Cove Security Review

Pashov Audit Group

Conducted by: Said, ZeroTrust01, samuraii77

December 30th 2024 - January 16th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Introduction

A time-boxed security review of the **Storm-Labs-Inc/cove-contracts-core** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Cove

Cove is a system for automated asset management that maximizes returns by using Programmatic Orders to rebalance portfolios efficiently, eliminating loss-versus-rebalancing (LVR) through off-chain trade matching and optimized execution via CoW Swap. It leverages ERC-4626 tokenized vaults to aggregate deposits, minimize slippage and MEV, and provide expert-curated strategies for seamless onchain yield generation.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

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

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 6b607d137f898c0f421b4ba4e748f41b09b41518

fixes review commit hash - 5d917e9026c2d6072c56d4e1f8e8c2bd69f9124e

Scope

The following smart contracts were in scope of the audit:

- `AnchoredOracle`
- `AssetRegistry`
- `BasketManager`
- `BasketToken`
- `FeeCollector`
- `Rescuable`
- `BasketManagerStorage`
- `Trades`
- `CoWSwapAdapter`
- `CoWSwapClone`
- `TokenSwapAdapter`
- `AutomaticWeightStrategy`
- `ManagedWeightStrategy`
- `StrategyRegistry`
- `WeightStrategy`
- `BasketManagerUtils`
- `BitFlag`
- `Errors`
- `MathUtils`

7. Executive Summary

Over the course of the security review, Said, ZeroTrust01, samuraii77 engaged with Cove to review Cove. In this period of time a total of **23** issues were uncovered.

Protocol Summary

Protocol Name	Cove
Repository	https://github.com/Storm-Labs-Inc/cove-contracts-core
Date	December 30th 2024 - January 16th 2025
Protocol Type	Yield farming management

Findings Count

Severity	**Amount**
High	2
Medium	5
Low	16
Total Findings	**23**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect basket USD value will cause incorrect results	High	Resolved
[H-02]	The target weight check does not account for redeem request assets	High	Resolved
[M-01]	Missing ERC20Permit init call in initialize()	Medium	Resolved
[M-02]	Loss of surplus if ERC-1271 order allows arbitrary data	Medium	Resolved
[M-03]	Deviation check upon validating external trades is unidirectional	Medium	Resolved
[M-04]	orderValidTo assumes stepDelay is always 15 minutes	Medium	Resolved
[M-05]	updateBitFlag does not update the index of the base asset	Medium	Resolved
[L-01]	ManagementFee calculation does not match the documentation	Low	Resolved
[L-02]	trade.maxAmount check in the _processInternalTrades()	Low	Resolved
[L-03]	proRataRedeem() lacks a check to verify if msg.sender is the operator	Low	Resolved
[L-04]	Fully claiming fees requires a batch of 2 claims	Low	Resolved
[L-05]	Hardcoded slippage is dangerous	Low	Resolved
[L-06]	Lack of restriction on BasketManager's execute	Low	Resolved

[L-07]	executeTokenSwap can be called with empty externalTrades	Low	Resolved
[L-08]	External trades tradeOwnership is not validated	Low	Resolved
[L-09]	The rebalance requirement check is too sensitive	Low	Resolved
[L-10]	_harvestManagementFee does not consider claimable fallback shares	Low	Acknowledged
[L-11]	Malicious users can decrease management fees accrual	Low	Resolved
[L-12]	Management fees might not accrue	Low	Resolved
[L-13]	Calling mint or withdraw with 0 shares/assets resulting in a loss	Low	Acknowledged
[L-14]	AutomaticWeightStrategy is not fully implemented	Low	Acknowledged
[L-15]	Not transferring already collected claimableSponsorFees	Low	Resolved
[L-16]	Precision loss in _processExternalTrades()	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Incorrect basket USD value will cause incorrect results

Severity

Impact: Medium

Likelihood: High

Description

Upon proposing a token swap, we have the following sequence of function calls:

```
uint256[] memory totalValues = new uint256[](numBaskets);
// 2d array of asset balances for each basket
uint256[][] memory basketBalances = new uint256[][](numBaskets);
_initializeBasketData(self, baskets, basketAssets, basketBalances, totalValues);
// NOTE: for rebalance retries the internal trades must be updated as well
_processInternalTrades(self, internalTrades, baskets, basketBalances);
_validateExternalTrades
  (self, externalTrades, baskets, totalValues, basketBalances);
if (!_isTargetWeightMet(
  !_isTargetWeightMet

)
         revert TargetWeightsNotMet();
}
```

The `totalValues` array is the total USD value of a basket, a basket per element of the array. It is populated upon calling `_initialBasketData()`, then it is provided in `_validateExternalTrades()` where the array is manipulated based on the trades. Afterwards, it is used upon checking the deviation in `_isTargetWeightMet()`:

```
uint256 afterTradeWeight = FixedPointMathLib.fullMulDiv
  (assetValueInUSD, _WEIGHT_PRECISION, totalValues[i]);
if (MathUtils.diff
  (proposedTargetWeights[j], afterTradeWeight) > _MAX_WEIGHT_DEVIATION) {
         return false;
}
```

The code functions correctly assuming that the USD value of a basket stays stationary during the `_processInternalTrades()` during the call. However, that is not actually the case due to the fact that there is a fee upon processing the internal trades:

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

This results in the USD value of both the `fromBasket` and the `toBasket` going down based on the fee amount, thus the deviation weight check will be inaccurate as the `totalValues` array is not changed during the internal trades processing, it is out-of-sync.

Recommendations

Provide the total values array upon processing the internal trades and account for the fees applied.

[H-02] The target weight check does not account for redeem request assets

Severity

Impact: High

Likelihood: Medium

Description

In the rebalance process, target balances depend not only on the configured target weights but also on the withdraw requests processed within the rebalance epoch. However, when the `_isTargetWeightMet` check is performed within `proposeTokenSwap` and `completeRebalance`, it only considers the configured target weights and does not account for the processed withdraw requests.

```
function _isTargetWeightMet(
        BasketManagerStorage storage self,
        address[] calldata baskets,
        uint64[][] calldata basketsTargetWeights,
        address[][] calldata basketAssets,
        uint256[][] memory basketBalances,
        uint256[] memory totalValues
    )
        private
        view
        returns (bool)
    {
        // _isTargetWeightMet
        //(self, baskets, basketTargetWeights, basketAssets, basketBalances, totalValu
        // Check if total weight change due to all trades is within the
        // _MAX_WEIGHT_DEVIATION threshold
        uint256 len = baskets.length;
        for (uint256 i = 0; i < len;) {
            // slither-disable-next-line calls-loop
            uint64[] calldata proposedTargetWeights = basketsTargetWeights[i];
            // nosemgrep:
            // solidity.performance.state-variable-read-in-a-loop.state-variable-read-
            address[] calldata assets = basketAssets[i];
            // nosemgrep:
            // solidity.performance.array-length-outside-loop.array-length-outside-loo
            uint256 proposedTargetWeightsLength = proposedTargetWeights.length;
            for (uint256 j = 0; j < proposedTargetWeightsLength;) {
                address asset = assets[j];
                uint256 assetValueInUSD =
                // nosemgrep:
                // solidity.performance.state-variable-read-in-a-loop.state-variable-r
                 self.eulerRouter.getQuote
                   (basketBalances[i][j], asset, _USD_ISO_4217_CODE);
                // Rounding direction: down
                uint256 afterTradeWeight =
                    FixedPointMathLib.fullMulDiv
                      (assetValueInUSD, _WEIGHT_PRECISION, totalValues[i]);
                if (MathUtils.diff(
                  MathUtils.diff

                ) > _MAX_WEIGHT_DEVIATION
                    return false;
                }
                unchecked {
                    // Overflow not possible: j is bounded by
                    // proposedTargetWeightsLength
                    ++j;
                }
            }
            unchecked {
                // Overflow not possible: i is bounded by len
                ++i;
            }
        }
        return true;
    }
```

This means that if the total withdrawal value of the base asset causes the base asset weight to deviate by more than `_MAX_WEIGHT_DEVIATION` (5%), the rebalance process will revert, and the current withdrawal request cannot be processed.

Recommendations

Consider the total redeem request in the `_isTargetWeightMet` check.

8.2. Medium Findings

[M-01] Missing ERC20Permit init call in `initialize()`

Severity

Impact: Medium

Likelihood: High

Description

In the Solidity smart contract BasketToken, which inherits from ERC20PermitUpgradeable, there seems to be an omission in the initialization process. The initialize function, intended to set up the contract's initial state, does not include a call to `__ERC20Permit_init()`. This function is critical for initializing the ERC20 Permit feature, which is part of the BasketToken contract. The ERC20 Permit feature allows for gasless transactions by enabling users to sign approvals for token transfers with their private keys. Not calling `__ERC20Permit_init()` means that this functionality will not be properly set up in the BasketToken contract.

link

Recommendations

Modify the initialize function to include a call to `__ERC20Permit_init()`. This function should be called with appropriate arguments (usually the name of the token) to properly initialize the ERC20 Permit feature.

[M-02] Loss of surplus if ERC-1271 order allows arbitrary data

Severity

Impact: High

Likelihood: Low

Description

CoWSwapClone::isValidSignature() do not check app data field.

This applies to all ERC-1271 orders where the app data field can be changed by an adversary in a way that keeps the signature valid for that order (for example, because isValidSignature ignores the appData field in the order).

An adversary can manipulate vulnerable ERC-1271 orders, thereby transferring part of the expected surplus from the user order to an address that the adversary controls.

More details can be seen here.

Recommendations

making the app data immutable at deployment time (or equal to bytes(0)), and have isValidSignature reject an order if the app data doesn't match.

[M-03] Deviation check upon validating external trades is unidirectional

Severity

Impact: Medium

Likelihood: Medium

Description

Upon validating external trades, we have the following piece of code:

```
info.sellValue = self.eulerRouter.getQuote
  (trade.sellAmount, trade.sellToken, _USD_ISO_4217_CODE);
info.internalMinAmount = self.eulerRouter.getQuote
  (info.sellValue, _USD_ISO_4217_CODE, trade.buyToken);
info.diff = MathUtils.diff(info.internalMinAmount, trade.minAmount);

// Check if the given minAmount is within the _MAX_SLIPPAGE threshold of
// internalMinAmount
if (info.internalMinAmount < trade.minAmount) {
         if (
           (info.diff * _WEIGHT_PRECISION) / info.internalMinAmount > _MAX_SLIPPAGE) {
               revert ExternalTradeSlippage();
         }
}
```

It checks whether the provided `trade.minAmount` is within the `MAX_SLIPPAGE` threshold of the internally computed minimum amount. However, as seen, it only does it whenever the internal minimum amount is lower than the provided one. If this is the intended design, then it is still better to check the deviation in the other scenario as well as the following scenario can occur:

1. The prices of the traded assets are 1:1, both are worth 100$
2. A swap proposer proposes to trade assets and sets a minimum amount of 0.98 assets to receive while selling 1 of the other asset
3. While the transaction is still in the mempool, the price of the asset to buy goes to 90$ while the asset to sell stays at 100$
4. The internal minimum amount would be ~1.11 assets as you are providing 100$ of one asset to buy an asset worth 90$
5. As it is higher than the provided minimum amount of 0.98, we will skip the check
6. The trade can then be executed at a much worse rate than the actual one and in the worst-case scenario, we could receive 0.98 assets which is less than 90$ while we gave 100$ worth of assets

Recommendations

Execute the deviation check on both sides.

[M-04] `orderValidTo` assumes `stepDelay` is always 15 minutes

Severity

Impact: High

Likelihood: Low

Description

When the rebalance process is performed and incorporates `externalTrades`, it will set `orderValidTo` to 15 minutes since `executeTokenSwap` is called by the token swap executor. This will set a deadline for the swap process to not exceed a certain time window.

```
function executeTokenSwap(
      ExternalTrade[]calldataexternalTrades,
      bytescalldata
    ) external payable override {
>>>     uint32 validTo = uint32(block.timestamp + 15 minutes);
        _cowswapAdapterStorage().orderValidTo = validTo;
        for (uint256 i = 0; i < externalTrades.length;) {
            _createOrder(
                externalTrades[i].sellToken,
                externalTrades[i].buyToken,
                externalTrades[i].sellAmount,
                externalTrades[i].minAmount,
                validTo
            );
            unchecked {
                // Overflow not possible: i is bounded by externalTrades.length
                ++i;
            }
        }
    }
```

The `completeRebalance` function also checks `stepDelay`, which is initially configured to 15 minutes. This gives the swap request some time to be executed before `completeRebalance` can be called.

```
function completeRebalance(
        BasketManagerStorage storage self,
        ExternalTrade[] calldata externalTrades,
        address[] calldata baskets,
        uint64[][] calldata basketTargetWeights,
        address[][] calldata basketAssets
    )
        external
    {
        // Revert if there is no rebalance in progress
        // slither-disable-next-line incorrect-equality
        if (self.rebalanceStatus.status == Status.NOT_STARTED) {
            revert NoRebalanceInProgress();
        }
        _validateBasketHash(self, baskets, basketTargetWeights, basketAssets);
        // Check if the rebalance was proposed more than 15 minutes ago
        // slither-disable-next-line timestamp
>>>     if (block.timestamp - self.rebalanceStatus.timestamp < self.stepDelay) {
            revert TooEarlyToCompleteRebalance();
        }
        // ...
    }
```

However, if `stepDelay` is changed, the `orderValidTo` is still set to 15 minutes. This will cause an issue. For instance, if `stepDelay` is configured to be greater than 15 minutes to allow more time for swap execution, the swap process will still have only 15 minutes for swap execution.

Recommendations

Consider matching the `orderValidTo` time window with the `stepDelay`.

[M-05] `updateBitFlag` does not update the index of the base asset

Severity

Impact: High

Likelihood: Low

Description

The `updateBitFlag` function allows the basket's asset list to be updated, with the restriction that the new `bitFlag` must be a superset of the previous `bitFlag`. During the `updateBitFlag` process, if all validations pass, the function iterates over the new asset list and updates `basketAssetToIndexPlusOne`. However, it does not update `basketTokenToBaseAssetIndexPlusOne`.

Updating a `bitFlag` that is a superset of the previous `bitFlag` can still potentially impact `basketTokenToBaseAssetIndexPlusOne`. Consider the following scenario:

1. The current bitFlag is `1110`, meaning the asset list contains three assets. In this scenario, the first asset in the returned array is set as `basketTokenToBaseAssetIndexPlusOne`, resulting in a value of 1 (0 + 1).
2. The new bitFlag is `1111`, which is a superset of the previous `bitFlag`. The asset list now contains four assets. However, since `basketTokenToBaseAssetIndexPlusOne` has not been updated, the base asset index will point to the newly included asset. This causes all functions relying on this index to process the wrong asset.

Recommendations

Consider updating `basketTokenToBaseAssetIndexPlusOne` when `updateBitFlag` is called.

8.3. Low Findings

[L-01] `ManagementFee` calculation does not match the documentation

The documentation states that the calculation should divide by `_MANAGEMENT_FEE_DECIMALS`, whereas the actual code divides by `(_MANAGEMENT_FEE_DECIMALS - feeBps)`.

```
function _harvestManagementFee(uint16 feeBps, address feeCollector) internal {
        // Checks
        if (feeBps > _MAX_MANAGEMENT_FEE) {
            revert InvalidManagementFee();
        }

                 uint256 timeSinceLastHarvest = block.timestamp - lastManagementFeeHar

        // 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 - feeBps) * uint256
  (365 days))
                    );
                    if (fee != 0) {
                        emit ManagementFeeHarvested(fee);
                        _mint(feeCollector, fee);
                        // Interactions
                        FeeCollector(feeCollector).notifyHarvestFee(fee);
                    }
                }
            }
        }
    }
```

[L-02] `trade.maxAmount` check in the `_processInternalTrades()`

link `trade.maxAmount` appears to be designed to protect the counterparty (`toBasket`).

- `fromBasket` ultimately receives no less than `trade.minAmount` of `buyToken`.
- `toBasket` pays no more than `trade.maxAmount` of `buyToken`.

Therefore, it should be compared with `initialBuyAmount` rather than `info.netBuyAmount`.

[L-03] `proRataRedeem()` lacks a check to verify if `msg.sender` is the operator

```
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);
        }

        // Interactions
        BasketManager(basketManager).proRataRedeem(totalSupply(), shares, to);

        // We intentionally defer the `_burn
        //()` operation until after the external call to
        // `BasketManager.proRataRedeem
        //()` to prevent potential price manipulation via read-only reentrancy attacks
        // performing the external interaction before updating balances, we
        // ensure that total supply and user balances

        // ERC777 tokens or plugins with callbacks).
        _burn(from, shares);
    }
```

The operator has the authority to manage all funds. If msg.sender is the operator, they can perform the redeem operation without requiring any allowance. Recommendation: if (msg.sender != from) { if (!isOperator[from][msg.sender]) { _spendAllowance(from, msg.sender, shares); } }

[L-04] Fully claiming fees requires a batch of 2 claims

Upon claiming sponsor or treasury fees, we call `BasketToken.proRataRedeem()`:

```
function proRataRedeem(uint256 shares, address to, address from) public {
        ...
        _harvestManagementFee(feeBps, feeCollector);
        ...

       BasketManager(basketManager).proRataRedeem(totalSupply(), shares, to);
       ...
}
```

There, we first harvest fees once again which again increments the treasury and sponsor fees based on the time passed since the last harvest and notify the fee collector about the harvest:

```
FeeCollector(feeCollector).notifyHarvestFee(fee);
```

This results in the following scenario to occur:

1. Treasury calls `FeeCollector.claimTreasuryFees()` to get their fees, which are held in the `claimableTreasuryFees` mapping for the according basket token, they have 100 shares to claim
2. The mapping is cleared and `BasketToken.proRataRedeem()` is called which harvests 50 fees, bringing up `claimableTreasuryFees` after the notification to 50 shares
3. The 100 shares are redeemed
4. There are still 50 shares to claim

To fully claim the fees, a batch of 2 calls is required. Instead, consider harvesting the fees before the fees to redeem are set so a single call is sufficient.

[L-05] Hardcoded slippage is dangerous

Upon validating external trades, we have the following code:

```
if ((info.diff * _WEIGHT_PRECISION) / info.internalMinAmount > _MAX_SLIPPAGE) {
     revert ExternalTradeSlippage();
}
```

We validate the deviation between amounts. The issue is the hardcoded and constant `_MAX_SLIPPAGE`. In times of quick and drastic price changes, the hardcoded slippage can lead to the inability to call the function which will then lead to the inability to rebalance the pools, resulting in stuck redeems. There was a recent case where a hardcoded slippage check disallowed calling an important function of a contract. Instead, add a setter for the slippage to be able to change it in such a scenario.

[L-06] Lack of restriction on `BasketManager`'s `execute`

Currently, the `execute` function can call arbitrary addresses and perform arbitrary operations, including transferring assets held within `BasketManager` and registered in the `assetRegistry`, which is a restricted operation under `rescue`. Consider restricting the `target` as well. If the `target` is an asset registered in the `assetRegistry`, the operation should be reverted.

[L-07] `executeTokenSwap` can be called with empty `externalTrades`

It is possible that during the rebalancing process, `externalTrades` is empty, and the external swap should not be performed. However, `executeTokenSwap` can still be called with empty `externalTrades`. While this has no serious impact, as it will only update the `validTo`, `timestamp`, and status to `TOKEN_SWAP_EXECUTED`, consider reverting `executeTokenSwap` when `externalTrades` length is 0.

[L-08] External trades `tradeOwnership` is not validated

Due to the lack of validation for `tradeOwnership`, it is possible for the total `tradeOwnership` in the external trade to be either lower or greater than `_WEIGHT_PRECISION`, which could break the asset balance calculation within the baskets. Consider verifying that the total `tradeOwnershi`p in each external trade is equal to `_WEIGHT_PRECISION`.

[L-09] The rebalance requirement check is too sensitive

Currently, as long as the difference between the current `balances` and the `targetBalances` of assets is not 0, `shouldRebalance` will be set to true.

```
function _isRebalanceRequired(
        address[] memory assets,
        uint256[] memory balances,
        uint256[] memory targetBalances
    )
        private
        pure
        returns (bool shouldRebalance)
    {
        uint256 assetsLength = assets.length;
        for (uint256 j = 0; j < assetsLength;) {
            // slither-disable-start calls-loop
            if (
>>>             MathUtils.diff(balances[j], targetBalances[j]) > 0 // nosemgrep
            ) {
                shouldRebalance = true;
                break;
            }
            // slither-disable-end calls-loop
            unchecked {
                // Overflow not possible: j is less than assetsLength
                ++j;
            }
        }
    }
```

This will make the rebalance decision too sensitive to balance changes. Consider adding a reasonable threshold instead of using 0.

[L-10] `_harvestManagementFee` does not consider claimable fallback shares

When calculating `currentTotalSupply` within `_harvestManagementFee`, the current balance and pending redeem requests of the `feeCollector` are excluded. However, the calculation does not account for the claimable fallback shares of the `feeCollector`, causing `currentTotalSupply` to potentially still include `feeCollector`'s shares. Consider including the claimable fallback shares in the `currentTotalSupply` calculation.

```
function _harvestManagementFee
      (uint16 feeBps, address feeCollector) internal {
        // Checks
        if (feeBps > _MAX_MANAGEMENT_FEE) {
            revert InvalidManagementFee();
        }

                 uint256 timeSinceLastHarvest = block.timestamp - lastManagementFeeHar

        // 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);
+                        - pendingRedeemRequest
+ (lastRedeemRequestId[feeCollector], feeCollector)
+                        - claimableFallbackShares(feeCollector);
                    uint256 fee = FixedPointMathLib.fullMulDiv(
                        currentTotalSupply,
                        feeBps * timeSinceLastHarvest,
                        ((_MANAGEMENT_FEE_DECIMALS - feeBps) * uint256
                          (365 days))
                    );
                    if (fee != 0) {
                        emit ManagementFeeHarvested(fee);
                        _mint(feeCollector, fee);
                        // Interactions
                        FeeCollector(feeCollector).notifyHarvestFee(fee);
                    }
                }
            }
        }
    }
```

[L-11] Malicious users can decrease management fees accrual

To accrue management fees, we have the following code:

```
uint256 currentTotalSupply = totalSupply() - balanceOf(

) - balanceOf
  (feeCollector
uint256 fee = FixedPointMathLib.fullMulDiv(
  currentTotalSupply,
  feeBps*timeSinceLastHarvest,

) * uint256(365 days
...
_mint(feeCollector, fee);
FeeCollector(feeCollector).notifyHarvestFee(fee);
```

As we are determining the total supply by decreasing it by the fee collector share balance, a malicious user can directly send tokens to the fee collector in order to artificially decrease the total supply. While this is a loss for the malicious user, this is especially problematic because of the `notifyHarvestFee()` functionality which makes the balance to claim into an internal accounting value. This results in the shares directly sent to the fee collector to be not claimable and results in a grief attack where both the attacker and the fee receiver lose funds.

The absolute same thing goes for the `pendingRedeemRequest` as users can directly schedule a redeem causing a lower `currentTotalSupply` value. The redeem can then never be claimed as the `FeeCollector` contract does not have a function to do that, nor does it have a function to set an operator.

Consider adding a rescue type function in the fee collector contract to make it impossible for an attacker to cause damage, also add a function to claim redeems/add an operator **or** remove the subtraction for the pending redeem request as realistically, it is unlikely that the fee collector would have any redeems to claim unless a malicious user scheduled one.

[L-12] Management fees might not accrue

Management fees accrue using the following formula:

```
uint256 fee = FixedPointMathLib.fullMulDiv(
  currentTotalSupply,
  feeBps*timeSinceLastHarvest,

) * uint256(365 days
```

The formula can round down and there are 2 reasons which make that a likely event. Firstly, malicious users can call `BasketToken.proRataRedeem()` by redeeming 1 share often enough for the `timeSinceLastHarvest` to be very low. The second reason is a wrong variable used for the total supply initialization:

```
uint256 requiredDepositShares = basketValue > 0 ? FixedPointMathLib.fullMulDiv
  (pendingDepositValue, totalSupply, basketValue) : pendingDeposit;
```

Upon the first rebalance when the basket value is 0, we get the shares using `pendingDeposit`. The variable used should actually be `pendingDepositValue` as firstly, we can see that's the variable used when computing the shares when the basket value is over 0, and secondly, this causes the total supply to be a low value if the base asset is a low decimal asset. For example, if the base asset is `WBTC`, the total supply will be in 8 decimals which is very prone to rounding down in the fee calculation.

Firstly, use `pendingDepositValue` so the decimals of the supply will not be low if a low decimal base asset is used. Secondly, consider refactoring `_harvestManagementFee()` to not update the time since the last harvest if no fees were accrued.

[L-13] Calling `mint` or `withdraw` with 0 shares/assets resulting in a loss

If users currently have unfulfilled pending withdrawals or deposits, calling `mint` or `withdraw` with 0 shares or assets could result in a loss of assets.

```
function mint(
      uint256shares,
      addressreceiver,
      addresscontroller
    ) public returns (uint256 assets
        // Checks
        _onlySelfOrOperator(controller);

                 DepositRequestStruct storage depositRequest = _depositRequests[lastDe
        uint256 fulfilledShares = depositRequest.fulfilledShares;
        uint256 depositAssets = depositRequest.depositAssets[controller];
        // @audit - should also check, if shares is 0, revert
>>>     if (shares != _maxMint
  (fulfilledShares, depositAssets, depositRequest.totalDepositAssets)) {
            revert MustClaimFullAmount();
        }
        // Effects
        assets = _claimableDepositRequest(fulfilledShares, depositAssets);
        _claimDeposit(depositRequest, assets, shares, receiver, controller);
    }
```

```
function withdraw(
      uint256assets,
      addressreceiver,
      addresscontroller
    ) public override returns (uint256 shares
        // @audit - should check assets is not 0
        // Checks
        _onlySelfOrOperator(controller);

                 RedeemRequestStruct storage redeemRequest = _redeemRequests[lastRedee
        uint256 fulfilledAssets = redeemRequest.fulfilledAssets;
        uint256 redeemShares = redeemRequest.redeemShares[controller];
>>>     if (assets != _maxWithdraw
  (fulfilledAssets, redeemShares, redeemRequest.totalRedeemShares)) {
            revert MustClaimFullAmount();
        }
        shares = _claimableRedeemRequest(fulfilledAssets, redeemShares);
        // Effects
        _claimRedemption(redeemRequest, assets, shares, receiver, controller);
    }
```

Since `fulfilledShares` and `fulfilledAssets` are 0, `_maxWithdraw` and `_maxMint` will return 0, allowing the check to pass. As a result, the shares and assets will be set to 0 within the `_claimDeposit` and `_claimRedemption` operations.

```
function _claimRedemption(
        RedeemRequestStruct storage redeemRequest,
        uint256 assets,
        uint256 shares,
        address receiver,
        address controller
    )
        internal
    {
        // Effects
>>>     redeemRequest.redeemShares[controller] = 0;
        emit Withdraw(msg.sender, receiver, controller, assets, shares);
        // Interactions
        IERC20(asset()).safeTransfer(receiver, assets);
    }
```

```
function _claimDeposit(
        DepositRequestStruct storage depositRequest,
        uint256 assets,
        uint256 shares,
        address receiver,
        address controller
    )
        internal
    {
        // Effects
>>>     depositRequest.depositAssets[controller] = 0;
        emit Deposit(controller, receiver, assets, shares);
        // Interactions
        _transfer(address(this), receiver, shares);
    }
```

Consider reverting if `shares` and `assets` are 0.

[L-14] `AutomaticWeightStrategy` is not fully implemented

```
contract
   AutomaticWeightStrategy is WeightStrategy, AccessControlEnumerable, Multicall {
    constructor(address admin) payable {
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
    }

@>     function getTargetWeights
  (uint256 bitFlag) public view virtual override returns (uint64[] memory targetWeight

@>     function supportsBitFlag
  (uint256 bitFlag) public view virtual override returns (bool) { }
}
```

It can be observed that getTargetWeights() and supportsBitFlag() do not have implemented logic. This may be a known issue.

Since AutomaticWeightStrategy is within the scope of the audit, this issue should be submitted.

Fully implement the getTargetWeights() and supportsBitFlag() functions.

[L-15] Not transferring already collected `claimableSponsorFees`

The `setSponsor` function allows the admin to change the sponsor of a given `basketToken`. However, it does not transfer the collected `claimableSponsorFees` to the previous sponsor, allowing the new sponsor to claim the fees collected by the previous sponsor. Consider transferring the `claimableSponsorFees` to the previous sponsor if the amount is greater than 0 when `setSponsor` is called.

[L-16] Precision loss in `_processExternalTrades()`

```
function _processExternalTrades(
        BasketManagerStorage storage self,
        ExternalTrade[] calldata externalTrades
    )
        private
    {
        uint256 externalTradesLength = externalTrades.length;
        uint256[2][] memory claimedAmounts = _completeTokenSwap
          (self, externalTrades);
        // Update basketBalanceOf with amounts gained from swaps
        for (uint256 i = 0; i < externalTradesLength;) {
            ExternalTrade calldata trade = externalTrades[i];
            // nosemgrep:
            // solidity.performance.array-length-outside-loop.array-length-outside-loo
            uint256 tradeOwnershipLength = trade.basketTradeOwnership.length;
            for (uint256 j; j < tradeOwnershipLength;) {

                                 BasketTradeOwnership calldata ownership = trade.baske
                address basket = ownership.basket;
                // Account for bought tokens
                self.basketBalanceOf[basket][trade.buyToken] +=
@>                    FixedPointMathLib.fullMulDiv
  (claimedAmounts[i][1], ownership.tradeOwnership, _WEIGHT_PRECISION);
                // Account for sold tokens

                                 self.basketBalanceOf[basket][trade.sellToken] = self.
@>                   + FixedPointMathLib.fullMulDiv
  (claimedAmounts[i][0], ownership.tradeOwnership, _WEIGHT_PRECISION)
@>                    - FixedPointMathLib.fullMulDiv
  (trade.sellAmount, ownership.tradeOwnership, _WEIGHT_PRECISION);
                unchecked {
                    // Overflow not possible: i is less than
                    // tradeOwnerShipLength.length
                    ++j;
                }
            }
            unchecked {
                // Overflow not possible: i is less than
                // externalTradesLength.length
                ++i;
            }
        }
    }
```

Taking bought tokens as an example:

If tradeOwnershipLength is 2, claimedAmounts[i][1] is 111, basket0 has a tradeOwnership of 0.3e18, and basket1 has a tradeOwnership of 0.7e18, then using FixedPointMathLib.fullMulDiv() for calculation results in basket0 receiving 33 buyToken and basket1 receiving 77 buyToken. However, 33 + 77 = 110 < 111.

The same issue may also occur with sold tokens.