                self.basketBalanceOf[trade.toBasket][trade.buyToken] =
                    basketBalances[info.toBasketIndex][info.toBasketBuyTokenIndex]
        -= initialBuyAmount; // nosemgrep

                self.basketBalanceOf[trade.toBasket][trade.sellToken] =

        basketBalances[info.toBasketIndex][info.toBasketSellTokenIndex]
        += info.netSellAmount; // nosemgrep
                ++i;
            }
        }
    }
```

The issue is that, even though the if case for the buy amount checks whether `info.netBuyAmount` is greater than the `basketBalances[info.toBasketIndex][info.toBasketBuyTokenIndex]`, the subtraction operation is performed with `initialBuyAmount`, which will always be greater than `info.netBuyAmount`. This can lead to an underflow when calculating the new basket balance. Normally, in Solidity versions 0.8.0 and above, the compiler will throw an error when an underflow is detected. However, in this case, the underflow is not detected by the compiler because the subtraction operation is performed within an `unchecked` block.

Impact

The underflow would lead to the basket balances being set to a very large number, potentially blocking the usability of the particular basket. This in turn could lead to loss of funds for the users.

Recommendations

We recommend ensuring that the if case checks the correct variable to prevent underflows. Additionally, we recommend removing the `unchecked` block to allow the compiler to detect underflows, as the gas optimization gained from using `unchecked` is minimal compared to the potential risks.

```
function _processInternalTrades(
    BasketManagerStorage storage self,
    InternalTrade[] calldata internalTrades,
    address[] calldata baskets,
    uint256[][] memory basketBalances
)
    private
{
    uint256 swapFee = self.swapFee; // Fetch swapFee once for gas optimization
    uint256 internalTradesLength = internalTrades.length;
    for (uint256 i = 0; i < internalTradesLength;) {
        // ...

        if (info.netBuyAmount > basketBalances[info.toBasketIndex][info.
            toBasketBuyTokenIndex]) {
        if (initialBuyAmount > basketBalances[info.toBasketIndex][info.
            toBasketBuyTokenIndex]) {

            revert IncorrectTradeTokenAmount();
        }

        unchecked {
          self.basketBalanceOf[trade.fromBasket][trade.sellToken] =
                  basketBalances[info.fromBasketIndex][info.sellTokenAssetIndex]
    -= trade.sellAmount;

          self.basketBalanceOf[trade.fromBasket][trade.buyToken] =
                  basketBalances[info.fromBasketIndex][info.buyTokenAssetIndex]
    += info.netBuyAmount;

          self.basketBalanceOf[trade.toBasket][trade.buyToken] =
                  basketBalances[info.toBasketIndex][info.toBasketBuyTokenIndex]
    -= initialBuyAmount;
```

```
        self.basketBalanceOf[trade.toBasket][trade.sellToken] =

    basketBalances[info.toBasketIndex][info.toBasketSellTokenIndex]
    += info.netSellAmount;
        ++i;
      }
    }
}
```

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit 8ae19aea ↗.

3.3. Missing rebalance-status check in `updateBitFlag()` leads to incorrect rebalancing

Target	BasketToken		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `updateBitFlag()` function lacks a rebalance-status check.

```
function updateBitFlag(address basket, uint256 bitFlag)
    external onlyRole(_TIMELOCK_ROLE) {
    // Checks
    // Check if basket exists
    uint256 indexPlusOne = _bmStorage.basketTokenToIndexPlusOne[basket];
    if (indexPlusOne == 0) {
        revert BasketTokenNotFound();
    }
    uint256 currentBitFlag = BasketToken(basket).bitFlag();
    if (currentBitFlag == bitFlag) {
        revert BitFlagMustBeDifferent();
    }
    // Check if the new bitFlag is inclusive of the current bitFlag
    if ((currentBitFlag & bitFlag) != currentBitFlag) {
        revert BitFlagMustIncludeCurrent();
    }
    address strategy = BasketToken(basket).strategy();
    if (!WeightStrategy(strategy).supportsBitFlag(bitFlag)) {
        revert BitFlagUnsupportedByStrategy();
    }
    bytes32 newId = keccak256(abi.encodePacked(bitFlag, strategy));
    if (_bmStorage.basketIdToAddress[newId] != address(0)) {
        revert BasketIdAlreadyExists();
    }
    // Remove the old bitFlag mapping and add the new bitFlag mapping
    bytes32 oldId = keccak256(abi.encodePacked(currentBitFlag, strategy));
    _bmStorage.basketIdToAddress[oldId] = address(0);
    _bmStorage.basketIdToAddress[newId] = basket;
    _bmStorage.basketAssets[basket]
    = AssetRegistry(_bmStorage.assetRegistry).getAssets(bitFlag);
```

```
    emit BasketBitFlagUpdated(basket, currentBitFlag, bitFlag, oldId, newId);
    // Update the bitFlag in the BasketToken contract
    BasketToken(basket).setBitFlag(bitFlag);
}
```

The `updateBitFlag()` function modifies the `basketAssets` array using the following line:

```
_bmStorage.basketAssets[basket]
    = AssetRegistry(_bmStorage.assetRegistry).getAssets(bitFlag);
```

This behavior can lead to conflicts during rebalancing if an admin changes assets using `updateBitFlag()`.

```
address[] memory assets = self.basketAssets[basket];
// nosemgrep:
    solidity.performance.array-length-outside-loop.array-length-outside-loop
uint256 assetsLength = assets.length;
basketBalances[i] = new uint256[](assetsLength);
for (uint256 j = 0; j < assetsLength;) {
address asset = assets[j];
// nosemgrep: solidity.performance.state-variable-read-in-a-loop.state-
    variable-read-in-a-loop
uint256 currentAssetAmount = self.basketBalanceOf[basket][asset];
basketBalances[i][j] = currentAssetAmount;
// nosemgrep: solidity.performance.state-variable-read-in-a-loop.state-
    variable-read-in-a-loop
totalValue_[i] += self.eulerRouter.getQuote(currentAssetAmount, asset,
    _USD_ISO_4217_CODE);
unchecked {
    // Overflow not possible: j is less than assetsLength
    ++j;
}
}
```

The rebalancing logic, which depends on the `basketAssets` array, may fetch incorrect or outdated assets. This leads to errors in balance calculations or rebalancing outcomes. Here is an error scenario.

1. The `updateBitFlag()` function is called during an active rebalance.
2. The `basketAssets` array is updated with a new set of assets via the `getAssets(bitFlag)` call.
3. The rebalancing logic uses the modified `basketAssets` array, potentially fetching incorrect assets:

4. This results in incorrect balances, total value calculations, or improper asset rebalancing.

Recommendations

Add a rebalance-status check to `updateBitFlag()` to prevent the function from being called while a rebalance is in progress.

Impact

If the `updateBitFlag()` function is called during a rebalance, it may modify the `basketAssets` array, leading to incorrect asset selection or balance calculation, potentially causing improper rebalancing and errors.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `425e4f9f` ↗.

3.4. Incorrect swap-fee calculation on `feeOnBuy`

Target	BasketManagerUtils		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The current implementation calculates the swap fee for the `buyAmount` after the fee deduction. According to the Cove documentation ↗, `buyAmount` is defined as the amount of the buy token being received before fee deduction. However, in the implementation, the fee is deducted from the `sellAmount` before determining the `buyAmount`.

```
info.netSellAmount = trade.sellAmount - info.feeOnSell;

// Calculate initial buyAmount based on netSellAmount
uint256 initialBuyAmount = self.eulerRouter.getQuote(
    self.eulerRouter.getQuote(info.netSellAmount, trade.sellToken,
    _USD_ISO_4217_CODE),
    _USD_ISO_4217_CODE,
    trade.buyToken
);

// Calculate fee on buyAmount
if (swapFee > 0) {
    info.feeOnBuy = FixedPointMathLib.fullMulDiv(initialBuyAmount, swapFee,
    20_000);
    self.collectedSwapFees[trade.buyToken] += info.feeOnBuy;
    emit SwapFeeCharged(trade.buyToken, info.feeOnBuy);
}
info.netBuyAmount = initialBuyAmount - info.feeOnBuy;
```

Impact

The current fee-calculation method leads to undercollected fees, causing financial loss to the protocol and inconsistencies with the documented behavior.

Recommendations

Use the `buyAmount` before any fee deductions instead of `initialBuyAmount` when calculating `feeOnBuy`.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `803fc0d1` ↗.

3.5. Management-fee calculation results in lower effective rate

Target	BasketToken		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The `BasketToken._harvestManagementFee()` mints new shares as a fee instead of transferring an existing token amount. This approach causes the effective fee rate to be lower than the intended rate due to dilution effects.

```
function _harvestManagementFee(uint16 feeBps, address feeCollector) internal {
    // Checks
    if (feeBps > _MAX_MANAGEMENT_FEE) {
        revert InvalidManagementFee();
    }
    uint256 timeSinceLastHarvest = block.timestamp
    - lastManagementFeeHarvestTimestamp;

    // Effects
    lastManagementFeeHarvestTimestamp = uint40(block.timestamp);
    if (feeBps != 0) {
        if (timeSinceLastHarvest != 0) {
            if (timeSinceLastHarvest != block.timestamp) {
                // remove shares held by the treasury or currently pending
    redemption from calculation
                uint256 currentTotalSupply = totalSupply()
    - balanceOf(feeCollector)
                    - pendingRedeemRequest(lastRedeemRequestId[feeCollector],
    feeCollector);
                uint256 fee = FixedPointMathLib.fullMulDiv(
                    currentTotalSupply, feeBps * timeSinceLastHarvest,
    _MANAGEMENT_FEE_DECIMALS * uint256(365 days)
                );
                if (fee != 0) {
                    emit ManagementFeeHarvested(fee);
                    _mint(feeCollector, fee);
                    // Interactions
                    FeeCollector(feeCollector).notifyHarvestFee(fee);
                }
            }
```

```
        }
    }
}
```

Take the following example.

If the total supply is 100 and the fee rate is 5%, the protocol mints five shares as the fee. However, after minting, the total supply becomes 105, making the effective fee rate 5/105 = 4.7%, which is less than the intended 5%. This discrepancy can result in the protocol consistently collecting a lower fee amount than expected.

Impact

The current fee-calculation method leads to undercollected fees, causing financial loss to the protocol and underperformance of fee collection for the protocol.

Recommendations

Adjust the calculation logic in `_harvestManagementFee()` to mint a higher amount of shares to match the intended fee rate after accounting for the dilution effect. It can use the formula `(total_supply * fee_rate) / (fee_precision - fee_rate)`.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `1a539a7f`↗.

3.6. Missing mapping update in `BasketToken.updateBitFlag()` causes rebalancing failure

Target	BasketToken		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `updateBitFlag()` function does not update the `basketAssetToIndexPlusOne` mapping after adding a new asset to the registry. This leads to errors during rebalancing when the rebalance cannot locate the new asset.

```
function updateBitFlag(address basket, uint256 bitFlag)
    external onlyRole(_TIMELOCK_ROLE) {
    // Checks
    // Check if basket exists
    uint256 indexPlusOne = _bmStorage.basketTokenToIndexPlusOne[basket];
    if (indexPlusOne == 0) {
        revert BasketTokenNotFound();
    }
    uint256 currentBitFlag = BasketToken(basket).bitFlag();
    if (currentBitFlag == bitFlag) {
        revert BitFlagMustBeDifferent();
    }
    // Check if the new bitFlag is inclusive of the current bitFlag
    if ((currentBitFlag & bitFlag) != currentBitFlag) {
        revert BitFlagMustIncludeCurrent();
    }
    address strategy = BasketToken(basket).strategy();
    if (!WeightStrategy(strategy).supportsBitFlag(bitFlag)) {
        revert BitFlagUnsupportedByStrategy();
    }
    bytes32 newId = keccak256(abi.encodePacked(bitFlag, strategy));
    if (_bmStorage.basketIdToAddress[newId] != address(0)) {
        revert BasketIdAlreadyExists();
    }
    // Remove the old bitFlag mapping and add the new bitFlag mapping
    bytes32 oldId = keccak256(abi.encodePacked(currentBitFlag, strategy));
    _bmStorage.basketIdToAddress[oldId] = address(0);
    _bmStorage.basketIdToAddress[newId] = basket;
```

```
    _bmStorage.basketAssets[basket]
    = AssetRegistry(_bmStorage.assetRegistry).getAssets(bitFlag);
    emit BasketBitFlagUpdated(basket, currentBitFlag, bitFlag, oldId, newId);
    // Update the bitFlag in the BasketToken contract
    BasketToken(basket).setBitFlag(bitFlag);
}
```

```
function basketTokenToRebalanceAssetToIndex(
    BasketManagerStorage storage self,
    address basketToken,
    address asset
)
    public
    view
    returns (uint256 index)
{
    index = self.basketAssetToIndexPlusOne[basketToken][asset];
    if (index == 0) {
        revert AssetNotFoundInBasket();
    }
    unchecked {
        // Overflow not possible: index is not 0
        return index - 1;
    }
}
```

Here is an error scenario.

1. A new asset is added to the `AssetRegistry`.
2. Then, `BasketManager.updateBitFlag()` is called to enable the new asset. The `basketAssetToIndexPlusOne` mapping does not include the index of the new asset because it is only initialized in `createNewBasket()`.
3. During rebalancing, it calls `basketTokenToRebalanceAssetToIndex()` via `_processInternalTrades()` in `proposeTokenSwap()`. This causes an `AssetNotFound` error as the new asset's index is missing from `basketAssetToIndexPlusOne`.
4. The rebalancing process fails, and the new asset cannot be included in the rebalance.

Impact

The current fee-calculation method leads to undercollected fees, causing financial loss to the protocol and underperformance of fee collection for the protocol.

Recommendations

Modify the `updateBitFlag()` function to ensure the `basketAssetToIndexPlusOne` mapping is updated whenever a new asset is added.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `9e67b7b8` ↗.

3.7. Potential price manipulation via read-only reentrancy in `BasketToken.proRataRedeem()`

Target	BasketToken		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

BasketToken inherits from ERC20PluginsUpgradeable, which is a modified version of 1inch's ERC20Plugins as an upgradable contract. However, ERC20Plugins has a potential reentrancy vulnerability when the from or to address is an arbitrary address. An attacker could add a plug-in to BasketToken to trigger a reentrancy attack.

```
function proRataRedeem(uint256 shares, address to, address from) public {
    // Effects
    uint16 feeBps = BasketManager(basketManager).managementFee(address(this));
    address feeCollector = BasketManager(basketManager).feeCollector();
    _harvestManagementFee(feeBps, feeCollector);
    if (msg.sender != from) {
        _spendAllowance(from, msg.sender, shares);
    }
    uint256 totalSupplyBefore = totalSupply();
    _burn(from, shares); // reentrancy point with `from`s plugin.
    // Interactions
    BasketManager(basketManager).proRataRedeem(totalSupplyBefore, shares, to);
}
```

When calling `proRataRedeem()`, `BasketToken.totalAssets()` could be affected by read-only reentrancy. If the basket supports tokens as assets that allow callbacks, such as ERC-777 tokens, an attacker can hijack the control flow using the receiver hook during the asset transfer in `proRataRedeem()`. Although BasketManager has a reentrancy guard in place, `BasketToken.totalAssets()` can still be called. If a third party uses the `totalAssets()` function for value or price calculations, they may end up calculating an imbalanced asset value by calling `bm.basketBalanceOf()` while the assets have not yet been fully redeemed in the loop.

Here is an attack scenario. Suppose a third party, such as a lending protocol, uses the Basket LP share tokens as collateral.

1. An attacker deposits a significant amount of assets into the basket.
2. The rebalance ends, and the attacker claims the basket tokens.

3. The attacker deposits the basket tokens into the lending protocol.
4. The attacker adds a malicious plug-in to trigger reentrancy.
5. The attacker calls `proRataRedeem()`.
6. It burns the basket tokens and calls the malicious plug-in contract in `ERC20PluginUpgradeable._updateBalances()`.
7. The total supply of the basket token has already decreased since the tokens were burned by calling `super._update()` in `ERC20PluginsUpgradeable._update()`.
8. The asset balances have not yet decreased before calling `BasketManager.proRataRedeem()`.
9. The LP-token price increases because the LP-token pricing calculation is based on `asset_values / total_supply_of_basket_i` (as described in the Cove Finance RFC documentation ↗).
10. Finally, the attacker borrows other tokens (excluding the basket token, which has its own reentrancy guard) by exploiting the manipulated basket-token price as collateral.

Impact

An attacker can manipulate the price of Basket LP tokens through read-only reentrancy during `proRataRedeem()`, potentially exploiting third-party protocols that rely on LP-token pricing for collateral, leading to the drainage of funds of third-party protocols.

Recommendations

To adhere to the check-effects-interactions pattern, call `_burn()` after calling `BasketManager.proRataRedeem()` and do not support any tokens that can cause reentrancy (resolved in another reentrancy issue; see Finding 3.8. ↗). Additionally, the documentation should warn third parties who use the price of BasketToken about the risk of read-only reentrancy to ensure third parties check for reentrancy.

Remediation

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit `39e19da0` ↗.

3.8. Potential asset manipulation via read-only reentrancy in `BasketManagerUtils.proRataRedeem()`

Target	BasketManagerUtils		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

When calling `proRataRedeem()`, `BasketToken.totalAssets()` could be affected by read-only reentrancy.

```
// Interactions
for (uint256 i = 0; i < assetsLength;) {
    address asset = assets[i];
    // nosemgrep: solidity.performance.state-variable-read-in-a-loop.state-variable-read-in-a-loop
    uint256 balance = self.basketBalanceOf[basket][asset];
    // Rounding direction: down
    // Division-by-zero is not possible: totalSupplyBefore is greater than 0
    uint256 amountToWithdraw = FixedPointMathLib.fullMulDiv(burnedShares, balance, totalSupplyBefore);
    if (amountToWithdraw > 0) {
        // nosemgrep: solidity.performance.state-variable-read-in-a-loop.state-variable-read-in-a-loop
        self.basketBalanceOf[basket][asset] = balance - amountToWithdraw;
        // Asset is an allowlisted ERC20 with no reentrancy problem in transfer
        // slither-disable-next-line reentrancy-no-eth
        IERC20(asset).safeTransfer(to, amountToWithdraw);
    }
    unchecked {
        // Overflow not possible: i is less than assetsLength
        ++i;
    }
}
```

If the basket supports tokens as assets that allow callbacks, such as ERC-777 tokens, an attacker can hijack the control flow using the receiver hook during the asset transfer in `proRataRedeem()`. Although BasketManager has a reentrancy guard in place, `BasketToken.totalAssets()` can still be called.

If a third party uses the `totalAssets()` function for value or price calculations, they may end up calculating an imbalanced asset value by calling `bm.basketBalanceOf()` while the assets have not yet been fully redeemed in the loop.

Impact

Third parties that utilize `totalAssets()` could receive incorrect values. If these values are used for price calculations, it may lead to asset losses for the third parties.

Recommendations

To adhere to the checks-effects-interactions pattern, separate the balance effects and asset transfer into two separate loops. Ensure that tokens that can cause reentrancy, such as ERC-777 tokens, are not supported as assets.

Remediation

Storm Labs will not support any tokens that can cause reentrancy.

This issue has been acknowledged by Storm Labs, and a fix was implemented in commit ae3f3050 ↗.

3.9. Mismatch between CoW Swap adapter sequence diagram and implementation

Target	Documentation		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The sequence diagram for the CoW Swap adapter pictured here ↗ differs from the implementation. Quotes for token swaps should be retrieved before calling `proposeTokenSwap()`, and the loop should begin at `proposeTokenSwap()`, as `completeRebalance` resets the status to `REBALANCE_PROPOSED`.

Recommendations

Fix the sequence diagram to reflect the actual implementation flow.

Impact

The mismatch between the sequence diagram and the implementation may cause confusion for users.

Remediation

The documentation has been updated by Storm Labs to reflect the actual implementation flow.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Transparent intent of redeemal might lead to unintended consequences

We have analyzed the design concerning the transparent intent of the "redemption" of shares during the rebalancing procedure, which implicitly requires swapping the basket tokens into base tokens.

It is evident that anyone can assume the Manager will eventually need to rebalance some of the baskets. When low-liquidity tokens are involved, the likelihood of potential issues increases. In such scenarios, for the rebalancing to occur, the Manager is either

- forced to accept a lower price than desirable for the base asset, or
- forced to wait for a satisfactory price.

In either case, it makes little sense for a malicious party to propose selling at an acceptable price for the baskets. Because baskets are designed to ensure that rebalancing occurs, a malicious party would have incentives to force a peer-to-peer exchange through CoW Swap that is acceptable enough for the baskets but leaves room for profit (assuming they can outbid private market makers and other DEXes that CoW Swap deals with).

Additionally, we have identified the following potential attack vector:

1. A malicious user monitors the `proposeRebalance` call.
2. The malicious user creates a honeypot peer-to-peer order at an attractive price so that the CoW Swap quote returns a higher `minAmount`, which the basket manager uses as a reference.
3. The basket manager calls `executeTokenSwap` and submits the order to the CoW Swap API off chain.
4. The malicious user times their action to cancel the order just in time so that it does not go through, forcing CoW Swap to attempt to fill the order with an out-of-bounds `minAmount` (i.e., outside what the honeypot proposed).
5. The swap is delayed temporarily, and the funds in the clone cannot be claimed back, thereby causing a temporary denial of service of the rebalancing.

Although this scenario is highly unlikely and involves some degree of risk for the malicious user, it is important to consider its implications.

> We note that the Cove Finance team has stated that the attack vectors are feasible; however, they do not consider using low-liquidity tokens in the baskets. We leave this section for potential future consideration.

4.2. Shares can round down to zero

The `proRataRedeem` function allows users to redeem their shares in the basket when the basket is not in the rebalancing state. The function calculates the amount of each asset to be redeemed based on the number of shares to be redeemed and the total supply of the basket. The function then transfers the calculated amount of each asset to the user. However, the function does not revert if the calculated amount of an asset is zero.

```
function proRataRedeem(
    BasketManagerStorage storage self,
    uint256 totalSupplyBefore,
    uint256 burnedShares,
    address to
)
    external
{
    // Checks
    if (totalSupplyBefore == 0) {
        revert ZeroTotalSupply();
    }
    if (burnedShares == 0) {
        revert ZeroBurnedShares();
    }
    if (burnedShares > totalSupplyBefore) {
        revert CannotBurnMoreSharesThanTotalSupply();
    }
    if (to == address(0)) {
        revert Errors.ZeroAddress();
    }
    // Revert if the basket is currently rebalancing
    if ((self.rebalanceStatus.basketMask & (1 <<
    self.basketTokenToIndexPlusOne[msg.sender] - 1)) != 0) {
        revert MustWaitForRebalanceToComplete();
    }

    address basket = msg.sender;
    address[] storage assets = self.basketAssets[basket];
    uint256 assetsLength = assets.length;

    // Interactions
    for (uint256 i = 0; i < assetsLength;) {
        address asset = assets[i];
        // nosemgrep: solidity.performance.state-variable-read-in-a-
    loop.state-variable-read-in-a-loop
```

```
        uint256 balance = self.basketBalanceOf[basket][asset];
        // Rounding direction: down
        // Division-by-zero is not possible: totalSupplyBefore is greater than
    0
        uint256 amountToWithdraw = FixedPointMathLib.fullMulDiv(burnedShares,
    balance, totalSupplyBefore);
        if (amountToWithdraw > 0) {
            // nosemgrep: solidity.performance.state-variable-read-in-a-
    loop.state-variable-read-in-a-loop
            self.basketBalanceOf[basket][asset] = balance - amountToWithdraw;
            // Asset is an allowlisted ERC20 with no reentrancy problem in
    transfer
            // slither-disable-next-line reentrancy-no-eth
            IERC20(asset).safeTransfer(to, amountToWithdraw);
        }
        unchecked {
            // Overflow not possible: i is less than assetsLength
            ++i;
        }
    }
}
```

This can lead to a situation where the user redeems shares but receives no assets in return. This is because the `burnedShares * balance / totalSupplyBefore` calculation can round down to zero when `burnedShares` is small.

> Due to lack of an adequate fix for the current code architecture, we leave this section for potential future consideration, should the code allow for a definite fix. As of the time of writing, a revert in the case of `amountToWithdraw` could be problematic for users. It may be that not all of the assets are redeemable at a rate greater than zero, but the user should still be able to redeem the ones that are. To prevent the price of the share token from inflating, the initial asset must be deposited after creating the basket token. Storm Labs can refer to this GitHub issue ↗, "Implement or recommend mitigations for ERC4626 inflation attacks", for typical mitigations for ERC-4626 vault inflation.

4.3. Weights can pass unchecked

The rebalancing mechanism allows the conversion and recalculation of the basket's multiple assets. When it happens, the rebalance passes through multiple stages of the contract. The `completeRebalance` finalizes the rebalance procedures if all the conditions are met.

```
function completeRebalance(
    BasketManagerStorage storage self,
    ExternalTrade[] calldata externalTrades,
    address[] calldata baskets,
    uint64[][] calldata basketTargetWeights
)
    external
{
    if (self.rebalanceStatus.status == Status.NOT_STARTED) {
        revert NoRebalanceInProgress();
    }
    _validateBasketHash(self, baskets, basketTargetWeights);
    // Check if the rebalance was proposed more than 15 minutes ago
    // slither-disable-next-line timestamp
    if (block.timestamp - self.rebalanceStatus.timestamp < 15 minutes) {
        revert TooEarlyToCompleteRebalance();
    }
    // if external trades are proposed and executed, finalize them and claim
    results from the trades
    if (self.rebalanceStatus.status == Status.TOKEN_SWAP_EXECUTED) {
        if (keccak256(abi.encode(externalTrades)) != self.externalTradesHash)
    {
            revert ExternalTradeMismatch();
        }
        _processExternalTrades(self, externalTrades);

    }

    uint256 len = baskets.length;
    uint256[] memory totalValue_ = new uint256[](len);
    // 2d array of asset amounts for each basket after all trades are settled
    uint256[][] memory afterTradeAmounts_ = new uint256[][](len);
    _initializeBasketData(self, baskets, afterTradeAmounts_, totalValue_);
    // Confirm that target weights have been met, if max retries is reached
    continue regardless
    if (self.retryCount < _MAX_RETRIES) {
        if (!_isTargetWeightMet(self, baskets, afterTradeAmounts_,
    totalValue_, basketTargetWeights)) {
            self.retryCount += 1;
            self.rebalanceStatus.timestamp = uint40(block.timestamp);
            self.externalTradesHash = bytes32(0);
            self.rebalanceStatus.status = Status.REBALANCE_PROPOSED;
            return;
        }
    }
    _finalizeRebalance(self, baskets);
```

```
}
```

Notice that if the target weights are not met, the function will retry the rebalance process up to `_MAX_RETRIES` times. However, the function does not revert if the target weights are not met after the maximum number of retries, and it simply finalizes the rebalance process with whatever weights were used. This could lead to a scenario where unfavorable weights are used to finalize the rebalance process, which could lead to a basket that is not properly balanced.

> Due to the current mechanism (as of the time of writing) of continuing the process of external trades even if the target weights are not met, the rebalance process cannot be reverted altogether. We thus leave this section for potential future consideration, as the current mechanism is intended. Should future design changes allow for a cancelation of the rebalance process, it would be important to revert in the case of unfavorable weights.

4.4. Denial-of-service risk where attackers can disrupt rebalance process with `BasketManagerUtils.completeRebalance()` functionality

The rebalance status progresses through the following states: `NOT_STARTED`, `REBALANCE_PROPOSED`, `TOKEN_SWAP_PROPOSED`, and then `TOKEN_SWAP_EXECUTED`. To advance the status, the functions `proposeRebalance()`, `proposeTokenSwap()`, `executeTokenSwap()`, and `completeRebalance()` must be called in that order.

```
if (self.rebalanceStatus.status == Status.NOT_STARTED) {
    revert NoRebalanceInProgress();
}
```

When `RebalanceStatus` is either `REBALANCE_PROPOSED` or `TOKEN_SWAP_PROPOSED`, `completeRebalance()` can be called.

```
if (!_isTargetWeightMet(self, baskets, afterTradeAmounts_, totalValue_,
    basketTargetWeights)) {
    // If target weights are not met and we have not reached max retries,
    revert to beginning of rebalance
    // to allow for additional token swaps to be proposed and increment
    retryCount.
    self.retryCount += 1;
    self.rebalanceStatus.timestamp = uint40(block.timestamp);
    self.externalTradesHash = bytes32(0);
    self.rebalanceStatus.status = Status.REBALANCE_PROPOSED;
    return;
```

```
}
```

The function only checks that the status is not in the initial state (`NOT_STARTED`) and silently sets an error instead of reverting. As a result, anyone can call this function as long as the status is not `NOT_STARTED` and 15 minutes has passed since the rebalance was proposed. When the function is called, the status reverts back to `REBALANCE_PROPOSED`. Imagine the following attack scenario:

1. The admin proposes a rebalance, changing the status to `REBALANCE_PROPOSED`.
2. The admin proposes a token swap, which changes the status to `TOKEN_SWAP_PROPOSED`.
3. Fifteen minutes after the rebalance has been proposed, the malicious user calls `completeRebalance()`, reverting the status back to `REBALANCE_PROPOSED`.
4. The admin must propose the token swap again.
5. Steps 2–3 are repeated to interfere with the rebalance.

As a result, the admin is unable to complete the rebalance process, forcing retries without success.

The team has stated that the attack vectors are feasible; however, they will complete rebalance within 15 minutes. We leave this section for potential future consideration, should the team decide to extend the time frame for rebalance completion, or significantly change the rebalance process.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: AnchoredOracle.sol

Function: `_getQuote(uint256 inAmount, address base, address quote)`

This function retrieves pricing quotes from both primary and anchor oracles.

Inputs

- `inAmount`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None at this level.
  - **Impact**: The amount of base tokens to be converted.
- `base`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None at this level.
  - **Impact**: The token for which the price is being determined.
- `quote`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None at this level.
  - **Impact**: The token against which the price is measured.

Branches and code coverage

Intended branches

- Calculate the `lowerBound` and `upperBound` for the price quote from the `primaryOracle`.
  - ☑ Test coverage
- Check if the `anchorOracle` price is within the `lowerBound` and `upperBound`.
  - ☑ Test coverage

Negative behavior

- Should not allow an anchor price outside the `lowerBound` and `upperBound`.
  - ☑ Negative test

5.2. Module: AssetRegistry.sol

Function: `addAsset(address asset)`

This function adds assets to the asset registry.

Inputs

- `asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: The asset to add to the registry.

Branches and code coverage

Intended branches

- Add `asset` to `_assetList`.
  - ☑ Test coverage
- Emit an `AddAsset` event.
  - ☑ Test coverage
- Set `indexPlusOne` to the length of `_assetList` plus one.
  - ☑ Test coverage
- Set `status` to `AssetStatus.ENABLED`.
  - ☑ Test coverage
- Set the bit in `enabledAssets` corresponding to the index of the asset to 1.
  - ☑ Test coverage

Negative behavior

- Should not allow adding the same asset twice.
  - ☑ Negative test
- Should not allow adding the zero address.
  - ☑ Negative test
- Should not allow adding more than `_MAX_ASSETS` assets.
  - ☑ Negative test
- Caller needs to have `MANAGER_ROLE`.
  - ☑ Negative test

Function: `setAssetStatus(address asset, AssetStatus newStatus)`

This function allows setting the status of an asset in the registry.

Inputs

- `asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is not the zero address and that it is enabled.
  - **Impact**: The asset to set the status for.
- `newStatus`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is not the same as the current status.
  - **Impact**: The new status to set for the asset.

Branches and code coverage

Intended branches

- Set the `enabledAssets` bit flag based on the index of the asset in the registry.
  - ☑ Test coverage
- Update the status of the asset.
  - ☑ Test coverage

Negative behavior

- Caller needs to have `MANAGER_ROLE`.
  - ☑ Negative test
- Should not allow setting the status of a nonexistent asset.
  - ☑ Negative test
- Should not allow setting the status to the same status.
  - ☑ Negative test

5.3. Module: BasketManagerUtils.sol

Function: `completeRebalance(BasketManagerStorage self, ExternalTrade[] externalTrades, address[] baskets, uint64[][] basketTargetWeights)`

This function allows the completion of a rebalance for the given baskets. The rebalance can be completed if it has been more than 15 minutes since the last action.

Inputs

- `self`
  - **Control**: Fully controlled by the calling contract.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `externalTrades`

    - **Control**: Fully controlled by the calling contract.
    - **Constraints**: Ensured that the hash of the external trades matches the stored hash.
    - **Impact**: The external trades to be processed.
- `baskets`
    - **Control**: Fully controlled by the calling contract.
    - **Constraints**: Baskets' hash is validated.
    - **Impact**: The baskets to complete the rebalance for.
- `basketTargetWeights`
    - **Control**: Fully controlled by the calling contract.
    - **Constraints**: Validated against the baskets that are to be rebalanced.
    - **Impact**: The target weights for each basket.

Branches and code coverage

Intended branches

- Validate the basket hash against the storage one, so that no arbitrary trades are processed.
    - ☑ Test coverage
- Process the external trades if the rebalance status is `TOKEN_SWAP_EXECUTED`. This assumes that the `externalTrades` have already been validated by this point.
    - ☑ Test coverage
- Initialize the basket data.
    - ☑ Test coverage
- Validate the target weights if the retry count is less than `_MAX_RETRIES`. Otherwise, take the risk and continue with possibly unmet target weights.
    - ☑ Test coverage
- Increment the retry count, update the timestamp, reset the external trades hash, and set the status to `REBALANCE_PROPOSED` so that a new swap can be proposed again — in the case of unmet target weights.
    - ☑ Test coverage
- Finalize the rebalance if all checks pass.
    - ☑ Test coverage

Negative behavior

- Should not allow completing the rebalance if the status is not `TOKEN_SWAP_PROPOSED`, `REBALANCE_PROPOSED` or `TOKEN_SWAP_EXECUTED`.
    - ☑ Negative test
- Should not allow a `block.timestamp - rebalanceStatus.timestamp` to be less than 15 minutes.
    - ☑ Negative test
- Should revert if the rebalance status is `NOT_STARTED`.
    - ☑ Negative test

- Should revert if the external trades hash does not match the stored hash.
  - ☑ Negative test
- Should return early if the target weights are met and the retry count is less than `_MAX_RETRIES`.
  - ☑ Negative test

Function: `createNewBasket(BasketManagerStorage self, string basketName, string symbol, address baseAsset, uint256 bitFlag, address strategy)`

This function allows the creation of new baskets.

Inputs

- `self`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `basketName`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None at this level.
  - **Impact**: The basket's name.
- `symbol`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The symbol of the basket.
- `baseAsset`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Check if the base asset is not the zero address.
  - **Impact**: The base asset of the basket.
- `bitFlag`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Checked that the bit flag is supported by the strategy registry and that the asset is not paused.
  - **Impact**: The bit flag for the basket.
- `strategy`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Checked that the strategy is supported by the strategy registry.
  - **Impact**: The strategy for the basket.

Branches and code coverage

Intended branches

- Set the base asset index for the basket.
  - ☑ Test coverage
- Add the basket to the list of basket tokens.
  - ☑ Test coverage
- Set the basket assets and the basket ID.
  - ☑ Test coverage
- Set the management fee for the basket.
  - ☑ Test coverage
- Set the index for the basket assets.
  - ☑ Test coverage
- Clone the basket-token implementation.
  - ☑ Test coverage
- Initialize the basket token with the given parameters.
  - ☑ Test coverage

Negative behavior

- Should not allow `baseAsset` to be the zero address.
  - ☑ Negative test
- Should revert if the maximum number of basket tokens has been reached.
  - ☑ Negative test
- Should revert if the basket token already exists.
  - ☑ Negative test
- Should revert if the strategy registry does not support the strategy.
  - ☑ Negative test
- Should revert if the asset is not enabled.
  - ☑ Negative test
- Should revert if the asset list is empty.
  - ☑ Negative test

Function: `proRataRedeem(BasketManagerStorage self, uint256 totalSupplyBefore, uint256 burnedShares, address to)`

This function allows redemption of shares when no rebalancing is in process. Shares are redeemed for each underlying asset in the basket, rather than for the initial base asset.

Inputs

- `self`
  - **Control**: Fully controlled by the calling contract.

  - **Constraints**: None.
  - **Impact**: The storage reference.
- `totalSupplyBefore`
  - **Control**: Fully controlled by the calling contract.
  - **Constraints**: Checked to not be zero at this level.
  - **Impact**: The supply of basket tokens before burning.
- `burnedShares`
  - **Control**: Fully controlled by the calling contract.
  - **Constraints**: Ensured that it is not zero and that it is less than `totalSupplyBefore`.
  - **Impact**: The shares that have been burned.
- `to`
  - **Control**: Fully controlled by the calling contract.
  - **Constraints**: Checked to not be zero at this level.
  - **Impact**: The address to which the redeemed assets are sent.

Branches and code coverage

Intended branches

- Assume that parameters are legitimate, as they are controlled by the calling contract.
  ☑ Test coverage
- Calculate the amount to withdraw based on the `burnedShares`, current basket balance, and `totalSupplyBefore`.
  ☑ Test coverage
- Update the `basketBalanceOf` mapping for each asset in the basket based on the amount that is withdrawn.
  ☑ Test coverage
- Transfer the assets to the `to` address.
  ☑ Test coverage

Negative behavior

- Revert if `totalSupplyBefore` is zero.
  ☑ Negative test
- Revert if `burnedShares` is zero.
  ☑ Negative test
- Revert if `burnedShares` is greater than `totalSupplyBefore`.
  ☑ Negative test
- Revert if `to` is the zero address.
  ☑ Negative test
- Revert if the basket is currently rebalancing.
  ☑ Negative test
- Should not transfer any assets if the amount to withdraw is zero.

☑ Negative test

Function: `proposeRebalance(BasketManagerStorage self, address[] baskets)`

This function allows proposing a rebalance for the given baskets.

Inputs

- `self`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `baskets`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None at this level.
  - **Impact**: The baskets that are to be rebalanced.

Branches and code coverage

Intended branches

- Set the `shouldRebalance` flag to true if the `pendingDeposits` are greater than zero.
  ☑ Test coverage
- Assume that all baskets within the baskets array are to maintain the same `shouldRebalance` flag. Alternatively, remove the baskets that do not actually require rebalancing from the proposal.
  ☑ Test coverage
- Set the basket mask to the bit mask of the baskets.
  ☑ Test coverage
- Set the timestamp to the current block timestamp.
  ☑ Test coverage
- Set the rebalance status to `REBALANCE_PROPOSED`.
  ☑ Test coverage
- Calculate the basket value and notify the basket token of the rebalance.
  ☑ Test coverage
- Set the `shouldRebalance` flag to true if the pending deposits are greater than zero.
  ☑ Test coverage
- Store the pending redeem value in storage.
  ☑ Test coverage
- Determine whether rebalance is required and set the `shouldRebalance` flag accordingly. Revert otherwise.

- ☑ Test coverage
- Update the basket hash with the array of baskets that need rebalancing and their target weights.
  - ☑ Test coverage

Negative behavior

- Should revert if rebalance is not required. If not all `shouldRebalance` flags are set to true, the function should revert.
  - ☑ Negative test
- Should not allow proposing a rebalance if the status is not `NOT_STARTED`.
  - ☑ Negative test
- Should not allow proposing a rebalance if the time since the last action is less than `_REBALANCE_COOLDOWN_SEC`.
  - ☑ Negative test
- Should not allow proposing a rebalance if the basket does not have any assets.
  - ☑ Negative test
- Should not allow proposing a rebalance if the basket has paused assets.
  - ☑ Negative test

Function: `proposeTokenSwap(BasketManagerStorage self, InternalTrade[] internalTrades, ExternalTrade[] externalTrades, address[] baskets, uint64[][] basketTargetWeights)`

This function allows the proposal of token swaps to rebalance the given baskets. If the proposed token swap results are not close to the target balances, this function will revert.

Inputs

- `self`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `internalTrades`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None at this level — some checks are performed in `_processInternalTrades`.
  - **Impact**: The list of internal trades to be processed. Assumed to be correct and valid as well as most optimally ordered.
- `externalTrades`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None at this level — some checks are performed in `_validateExternalTrades`.

    - **Impact**: The list of external trades to be processed. Assumed to be correct and valid as well as most optimally ordered.
- `baskets`
    - **Control**: Fully controlled by the calling function.
    - **Constraints**: The baskets' hash and target weights are validated against the stored `basketHash`.
    - **Impact**: The lists of baskets for which the token swap is proposed. Assumed to be correct and valid.
- `basketTargetWeights`
    - **Control**: Fully controlled by the calling function.
    - **Constraints**: The target weights are validated against the baskets that are to be rebalanced.
    - **Impact**: The target weights for each basket. Assumed to be correct and valid.

Branches and code coverage

Intended branches

- Validate the basket hash against the storage one so that no arbitrary trades are processed.
    - ☑ Test coverage
- Initialize the basket data.
    - ☑ Test coverage
- Update the storage with the new rebalance status as well as the external trades hash.
    - ☑ Test coverage
- Process the internal trades and validate the external ones.
    - ☑ Test coverage
- Estimate whether the target weights would theoretically be met.
    - ☑ Test coverage

Negative behavior

- Should not allow empty internal and external trades.
    - ☑ Negative test
- Should not allow proposing a token swap if the status is not `REBALANCE_PROPOSED`.
    - ☑ Negative test
- Should revert if the target weights are not met.
    - ☑ Negative test

Function: `_finalizeRebalance(BasketManagerStorage self, address[] baskets)`

This function performs the finalization of the rebalance process.

Inputs

- `self`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `baskets`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The basket addresses to finalize the rebalance for.

Branches and code coverage

Intended branches

- Reset the `self.rebalanceStatus.baskethash` to zero.
  - ☑ Test coverage
- Reset the `self.rebalanceStatus.basketMask` to zero.
  - ☑ Test coverage
- Increment the `self.rebalanceStatus.epoch` by one.
  - ☑ Test coverage
- Set the `self.rebalanceStatus.timestamp` to the current block timestamp.
  - ☑ Test coverage
- Set the `self.rebalanceStatus.status` to `Status.NOT_STARTED`.
  - ☑ Test coverage
- Reset the `self.externalTradesHash` to zero.
  - ☑ Test coverage
- Reset the `self.retryCount` to zero.
  - ☑ Test coverage
- Update the `self.basketBalanceOf` for each asset in the basket.
  - ☑ Test coverage
- Call `BasketToken.fulfillRedeem` for each asset in the basket if there are withdrawable funds. Otherwise, call `BasketToken.fallbackRedeemTrigger`.
  - ☑ Test coverage

Negative behavior

- Should not be callable directly.
  - ☑ Negative test

Function: `_processExternalTrades(BasketManagerStorage self, ExternalTrade[] externalTrades)`

This function handles the processing of external trades.

Inputs

- `self`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `externalTrades`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None. Assumed to be validated by upper-level functions.
  - **Impact**: The external trades to be processed.

Branches and code coverage

Intended branches

- Assume that the trades are most efficiently ordered. This is not enforced in code, and responsibility for this comes from the proposer.
- Finalize the `tokenSwap` via CoW Swap. (Handled in the `_completeTokenSwap` call.)
  - ☑ Test coverage
- Update the basket balance of every basket with the results of the trades if applicable.
  - ☑ Test coverage

Negative behavior

- Should not be directly callable by external users.
  - ☑ Negative test

Function: `_processInternalTrades(BasketManagerStorage self, InternalTrade[] internalTrades, address[] baskets, uint256[][] basketBalances)`

This function handles the processing of internal trades.

Inputs

- `self`
  - **Control**: Controlled by the `proposeTokenSwap` function.
  - **Constraints**: None.
  - **Impact**: The storage reference.
- `internalTrades`
  - **Control**: Controlled by the `proposeTokenSwap` function.
  - **Constraints**: None at this level.
  - **Impact**: The internal trades to be processed.
- `baskets`

    - **Control**: Controlled by the `proposeTokenSwap` function.
    - **Constraints**: None at this level.
    - **Impact**: The baskets that are being rebalanced.
- `basketBalances`
    - **Control**: Controlled by the `proposeTokenSwap` function.
    - **Constraints**: None at this level.
    - **Impact**: The basket's balances.

Branches and code coverage

Intended branches

- Assume that the trades are most efficiently ordered. This is not enforced in code, and responsibility for this comes from the proposer.
- Calculate the fee on the `sellAmount` (if any).
    - ☑ Test coverage
- Calculate the fee on the `buyAmount` (if any).
    - ☑ Test coverage
- Calculate the initial `buyAmount` based on `netSellAmount` through the EulerRouter.
    - ☑ Test coverage
- Update the basket balances based on the trade.
    - ☑ Test coverage

Negative behavior

- Should not allow the `initialBuyAmount` to be greater than the balance of the token in the basket.
    - ☑ Negative test
- Should not allow having a `netBuyAmount` less than the `minAmount` or greater than the `maxAmount`.
    - ☑ Negative test
- Should not allow the `sellAmount` to be greater than the balance of the token in the basket.
    - ☑ Negative test

5.4. Module: BasketToken.sol

Function: `cancelDepositRequest()`

This function cancels a pending deposit request and transfers the assets back to the user.

Branches and code coverage

Intended branches

- Update `depositRequest`.
  - ☑ Test coverage
- Transfer `pendingDeposit` assets to the user.
  - ☑ Test coverage

Negative behavior

- Revert if the current request has no pending deposits.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), msg.sender, pendingDeposit)`
  - **What is controllable?** Can increase `pendingDeposit` by depositing more assets.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `cancelRedeemRequest()`

This function cancels a pending redeem request and transfers the shares back to the user.

Branches and code coverage

Intended branches

- Update `redeemRequest`.
  - ☑ Test coverage
- Transfer `pendingRedeem` shares to the user.
  - ☑ Test coverage

Negative behavior

- Revert if the current request has no pending redemptions.
  - ☑ Negative test

Function call analysis

- `this._transfer(address(this), msg.sender, pendingRedeem) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `msg.sender` and `pendingRedeem`.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `msg.sender`'s plugins, but `ERC20PluginsUpgradeable._update` has a reentrancy guard.

Function: `claimFallbackShares(address receiver, address controller)`

This function allows the basket manager to claim shares given for a previous redemption request in the event a redemption fulfillment for a given epoch fails.

Inputs

- `receiver`
  - **Control**: Fully controlled by the basket manager.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the shares.
- `controller`
  - **Control**: Fully controlled by the basket manager.
  - **Constraints**: None at this level.
  - **Impact**: The address of the controller of the redemption request.

Branches and code coverage

Intended branches

- Set `controller`'s `redeemShares` of the current request to zero.
  - ☑ Test coverage
- Transfer `shares` to `receiver`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager or an operator.
  - ☑ Negative test
- Revert if the claimable shares are zero.
  - ☑ Negative test

Function call analysis

- `this._transfer(address(this), receiver, shares) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from,`

`address to, uint256 amount)`

- **What is controllable?** `receiver` and `shares` — `shares` is partially controllable by the caller.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `receiver`'s plug-ins, but this function follows the checks-effects-interactions pattern.

Function: `claimFallbackShares()`

This function wraps the `claimFallbackShares` function to allow the caller to claim their own fallback shares.

Branches and code coverage

Intended branches

- Call `claimFallbackShares` with `msg.sender` as both `receiver` and `controller`.
  - ☑ Test coverage

Function: `deposit(uint256 assets, address receiver, address controller)`

This function transfers a user's shares owed for a previously fulfilled deposit request.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This must match the claimable deposit assets.
  - **Impact**: The amount of assets previously requested for deposit.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the shares.
- `controller`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must be the controller or an operator of the controller.
  - **Impact**: The address of the controller of the deposit request.

Branches and code coverage

Intended branches

- Set `depositRequest.depositAssets[controller]` to zero.
  - ☑ Test coverage
- Transfer `shares` to `receiver`.
  - ☑ Test coverage

Negative behavior

- Revert if `assets` is zero.
  - ☑ Negative test
- Revert if the caller is not the controller or an operator of the controller.
  - ☑ Negative test
- Revert if `assets` does not match the claimable deposit assets.
  - ☑ Negative test

Function call analysis

- `this._claimDeposit(depositRequest, assets, shares, receiver, controller) -> this._transfer(address(this), receiver, shares) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `assets`, `receiver`, and `controller`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `receiver`'s plug-ins, but this function follows the checks-effects-interactions pattern.

Function: `deposit(uint256 assets, address receiver)`

This function wraps the `deposit` function to allow the caller to deposit assets without specifying a controller.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The amount of assets to be claimed.
- `receiver`

- **Control**: Fully controlled by the caller.
- **Constraints**: None at this level.
- **Impact**: The address to receive the shares.

Branches and code coverage

Intended branches

- Call `deposit` with `assets`, `receiver`, and `msg.sender` as arguments.
  - ☑ Test coverage

Function: `fallbackRedeemTrigger()`

This function allows the basket manager to trigger a fallback redemption in the event of a failed redemption fulfillment. It allows users to claim their shares back for a redemption in the future and advances the redemption epoch.

Branches and code coverage

Intended branches

- Update `redeemRequest`.
  - ☑ Test coverage
- Set `fallbackTriggered` to true.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager.
  - ☑ Negative test
- Revert if the current request has already been fallbacked.
  - ☑ Negative test
- Revert if the current request has already been fulfilled.
  - ☑ Negative test
- Revert if the current request has no pending redemptions.
  - ☑ Negative test

Function: `fulfillDeposit(uint256 shares)`

This function allows the basket manager to fulfill all pending deposit assets for the current request. The assets are transferred to the basket manager.

Inputs

- `shares`
  - **Control**: Fully controlled by the basket manager.
  - **Constraints**: Nonzero.
  - **Impact**: The amount of shares the current deposit will be fulfilled with.

Branches and code coverage

Intended branches

- Set `fulfilledShares` of the current request to `shares`.
  - ☑ Test coverage
- Mint `shares` to the basket.
  - ☑ Test coverage
- Transfer total deposit assets to the basket manager.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager.
  - ☑ Negative test
- Revert if `depositRequest.totalDepositAssets` is zero.
  - ☑ Negative test
- Revert if `shares` is zero.
  - ☑ Negative test
- Revert if the current request has already been fulfilled.
  - ☑ Negative test

Function call analysis

- this._mint(address(this), shares) -> this._update(address(0), account, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)
  - **What is controllable?** `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- SafeERC20.safeTransfer(IERC20(this.asset()), msg.sender, assets)
  - **What is controllable?** Can increase `assets` by depositing more assets.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `fulfillRedeem(uint256 assets)`

This function allows the basket manager to fulfill all pending redeem requests for the current request. The shares in this contract are burned, and the assets are transferred to the basket manager.

Inputs

- `assets`
  - **Control**: Fully controlled by the basket manager.
  - **Constraints**: Nonzero.
  - **Impact**: The amount of assets the current redemption will be fulfilled with.

Branches and code coverage

Intended branches

- Set `fulfilledAssets` of the current request to `assets`.
  - ☑ Test coverage
- Burn `sharesPendingRedemption` shares from the basket.
  - ☑ Test coverage
- Transfer `assets` to the basket manager.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager.
  - ☑ Negative test
- Revert if `sharesPendingRedemption` is zero.
  - ☑ Negative test
- Revert if `assets` is zero.
  - ☑ Negative test
- Revert if the current request has already been fulfilled.
  - ☑ Negative test

Function call analysis

- `this._burn(address(this), sharesPendingRedemption) -> this._update(account, address(0), value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(addres plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `sharesPendingRedemption`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `SafeERC20.safeTransferFrom(IERC20(this.asset()), msg.sender, address(this), assets)`
  - **What is controllable?** Can increase `assets` by redeeming more shares.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `mint(uint256 shares, address receiver, address controller)`

This function transfers a user's shares owed for a previously fulfilled deposit request.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This must match `_maxMint(fulfilledShares, depositAssets, depositRequest.totalDepositAssets)`.
  - **Impact**: The amount of shares to receive.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the shares.
- `controller`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must be the controller or an operator of the controller.
  - **Impact**: The address of the controller of the deposit request.

Branches and code coverage

Intended branches

- Set `depositRequest.depositAssets[controller]` to zero.
  - ☑ Test coverage
- Transfer `shares` to `receiver`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller or an operator of the controller.
  - ☑ Negative test
- Revert if `shares` does not match `_maxMint(fulfilledShares, depositAssets, depositRequest.totalDepositAssets)`.
  - ☑ Negative test

Function call analysis

- `this._claimDeposit(depositRequest, assets, shares, receiver, controller) -> this._transfer(address(this), receiver, shares) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `shares`, `receiver`, and `controller`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `receiver`'s plug-ins, but this function follows the checks-effects-interactions pattern.

Function: `mint(uint256 shares, address receiver)`

This function wraps the `mint` function to allow the caller to mint shares without specifying a controller.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The amount of shares to receive.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the shares.

Branches and code coverage

Intended branches

- Call `mint` with `shares`, `receiver`, and `msg.sender` as arguments.
  - ☑ Test coverage

Function call analysis

- `this.mint(shares, receiver, msg.sender) -> this._claimDeposit(depositRequest, assets, shares, receiver, controller) -> this._transfer(address(this), receiver, shares) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update ->`

`ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`

- **What is controllable?** `shares`, `receiver`, and `controller`.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `receiver`'s plug-ins, but this function follows the checks-effects-interactions pattern.

Function: `prepareForRebalance(uint16 feeBps, address feeCollector)`

This function advances the deposit and redeem epochs. It ensures that the previous deposit and redeem requests have been fulfilled before advancing the epochs. It also harvests the management fee.

Inputs

- `feeBps`
  - **Control**: Controllable by the basket manager.
  - **Constraints**: The fee must be less than or equal to 3,000.
  - **Impact**: The management fee in basis points to be harvested.
- `feeCollector`
  - **Control**: Controllable by the basket manager.
  - **Constraints**: None at this level.
  - **Impact**: The address that will receive the harvested management fee.

Branches and code coverage

Intended branches

- Increase `nextDepositRequestId` by two.
  - ☑ Test coverage
- Increase `nextRedeemRequestId` by two.
  - ☑ Test coverage
- Call `_harvestManagementFee(feeBps, feeCollector)`.
  - ☑ Test coverage
- Update `lastManagementFeeHarvestTimestamp` to `block.timestamp`.
  - ☑ Test coverage
- Mint `fee` to `feeCollector`.
  - ☑ Test coverage
- Call `FeeCollector(feeCollector).notifyHarvestFee(fee)`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager.
  - ☑ Negative test
- Revert if `previousDepositRequest.totalDepositAssets` is greater than zero and `previousDepositRequest.fulfilledShares` is zero.
  - ☑ Negative test
- Revert if `previousRedeemRequest.totalRedeemShares` is greater than zero, `previousRedeemRequest.fulfilledAssets` is zero, and `previousRedeemRequest.fallbackTriggered` is false.
  - ☑ Negative test
- Revert if `feeBps` is greater than 3,000.
  - ☑ Negative test

Function call analysis

- `this._harvestManagementFee(feeBps, feeCollector)` -> `this._mint(feeCollector, fee)` -> `this._update(address(0), account, value)` -> `ERC20PluginsUpgradeable._update`
  - **What is controllable?** `feeCollector` and `fee` are controllable by the basket manager.
  - **If the return value is controllable, how is it used and how can it go wrong?** `feeBps` and `feeCollector` are controllable by the basket manager.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `owner`'s plug-ins before updating claimable fees in FeeCollector, but `ERC20PluginsUpgradeable._update` has a reentrancy guard.
- `this._harvestManagementFee(feeBps, feeCollector)` -> `FeeCollector(feeCollector).notifyHarvestFee(fee)`
  - **What is controllable?** `feeBps` and `feeCollector` are controllable by the basket manager.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `proRataRedeem(uint256 shares, address to, address from)`

This function enables the caller to immediately redeem shares for all assets associated with this basket. This is a synchronous operation and does not require the rebalance process to be completed.

Inputs

- `shares`

- **Control**: Fully controlled by the caller.
- **Constraints**: None at this level.
- **Impact**: The amount of shares to redeem.

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to which the redeemed assets are sent.
- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: If `from` is not the caller, the caller must be approved by `from`.
  - **Impact**: The address to redeem shares from.

Branches and code coverage

Intended branches

- Call `_harvestManagementFee` with `feeBps` and `feeCollector`.
  - ☑ Test coverage
- Call `_spendAllowance` with `from` as `owner` and `msg.sender` as `spender`.
  - ☑ Test coverage
- Burn `shares` from the basket.
  - ☑ Test coverage
- Call `proRataRedeem` with `shares` and `to` as arguments.
  - ☑ Test coverage

Function call analysis

- `BasketManager(this.basketManager).managementFee(address(this))`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `BasketManager(this.basketManager).feeCollector()`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._harvestManagementFee(feeBps, feeCollector)` -> `this._mint(feeCollector, fee)` -> `this._update(address(0), account, value)` -> `ERC20PluginsUpgradeable._update` -> `ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `feeCollector` and `fee` are controllable by the basket

manager.

- **If the return value is controllable, how is it used and how can it go wrong?** `feeBps` and `feeCollector` are controllable by the basket manager.
- **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `owner`'s plugins before updating claimable fees in FeeCollector but ERC20PluginsUpgradeable._update has reentrancy guard.

- `this._harvestManagementFee(feeBps, feeCollector) -> FeeCollector(feeCollector).notifyHarvestFee(fee)`
  - **What is controllable?** `feeBps` and `feeCollector` are controllable by the basket manager.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `_burn(from, shares) -> this._update(account, address(0), value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `from` and `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `from`'s plugins before calling `proRataRedeem` which decreases asset balances. It can be used to manipulate the price of this LP token.
- `BasketManager(this.basketManager).proRataRedeem(totalSupplyBefore, shares, to)`
  - **What is controllable?** `totalSupplyBefore`, `shares`, and `to` — `totalSupplyBefore` can be partially controllable by depositing or redeeming huge shares.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeem(uint256 shares, address receiver, address controller)`

This function transfers a user's assets owed for a previously fulfilled redemption request.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This must match the claimable redeem request.
  - **Impact**: The amount of shares to be claimed.

- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the assets.
- `controller`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must be the controller or an operator of the controller.
  - **Impact**: The address of the controller of the redeem request.

Branches and code coverage

Intended branches

- Set `redeemRequest.redeemShares[controller]` to zero.
  - ☑ Test coverage
- Transfer `assets` to `receiver`.
  - ☑ Test coverage

Negative behavior

- Revert if `shares` is zero.
  - ☑ Negative test
- Revert if the caller is not the controller or an operator of the controller.
  - ☑ Negative test
- Revert if `shares` does not match the claimable redeem request.
  - ☑ Negative test

Function call analysis

- `this._claimRedemption(redeemRequest, assets, shares, receiver, controller) -> SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** `assets`, `receiver`, and `controller` — `shares` is partially controllable by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `requestDeposit(uint256 assets, address controller, address owner)`

This function allows a user to request a deposit of assets to the basket. The assets are immediately transferred to this contract.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Nonzero.
  - **Impact**: The amount of assets to deposit.
- `controller`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `controller` must not have the last pending and claimable deposit request.
  - **Impact**: The address of the controller of the deposit request being created.
- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address of the owner of the assets being deposited.

Branches and code coverage

Intended branches

- Check `AssetRegistry(assetRegistry).hasPausedAssets(bitFlag)`.
  - ☑ Test coverage
- Update `depositRequest` and `lastDepositRequestId` mappings.
  - ☑ Test coverage
- Transfer `owner`'s `assets` to this contract.
  - ☑ Test coverage

Negative behavior

- Revert if `assets` is zero.
  - ☑ Negative test
- Revert if `controller` has the last pending deposit request.
  - ☑ Negative test
- Revert if `controller` has the last claimable deposit request.
  - ☑ Negative test
- Revert if any of the assets in the `bitFlag` are paused.
  - ☑ Negative test

Function call analysis

- `AssetRegistry(this.assetRegistry).hasPausedAssets(this.bitFlag)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** A user can request a deposit of paused assets.

    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20.safeTransferFrom(IERC20(this.asset()), owner, address(this), assets)`
    - **What is controllable?** `owner` and `assets`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `requestRedeem(uint256 shares, address controller, address owner)`

This function allows a user to request a redemption of shares from the basket. The shares are immediately transferred to this contract.

Inputs

- `shares`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Nonzero.
    - **Impact**: The amount of shares to redeem.
- `controller`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The `controller` must not have the last pending/claimable redeem request and claimable fallback shares.
    - **Impact**: The address of the controller of the redeem request being created.
- `owner`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None at this level.
    - **Impact**: The address of the owner of the redeemed shares.

Branches and code coverage

Intended branches

- Check `AssetRegistry(assetRegistry).hasPausedAssets(bitFlag)`.
    - ☑ Test coverage
- Call `_spendAllowance(owner, msg.sender, shares)` if `msg.sender` is not `owner` and `owner` is not an operator of `msg.sender`.
    - ☑ Test coverage
- Update `redeemRequest` and `lastRedeemRequestId` mappings.
    - ☑ Test coverage
- Transfer `owner`'s `shares` to this contract.

- ☑ Test coverage

Negative behavior

- Revert if `shares` is zero.
  - ☑ Negative test
- Revert if `controller` has the last pending redeem request.
  - ☑ Negative test
- Revert if `controller` has the last claimable redeem request.
  - ☑ Negative test
- Revert if `controller` has claimable fallback shares.
  - ☑ Negative test
- Revert if any of the assets in the `bitFlag` are paused.
  - ☑ Negative test

Function call analysis

- `AssetRegistry(this.assetRegistry).hasPausedAssets(this.bitFlag)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** A user can request a deposit for paused assets.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._transfer(owner, address(this), shares) -> this._update(from, to, value) -> ERC20PluginsUpgradeable._update -> ERC20PluginsUpgradeable._updateBalances(address plugin, address from, address to, uint256 amount)`
  - **What is controllable?** `owner` and `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can reenter the contract while updating balances calling `owner`'s plug-ins, but `ERC20PluginsUpgradeable._update` has a reentrancy guard.

Function: `setBitFlag(uint256 bitFlag_)`

This function allows the basket manager to set a new bit flag for this basket.

Inputs

- `bitFlag_`
  - **Control**: Fully controlled by the basket manager.
  - **Constraints**: None at this level.
  - **Impact**: The new bit flag for this basket.

Branches and code coverage

Intended branches

- Update `bitFlag` to a new value.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the basket manager.
  - ☑ Negative test

Function: `setOperator(address operator, bool approved)`

This function sets an operator's ability to act on behalf of a controller.

Inputs

- `operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address of the operator to set the status for.
- `approved`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The status of the operator.

Branches and code coverage

Intended branches

- Set `msg.sender`'s status in `isOperator` for `operator` to `approved`.
  - ☑ Test coverage

Function: `withdraw(uint256 assets, address receiver, address controller)`

This function transfers a user's assets owed for a previously fulfilled redemption request.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.

- **Constraints**: This must match `_maxWithdraw(fulfilledAssets, redeemShares, redeemRequest.totalRedeemShares)`.
- **Impact**: The amount of assets previously requested for redemption.

- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to receive the assets.
- `controller`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must be the controller or an operator of the controller.
  - **Impact**: The address of the controller of the redeem request.

Branches and code coverage

Intended branches

- Set `redeemRequest.redeemShares[controller]` to zero.
  - ☑ Test coverage
- Transfer `assets` to `receiver`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller or an operator of the controller.
  - ☑ Negative test
- Revert if `assets` does not match `_maxWithdraw(fulfilledAssets, redeemShares, redeemRequest.totalRedeemShares)`.
  - ☑ Negative test

Function call analysis

- `this._claimRedemption(redeemRequest, assets, shares, receiver, controller) -> SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** `assets`, `receiver`, and `controller` — `shares` is partially controllable by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.5. Module: CoWSwapAdapter.sol

Function: `completeTokenSwap(ExternalTrade[] externalTrades)`

This function completes the token swaps by claiming the tokens from the CoWSwapClone contracts.

Inputs

- `externalTrades`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The external trades that were executed are used to claim the tokens.

Branches and code coverage

Intended branches

- Call `claim` on each CoWSwapClone contract.
  - ☑ Test coverage

Function call analysis

- `CoWSwapClone(swapContract).claim()`
  - **What is controllable?** `swapContract`. The address is determined by the `keccak256` hash of the external trade details.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is used to store the claimed amounts of the sell and buy tokens. It is used to emit an event.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `executeTokenSwap(ExternalTrade[] externalTrades, bytes)`

This function is used to execute a series of token swaps by creating orders on the CoWSwap protocol.

Inputs

- `externalTrades`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The external trades to be used for the token swap.

Branches and code coverage

Intended branches

- Call `_createOrder` for each external trade.
  - ☑ Test coverage

5.6. Module: CoWSwapClone.sol

Function: `initialize()`

This function initializes this contract by approving the vault relayer to spend the maximum amount of the sell token.

Branches and code coverage

Intended branches

- Call `IERC20(sellToken()).forceApprove(_VAULT_RELAYER, type(uint256).max)`.
  - ☑ Test coverage

5.7. Module: FeeCollector.sol

Function: `claimSponsorFee(address basketToken)`

This function is used to claim the sponsor fee for a given basket token.

Inputs

- `basketToken`
  - **Control**: Fully controlled by the sponsor or admin.
  - **Constraints**: `basketToken` must have the `_BASKET_TOKEN_ROLE` role.
  - **Impact**: The basket token to claim the sponsor fee for.

Branches and code coverage

Intended branches

- Update `claimableSponsorFees` for the given `basketToken` with zero.
  - ☑ Test coverage
- Call `BasketToken(basketToken).proRataRedeem(fee, sponsor, address(this))`.
  - ☑ Test coverage

Negative behavior

- Revert if `basketToken` does not have the `_BASKET_TOKEN_ROLE` role.
  - ☑ Negative test
- Revert if the caller is not the sponsor or an admin.
  - ☑ Negative test

Function call analysis

- `this._checkIfBasketToken(basketToken) -> this._basketManager.hasRole(FeeCollecto token)`
  - **What is controllable?** `basketToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `BasketToken(basketToken).proRataRedeem(fee, sponsor, address(this))`
  - **What is controllable?** `basketToken` and `sponsor`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `claimTreasuryFee(address basketToken)`

This function is used to claim the treasury fee for a given basket token.

Inputs

- `basketToken`
  - **Control**: Fully controlled by the protocol treasury or admin.
  - **Constraints**: `basketToken` must have the `_BASKET_TOKEN_ROLE` role.
  - **Impact**: The basket token to claim the treasury fee for.

Branches and code coverage

Intended branches

- Update `claimableTreasuryFees` for the given `basketToken` with zero.
  - ☑ Test coverage
- Call `BasketToken(basketToken).proRataRedeem(fee, protocolTreasury, address(this))`.
  - ☑ Test coverage

Negative behavior

- Revert if caller is not the protocol treasury or not an admin.
  - ☑ Negative test
- Revert if `basketToken` does not have the `_BASKET_TOKEN_ROLE` role.
  - ☑ Negative test

Function call analysis

- `this._checkIfBasketToken(basketToken) -> this._basketManager.hasRole(FeeCollecto token)`
  - **What is controllable?** `basketToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `BasketToken(basketToken).proRataRedeem(fee, this.protocolTreasury, address(this))`
  - **What is controllable?** `basketToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `notifyHarvestFee(uint256 shares)`

This function is used to notify the FeeCollector of the fees collected from the basket token and increase the claimable fees for the sponsor and treasury.

Inputs

- `shares`
  - **Control**: Fully controlled by the basket token.
  - **Constraints**: None.
  - **Impact**: The amount of shares to be collected.

Branches and code coverage

Intended branches

- Update `claimableSponsorFees` for the given `basketToken` with the `sponsorFee` split by the `sponsorFeeSplit`.
  - ☑ Test coverage
- Update `claimableTreasuryFees` for the given `basketToken` with the remaining `shares` after the sponsor fee split.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not a basket token.
  - ☑ Negative test

Function call analysis

- `this._checkIfBasketToken(basketToken) -> this._basketManager.hasRole(FeeCollecto token)`
  - **What is controllable?** `basketToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setProtocolTreasury(address treasury)`

This function is used to set the protocol treasury address.

Inputs

- `treasury`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is not the zero address.
  - **Impact**: The new treasury address.

Branches and code coverage

Intended branches

- Update the protocol treasury address.
  - ☑ Test coverage

Negative behavior

- Revert if `treasury` is the zero address.
  - ☑ Negative test

Function: `setSponsor(address basketToken, address sponsor)`

This function is used to set the sponsor for a given basket token.

Inputs

- `basketToken`
  - **Control**: Fully controlled by the admin.
  - **Constraints**: `basketToken` must have the `_BASKET_TOKEN_ROLE` role.
  - **Impact**: The basket token to set the sponsor for.
- `sponsor`

- **Control**: Fully controlled by the admin.
- **Constraints**: None.
- **Impact**: The new sponsor address.

Branches and code coverage

Intended branches

- Update `basketTokenSponsors` for the given `basketToken` with the new `sponsor`.
  - ☑ Test coverage

Negative behavior

- Revert if `basketToken` does not have the `_BASKET_TOKEN_ROLE` role.
  - ☑ Negative test

Function call analysis

- `this._checkIfBasketToken(basketToken) -> this._basketManager.hasRole(FeeCollecto token)`
  - **What is controllable?** `basketToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.8. Module: ManagedWeightStrategy.sol

Function: `setTargetWeights(uint256 bitFlag, uint64[] newTargetWeights)`

This function allows setting the target weights for assets.

Inputs

- `bitFlag`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. The bit flag must represent at least two assets.
  - **Impact**: The bit flag represents the assets.
- `newTargetWeights`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The length of the array must match the number of assets.
  - **Impact**: The new target weights for the assets.

Branches and code coverage

Intended branches

- Update the `targetWeights` mapping with the new weights.
  - ☑ Test coverage
- Update the `lastUpdated` mapping with the new epoch and timestamp.
  - ☑ Test coverage

Negative behavior

- Should only be callable by the manager role.
  - ☑ Negative test
- Should not allow bit flags with less than two assets.
  - ☑ Negative test
- Should not allow a weights array with a length different from the number of assets.
  - ☑ Negative test
- Should not allow a weights array with a sum different from the precision.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet.

During our assessment on the scoped Cove contracts, we discovered nine findings. Two critical issues were found. Three were of high impact, two were of medium impact, one was of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.