hexens x basisOS

Security Review Report for BasisOS

November 2024

Table of Contents

- Missing Asset/Product Check When Setting New Strategy
- Invalidation of setLimitDecreaseCollateral Validation Logic when setting new setCollateralMinMax
- Missing disableinitializers() to prevent uninitialized contracts
- BasisStrategy::_afterIncreasePosition may send asset to vault without LogarithmVault.processingPendingWithdrawRequests
- Use Custom Errors
- Use Ownable2StepUpgradeable for all contracts
- Constant variables should be marked as private

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit evaluates the Logarithm protocol, an on-chain public derivative crossing infrastructure designed to optimize leveraged trades and generate asymmetric yields. The Basis Strategy System underpins this framework, enabling delta-neutral basis trading by earning yield from funding payments. By pairing spot purchases with perpetual short positions, the system effectively hedges market exposure while capitalizing on funding revenue. Built with security, flexibility, and efficiency in mind, the protocol offers a yield-generating strategy that minimizes directional risk while leveraging perpetual funding markets.

Our security assessment involved a thorough review of eleven smart contracts over three weeks.

During the audit, we identified four high-severity vulnerabilities that could impact the protocol's rebalance process and potentially allow the theft of user funds. Additionally, we discovered four medium-severity vulnerabilities, seven low-risk issues, and three informational findings.

All reported issues were fixed by the development team and subsequently validated by us.

As a result, we can confidently state that the security and overall code quality of the protocol have improved following our audit.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 GitHub Repository

📌 **Commit:** `2c2717ce4d3f6f86b9b09d2c9a5ac492af67a491`

The issues described in this report were fixed in the following commit:

🔗 GitHub Repository

📌 **Commit:** `7e70ce197b943410655715b98fdde15d4072e38f`

▪ Changelog

25 November 2024	Audit Start
24 December 2024	Initial Report
14 January 2025	Revision Received
30 January 2025	Final Report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

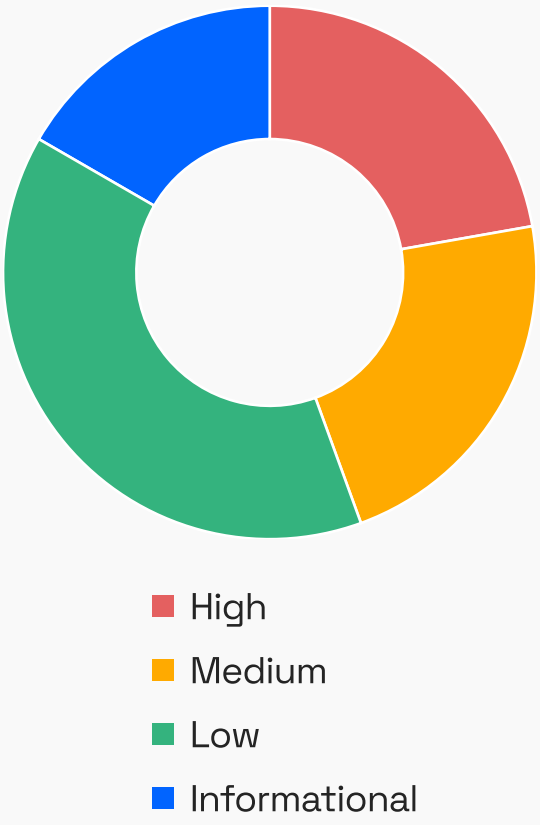
- **Issue Symbolic Codes**

Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of Findings
Critical	0
High	4
Medium	4
Low	7
Informational	3
Total:	**18**



6. Weaknesses

This section contains the list of discovered weaknesses.

LOGLAB-13 | The strategy does not pause when the deviation of sizeDeltaInTokens exceeds the threshold

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/strategy/BasisStrategy.sol#L975-L980

Description:

The function **BasisStrategy._afterDecreasePosition()** is called after the hedge position is decreased. This function returns a boolean value, **shouldPause**, to indicate whether the strategy should pause.

To determine the value of **shouldPause**, the function checks whether the deviations of **sizeDeltaInTokens** and **collateralDeltaAmount** between the response and request exceed a specific threshold. If either deviation exceeds the threshold, **shouldPause** should be set to **true**.

For the parameter **sizeDeltaInTokens**, **shouldPause** is correctly set to **true** in line 950 if **exceedsThreshold == true** and **sizeDeviation < 0**. However, for the parameter **collateralDeltaAmount**, **shouldPause** is incorrectly set to **exceedsThreshold** in line 979. This behavior is incorrect because the value of **shouldPause** gets overwritten, even if it was already set to **true** due to the **sizeDeltaInTokens** deviation check.

As a result, the strategy will not pause even when the **sizeDeltaInTokens** deviation exceeds the threshold.

```
    if (requestParams.collateralDeltaAmount > 0) {
        (bool exceedsThreshold,) = _checkDeviation(
            responseParams.collateralDeltaAmount,
requestParams.collateralDeltaAmount, _responseDeviationThreshold
        );
        shouldPause = exceedsThreshold;
    }
```

Remediation:

Consider modifying the update as follows:

```
shouldPause = shouldPause | exceedsThreshold;
```

LOGLAB-10 | Unable to redeem all user shares or sell all products and close the hedge when the decreaseSizeMax config is different from type(uint256).max

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/strategy/BasisStrategy.sol#L577-L584, src/strategy/BasisStrategy.sol#L743-L758

Description:

To process the withdrawal of assets for users, the operator needs to call the **BasisStrategy::deutilize()** function to sell the spot product and send a decrease collateral request to the agent via the **OffChainPositionManager::_adjustPosition()** function. After that, the agent will execute the off-chain order and call the **OffChainPositionManager::reportStateAndExecuteRequest()** function, sending the decreased collateral (assets) to the manager contract and triggering the **BasisStrategy::afterAdjustPosition()** function. Then, the **BasisStrategy::_afterDecreasePosition()** function will be triggered, which pulls funds from the manager contract, attempts to balance both the spot and hedge positions, and processes the withdrawal of the vault by transferring all existing assets.

However, there is a special case when the user attempts to withdraw all assets, or when the operator attempts to sell all products and close the hedge. In this case, the **spotSellCallback()** function will set the **sizeDeltaInTokens** and **collateralDeltaAmount** of the request to **type(uint256).max**.

```
function spotSellCallback(uint256 assetDelta, uint256 productDelta) external
authCaller(spotManager()) {
    [...]
    if (!processingRebalanceDown()) {
        if ($.vault.totalSupply() == 0 || ISpotManager(_msgSender()).exposure()
== 0) {
            // in case of redeeming all by users,
            // or selling out all product
            // close hedge position
            sizeDeltaInTokens = type(uint256).max;
            collateralDeltaAmount = type(uint256).max;
            $.pendingDecreaseCollateral = 0;
        }
```

_afterDecreasePosition function also handles this case by resetting the value of **requestParams** to **responseParams** (the actual delta amount executed by the agent):

```
function _afterDecreasePosition(IHedgeManager.AdjustPositionPayload calldata
responseParams)
    private
    returns (bool shouldPause)
{
    BasisStrategyStorage storage $ = _getBasisStrategyStorage();
    IHedgeManager.AdjustPositionPayload memory requestParams = $.requestParams;
    if (requestParams.sizeDeltaInTokens == type(uint256).max) {
        // when closing hedge
        requestParams.sizeDeltaInTokens = responseParams.sizeDeltaInTokens;
        requestParams.collateralDeltaAmount =
responseParams.collateralDeltaAmount;
    }
    [...]
}
```

The problem is that the **requestParams** variables are capped by the hedge manager's min-max config in the **BasisStrategy::_adjustPosition()** function.

```
function _adjustPosition(uint256 sizeDeltaInTokens, uint256
collateralDeltaAmount, bool isIncrease)
    [...]
    if (sizeDeltaInTokens > 0) {
        uint256 min;
        uint256 max;
        if (isIncrease) (min, max) = $.hedgeManager.increaseSizeMinMax();
        else (min, max) = $.hedgeManager.decreaseSizeMinMax();

        sizeDeltaInTokens = _clamp(min, sizeDeltaInTokens, max);
    }
```

Therefore, if the **decreaseSizeMax** config of the hedge manager is set to something other than **type(uint256).max**, it never enters the branch in the **_afterDecreasePosition** function. This means that **requestParams** will not be reset to **responseParams** in the case of closing the hedge or redeeming all assets.

In this case, the **_afterDecreasePosition** function always calculates a large deviation between **responseParams.sizeDeltaInTokens** (the actual decreased size by the agent from off-chain) and **requestParams.sizeDeltaInTokens** (which is capped by the decreaseSizeMax config).

As a result, the deviation exceeds the threshold significantly, leading to a large amount of assets being returned to the spot manager and the product being bought again.

```
function _afterDecreasePosition(IHedgeManager.AdjustPositionPayload calldata
responseParams)
    [...]
    if (requestParams.sizeDeltaInTokens > 0) {
        uint256 _pendingDeutilizedAssets = $.pendingDeutilizedAssets;
        delete $.pendingDeutilizedAssets;
        (bool exceedsThreshold, int256 sizeDeviation) = _checkDeviation(
            responseParams.sizeDeltaInTokens, requestParams.sizeDeltaInTokens,
_responseDeviationThreshold
        );
        if (exceedsThreshold) {
            shouldPause = true;
            if (sizeDeviation < 0) {
                uint256 sizeDeviationAbs = uint256(-sizeDeviation);
                uint256 assetsToBeReverted;
                if (sizeDeviationAbs == requestParams.sizeDeltaInTokens) {
                    assetsToBeReverted = _pendingDeutilizedAssets;
                } else {
                    assetsToBeReverted =
                        _pendingDeutilizedAssets.mulDiv(sizeDeviationAbs,
requestParams.sizeDeltaInTokens);
                }
                if (assetsToBeReverted > 0) {
                    ISpotManager _spotManager = $.spotManager;
                    _asset.safeTransfer(address(_spotManager),
assetsToBeReverted);
                    _spotManager.buy(assetsToBeReverted,
ISpotManager.SwapType.MANUAL, "");
                }
            }
        }
```

As a consequence, only a few assets from the spot will be sent to the vault, while the hedge position reduces significantly in size and collateral, leading to an imbalanced and unsafe situation between the spot and hedge positions. As a result, it becomes impossible to sell all of the spot products to process the withdrawal of assets for redeeming all shares.

Remediation:

The **BasisStrategy::_adjustPosition** function shouldn't cap the **sizeDeltaInTokens** value if it is equal to **type(uint256).max**. The condition should be updated as follows:

```
if (sizeDeltaInTokens > 0 && sizeDeltaInTokens != type(uint256).max) {
```

Proof of concept:

- In testing, **decreaseSizeMax** and **decreaseCollateralMax** configs are set to **type(uint256).max** in file **test/base/OffChainTest.sol**
- When running the **test_deutilize_lastRedeemBelowRequestedAssets** test function in **test/unit/BasisStrategyOffchain.t.sol**, the vault receives 9749999999 asset tokens, and the redemption is successful
- Change both the **decreaseSizeMax** and **decreaseCollateralMax** configs to **1e24** and run the above test again. The vault only receives 2250000000 asset tokens, and the test reverts because **vault.isClaimable(requestKey)** returns false (not enough assets processed to withdraw)
- Change the **decreaseSizeMax** config to **1e24** and keep **decreaseCollateralMax** as **type(uint256).max**, then run the above test again. The test reverts at the **reportStateAndExecuteRequest()** call after executing the off-chain order due to overflow behavior during the deviation calculation with a very large **requestParams.collateralDeltaAmount** (the maximum of uint256)

LOGLAB-17 | pendingDecreaseCollateral variable isn't excluded from the positionNetBalance() value in the leverage and rebalance calculations, which may lead to incorrect rebalance actions for the strategy

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/strategy/BasisStrategy.sol#L779-L884

Description:

During a partial deutilizing action, the **BasisStrategy::spotSellCallback()** function will assign the **collateralDeltaToDecrease** value to the **$.pendingDecreaseCollateral** storage variable when **collateralDeltaToDecrease** is lower than the **limitDecreaseCollateral** value from the hedge manager.

```
collateralDeltaToDecrease += _pendingDecreaseCollateral;
uint256 limitDecreaseCollateral = _hedgeManager.limitDecreaseCollateral();
if (collateralDeltaToDecrease < limitDecreaseCollateral) {
    $.pendingDecreaseCollateral = collateralDeltaToDecrease;
} else {
    collateralDeltaAmount = collateralDeltaToDecrease;
}
```

After that, the **collateralDeltaAmount** requested for the deutilization will be 0, as **pendingDecreaseCollateral** will be stacked and considered in the next deutilization.

An important point is that **pendingDecreaseCollateral** shouldn't be counted toward the net balance of the hedge position because that collateral amount should have already been decreased from the hedge position during the earlier size reduction.
Therefore, in the calculation of **collateralDeltaToDecrease** in the **spotSellCallback()** and **_pendingDeutilization()** functions, the **pendingDecreaseCollateral** variable is subtracted from the net balance of the hedge manager.

```
function spotSellCallback(uint256 assetDelta, uint256 productDelta) external
authCaller(spotManager()) {
    [...]
    // when partial deutilizing
    IHedgeManager _hedgeManager = $.hedgeManager;
    uint256 positionNetBalance = _hedgeManager.positionNetBalance();
    uint256 _pendingDecreaseCollateral = $.pendingDecreaseCollateral;
    if (_pendingDecreaseCollateral > 0) {
        (, positionNetBalance) =
positionNetBalance.trySub(_pendingDecreaseCollateral);
    }
    uint256 positionSizeInTokens = _hedgeManager.positionSizeInTokens();
    uint256 collateralDeltaToDecrease =
        positionNetBalance.mulDiv(productDelta, positionSizeInTokens);
```

```
function _pendingDeutilization(InternalPendingDeutilization memory params)
private view returns (uint256) {
    [...]
    deutilization = positionSizeInTokens.mulDiv(
        totalPendingWithdraw - _pendingDecreaseCollateral,
        positionSizeInAssets + positionNetBalance - _pendingDecreaseCollateral
    );
```

However, in the **_checkUpkeep()** function, the leverage is calculated using the **OffChainPositionManager::currentLeverage()** function, which relies on the original **positionNetBalance** and does not exclude the **pendingDecreaseCollateral** variable. Additionally, the delta collateral amounts for rebalance up and rebalance down are still calculated based on the original **positionNetBalance**.

```
function _checkUpkeep() private view returns (InternalCheckUpkeepResult memory
result) {
    [...]
    uint256 currentLeverage = _hedgeManager.currentLeverage();
    bool _processingRebalanceDown = $.processingRebalanceDown;
    uint256 _maxLeverage = $.maxLeverage;
    uint256 _targetLeverage = $.targetLeverage;

    (bool rebalanceUpNeeded, bool rebalanceDownNeeded, bool deleverageNeeded) =
        _checkRebalance(currentLeverage, $.minLeverage, _maxLeverage,
$.safeMarginLeverage);

    [...]
```

```
    if (rebalanceDownNeeded) {
        uint256 idleAssets = _vault.idleAssets();
        (uint256 minIncreaseCollateral,) =
_hedgeManager.increaseCollateralMinMax();
        result.deltaCollateralToIncrease =
_calculateDeltaCollateralForRebalance(
            _hedgeManager.positionNetBalance(), currentLeverage,
_targetLeverage
        );

    [...]

    if (rebalanceUpNeeded) {
      result.deltaCollateralToDecrease = _calculateDeltaCollateralForRebalance(
          _hedgeManager.positionNetBalance(), currentLeverage, _targetLeverage
      );

    [...]
```

The impact of not excluding **pendingDecreaseCollateral** from **positionNetBalance** in these actions is that incorrect leverage and rebalance amount will be considered, potentially leading to improper rebalancing for the strategy. Furthermore, when **pendingDecreaseCollateral** is executed after a larger request in subsequent deutilizations, it causes the leverage to deviate in the opposite direction, resulting in an imbalanced state for the strategy.

Scenario:

1. In the first deutilization, if **collateralDeltaToDecrease** is lower than the limit, it will be stacked to **pendingDecreaseCollateral**, and no collateral will be requested to decrease from the agent. However, the size of the hedge position is still requested to decrease.
2. The agent reduces the size of the hedge position corresponding to the first deutilization off-chain and calls **reportStateAndExecuteRequest()** with 0 collateral to decrease. This still succeeds because the previous request was 0.
3. Now, in the **_checkUpkeep()** function, the leverage will decrease because the size of the hedge position was reduced, but it still considers the original **positionNetBalance** without excluding **pendingDecreaseCollateral**. Since **pendingDecreaseCollateral** should have been removed from the hedge position, this leverage may lead to an incorrect state, where **rebalanceUpNeeded** is true, causing the strategy to rebalance incorrectly.
4. The operator executes the second deutilization, and **collateralDeltaToDecrease** (which already includes the pendingDecreaseCollateral) is now larger than the limit, so it requests the total collateralDeltaAmount from both deutilizations.

5. The agent reduces the size of the hedge position corresponding to the second deutilization off-chain and calls **reportStateAndExecuteRequest()** with the total collateral to decrease from both deutilizations.
6. Now, in the **_checkUpkeep()** function, the calculated leverage will increase because the size of the hedge position was reduced only by the spot size of the second deutilization, but it removed the total **collateralDeltaToDecrease** from both deutilizations from the **positionNetBalance**. This leverage may lead to an incorrect state, where **rebalanceDownNeeded** is true, causing the strategy to rebalance incorrectly.

Remediation:

Consider always excluding the **pendingDecreaseCollateral** of the strategy in the calculation of **positionNetBalance** and **currentLeverage** of the hedge manager.So, it should be handled in the functions of **OffChainPositionManager** instead of being excluded in the strategy contract within the **spotSellCallback()** and **_pendingDeutilization()** functions. Additionally, note that in the **_afterDecreaseCollateral** function, the calculation of rebalance down **processingRebalanceDown** should be placed after transferring collateral and updating **$.pendingDecreaseCollateral**.

LOGLAB-15 | Last withdrawer could give next depositor 0 shares

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

src/vault/LogarithmVault.sol#L510-L513

Description:

The **claim** function has a different workflow for the last redeem, this requires **totalSupply** = 0 and **utilizedAssets** = 0 (meaning the last withdraw was executed).

To be able to claim, a user must first make a withdrawal request. This causes their tokens to be burned, which can make the total supply go to 0 if they are the last person to withdraw.

This is important because it triggers the **isLast** case in the **claim** function. This allows an attacker to receive all idle assets from the vault, since they would be the last person to withdraw.

However before claiming, an attacker can do a token transfer of for example **TEN_THOUSANDS_USDC** to the **vault** and **deposit** 1 wei, to receive 1 **share** since the **totalSupply** is 0. This would give the next depositor 0 shares when they deposit an amount under that.

The attacker can make another withdrawal request for this 1 wei, to make the **totalSupply** 0.

Now, the attacker can claim the first withdraw request (because total supply is 0 again) and receive all the remaining assets in the vault, including the tokens that were transferred. This allows the attacker to take all of the second depositor's assets, while the second depositor loses all funds.

1. User has 10,000 USDC of assets and the corresponding shares in the vault
2. The user creates a withdrawal request, **withdrawRequestID = 1**
3. The user transfers 10,000 USDC to the vault
4. The user deposits 1 wei and receives **1 share** in return
5. A victim deposits 5,000 USDC into the vault but receives **0 shares**
6. The user initiates a second withdrawal request, creating **withdrawRequestID = 2**
7. The user then claims the withdrawal with **withdrawRequestID = 1** and receives a profit of 5,000 USDC

```
  function claim(bytes32 withdrawRequestKey) public virtual returns (uint256) {
    -- Snip --
        if (isLast) {
            -- Snip --
            if (shortfall > 0) {
               -- Snip --
            } else {
                uint256 _idleAssets = idleAssets(); <= transfer transfered
funds back to user
                executedAssets = withdrawRequest.requestedAssets + _idleAssets;
                $.assetsToClaim += _idleAssets;
            }
        } else {
            executedAssets = withdrawRequest.requestedAssets;
        }

        $.assetsToClaim -= executedAssets;
        IERC20(asset()).safeTransfer(withdrawRequest.receiver, executedAssets);
        emit Claimed(withdrawRequest.receiver, withdrawRequestKey,
executedAssets);
        return executedAssets;
    }
```

Remediation:

Possible remediation for this issue:

- Revert Deposits with Zero Shares
- Slippage Protection: Allow users to set a minimum share amount they wish to receive on deposit

Proof of concept:

Add the following code to **BasisStrategyBaseTest.t.sol**, based on the last withdrawer flow in the test at **test/unit/BasisStrategyBase.t.sol** lines 593–621.

```
function test_poc() public {

    uint256 balBefore = IERC20(asset).balanceOf(user1);
    console.log("user1 original deposit.");
    _deposit(user1,TEN_THOUSANDS_USDC);

    (uint256 pendingUtilizationInAsset,) = strategy.pendingUtilizations();
    _utilize(pendingUtilizationInAsset);

    console.log("vault.idleAssets()",vault.idleAssets());

    vm.startPrank(user1);
    vault.requestRedeem(vault.balanceOf(user1), user1, user1);
    vm.stopPrank();

    (, uint256 pendingDeutilization) = strategy.pendingUtilizations();
    _deutilize(pendingDeutilization);

    bytes32 user1RequestKey = vault.getWithdrawKey(user1, 0);
    LogarithmVault.WithdrawRequest memory req =
vault.withdrawRequests(user1RequestKey);

    vm.startPrank(user1);
    IERC20(asset).transfer(address(vault), TEN_THOUSANDS_USDC);
    vm.stopPrank();

    console.log("user1 deposit after withdrawRequest");
    _deposit(user1,1);
    console.log("victim first deposit");
    _deposit(user2, TEN_THOUSANDS_USDC/2);

    vm.startPrank(user1);
    vault.requestRedeem(vault.balanceOf(user1), user1, user1);
    vm.stopPrank();
```

```
vm.startPrank(user1);
    vault.claim(user1RequestKey);
    uint256 balAfter = IERC20(asset).balanceOf(user1);
    console.log("balBefore", balBefore);
    console.log("balAfter", balAfter);
    console.log("profit: ", balAfter - balBefore );
}

// added logging to the original function
function _deposit(address from, uint256 assets) internal {
    vm.startPrank(from);
    IERC20(asset).approve(address(vault), assets);
    uint256 shares = vault.deposit(assets, from);
    console.log("Deposited assets: ", assets);
    console.log("Received shares: ", shares);
    vm.stopPrank();
}
```

Output:

```
Logs:
user1 original deposit.
Deposited assets:  10000000000
Received shares:  10000000000

vault.idleAssets() 0

user1 deposit after withdrawRequest
Deposited assets:  1
Received shares:  1

victim first deposit
Deposited assets:  5000000000
Received shares:  0

balBefore 10000000000000
balAfter 10004989484284
profit:  4989484284
```

LOGLAB-22 | Lack of slippage protection for manual swap in SpotManager

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

src/spot/SpotManager.sol#L145-L188, src/libraries/uniswap/ManualSwapLogic.sol#L16-L30

Description:

In the **SpotManager** contract, the **buy()** and **sell()** functions attempt to swap tokens between assets and products via the 1inch router or through manual swaps using Uniswap V3 Pools. However, these functions lack slippage protection for the swaps, such as a **minAmountOut** variable. The **INCH_V6** swap type is unaffected because the **minAmountOut** can be encoded into **swapData** for swaps via the 1inch router. However, the issue arises with the **MANUAL** swap type, which swaps directly through Uniswap V3 Pools without any slippage protection.

```
amountOut = ManualSwapLogic.swap(amount, $.productToAssetSwapPath);
```

This may put the assets of SpotManager at risk of slippage due to price fluctuations in Uniswap V3 Pools. It could harm the protocol if swaps are performed under unfavorable price or pool conditions.

```
function buy(uint256 amount, SwapType swapType, bytes calldata swapData)
external authCaller(strategy()) {
    uint256 amountOut;
    if (swapType == SwapType.INCH_V6) {
        bool success;
        (amountOut, success) = InchAggregatorV6Logic.executeSwap(amount,
asset(), product(), true, swapData);
        if (!success) {
            revert Errors.SwapFailed();
        }
    } else if (swapType == SwapType.MANUAL) {
        SpotManagerStorage storage $ = _getSpotManagerStorage();
        amountOut = ManualSwapLogic.swap(amount, $.assetToProductSwapPath);
    } else {
        // TODO: fallback swap
        revert Errors.UnsupportedSwapType();
    }
    emit SpotBuy(amount, amountOut);

    IBasisStrategy(_msgSender()).spotBuyCallback(amount, amountOut);
}

function sell(uint256 amount, SwapType swapType, bytes calldata swapData)
external authCaller(strategy()) {
    uint256 amountOut;
    if (swapType == SwapType.INCH_V6) {
        bool success;
        (amountOut, success) = InchAggregatorV6Logic.executeSwap(amount,
asset(), product(), false, swapData);
        if (!success) {
            revert Errors.SwapFailed();
        }
    } else if (swapType == SwapType.MANUAL) {
        SpotManagerStorage storage $ = _getSpotManagerStorage();
        amountOut = ManualSwapLogic.swap(amount, $.productToAssetSwapPath);
    } else {
        // TODO: fallback swap
        revert Errors.UnsupportedSwapType();
    }
    emit SpotSell(amountOut, amount);

    IBasisStrategy(_msgSender()).spotSellCallback(amountOut, amount);
}
```

```
function swap(uint256 amountIn, address[] memory path) external returns
(uint256 amountOut) {
    address tokenIn = path[0];
    uint256 balance = IERC20(tokenIn).balanceOf(address(this));
    if (balance < amountIn) {
        revert Errors.SwapAmountExceedsBalance(amountIn, balance);
    }

    for (uint256 i; i <= path.length / 2; i += 2) {
        address pool = path[i + 1];
        amountIn = exactInputInternal(
            amountIn, address(this), pool, path[i] < path[i + 2],
abi.encode(path[i], path[i + 2], address(this))
        );
    }
    amountOut = amountIn;
}

function exactInputInternal(uint256 amountIn, address recipient, address pool,
bool zeroForOne, bytes memory data)
    internal
    returns (uint256 amountOut)
{
    (int256 amount0, int256 amount1) = IUniswapV3Pool(pool).swap(
        recipient,
        zeroForOne,
        amountIn.toInt256(),
        zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.MAX_SQRT_RATIO - 1,
        data
    );
    amountOut = uint256(-(zeroForOne ? amount1 : amount0));
}
```

Remediation:

A **minAmountOut** variable should be added to the **buy()** and **sell()** functions to validate the amount received from the swap.

LOGLAB-18 | $.pendingDecreaseCollateral variable will be updated incorrectly if the agent executes an insufficient response, leading to an imbalance in the strategy after unpausing

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

src/strategy/BasisStrategy.sol#L946-L966, src/strategy/BasisStrategy.sol#L578-L590
src/strategy/BasisStrategy.sol#L590-L609

Description:

When the agent executes the deutilize request with an insufficient response, the **_afterDecreasePosition()** function will revert the request to the spot manager, returning the assets corresponding to the insufficient response size, and will repurchase the product in the spot manager. After that, it returns **shouldPause** as **true**, causing the strategy to be paused.

```
function _afterDecreasePosition(IHedgeManager.AdjustPositionPayload calldata responseParams)
    [...]
    (bool exceedsThreshold, int256 sizeDeviation) = _checkDeviation(
        responseParams.sizeDeltaInTokens, requestParams.sizeDeltaInTokens, _responseDeviationThreshold
    );
    if (exceedsThreshold) {
        shouldPause = true;
        if (sizeDeviation < 0) {
            uint256 sizeDeviationAbs = uint256(-sizeDeviation);
            uint256 assetsToBeReverted;
            if (sizeDeviationAbs == requestParams.sizeDeltaInTokens) {
                assetsToBeReverted = _pendingDeutilizedAssets;
            } else {
                assetsToBeReverted =
                    _pendingDeutilizedAssets.mulDiv(sizeDeviationAbs, requestParams.sizeDeltaInTokens);
            }
            if (assetsToBeReverted > 0) {
                ISpotManager _spotManager = $.spotManager;
```

```
_asset.safeTransfer(address(_spotManager), assetsToBeReverted);
                _spotManager.buy(assetsToBeReverted,
ISpotManager.SwapType.MANUAL, "");
            }
        }
    }
    [...]
    if (requestParams.collateralDeltaAmount > 0) {
        (bool exceedsThreshold,) = _checkDeviation(
            responseParams.collateralDeltaAmount,
requestParams.collateralDeltaAmount, _responseDeviationThreshold
        );
        shouldPause = exceedsThreshold;
    }
```

The issue arises because, during this deutilization, **requestParams.sizeDeltaInTokens** represents only the size of the current request, while **requestParams.collateralDeltaAmount** may also include the collateral delta from previous deutilizations stored in the **$.pendingDecreaseCollateral** variable. Since the previous deutilizations were executed correctly, **$.pendingDecreaseCollateral** is supposed to be decreased from the hedge position. However, the **$.pendingDecreaseCollateral** variable will be updated incorrectly if the response for the current deutilization is insufficient.

Consider 2 cases:

1. The current deutilization is fully deutilizing
   In this case, **spotSellCallback()** already updates **$.pendingDecreaseCollateral** to 0 without considering the result of the response

```
if ($.vault.totalSupply() == 0 || ISpotManager(_msgSender()).exposure() == 0) {
    // in case of redeeming all by users,
    // or selling out all product
    // close hedge position
    sizeDeltaInTokens = type(uint256).max;
    collateralDeltaAmount = type(uint256).max;
    $.pendingDecreaseCollateral = 0;
} else if (status == StrategyStatus.FULL_DEUTILIZING) {
    (uint256 min,) = $.hedgeManager.decreaseCollateralMinMax();
    uint256 pendingWithdraw = assetsToDeutilize();
    collateralDeltaAmount = min > pendingWithdraw ? min : pendingWithdraw;
    $.pendingDecreaseCollateral = 0;
}
```

So, if the response of this deutilization is insufficient, **$.pendingDecreaseCollateral** will be cleared, resulting in missing an collateral amount that should have been decreased from the hedge manager based on previous requests. Since the previous deutilizations successfully reduced the size of both the spot manager and hedge manager without adjusting the collateral, the leverage and state of the hedge manager will become imbalanced. Additionally, after unpausing the strategy, $.pendingDecreaseCollateral (which is now 0) will be in an incorrect state for the previous withdrawal request.

2. The current deutilization is partially deutilizing, and **collateralDeltaToDecrease** is larger than **limitDecreaseCollateral**.
   In this case, **spotSellCallback()** will add **$.pendingDecreaseCollateral** to the current **collateralDeltaAmount** for the request. However, **$.pendingDecreaseCollateral** remains unchanged without any updates.

```
else {
    // when partial deutilizing
    IHedgeManager _hedgeManager = $.hedgeManager;
    uint256 positionNetBalance = _hedgeManager.positionNetBalance();
    uint256 _pendingDecreaseCollateral = $.pendingDecreaseCollateral;
    if (_pendingDecreaseCollateral > 0) {
        (, positionNetBalance) =
positionNetBalance.trySub(_pendingDecreaseCollateral);
    }
    uint256 positionSizeInTokens = _hedgeManager.positionSizeInTokens();
    uint256 collateralDeltaToDecrease =
        positionNetBalance.mulDiv(productDelta, positionSizeInTokens);
    collateralDeltaToDecrease += _pendingDecreaseCollateral;
    uint256 limitDecreaseCollateral = _hedgeManager.limitDecreaseCollateral();
    if (collateralDeltaToDecrease < limitDecreaseCollateral) {
        $.pendingDecreaseCollateral = collateralDeltaToDecrease;
    } else {
        collateralDeltaAmount = collateralDeltaToDecrease;
    }
}
```

After that, **_afterDecreasePosition** updates **$.pendingDecreaseCollateral** by subtracting the collateral reduced based on the response.

```
if (responseParams.collateralDeltaAmount > 0) {
    // the case when deutilizing for withdrawals and rebalancing Up
    (, $.pendingDecreaseCollateral) =
$.pendingDecreaseCollateral.trySub(responseParams.collateralDeltaAmount);
    _asset.safeTransferFrom(_msgSender(), address(this),
responseParams.collateralDeltaAmount);
}
```

This behavior is incorrect when the response is insufficient because **responseParams.collateralDeltaAmount** is expected to include the collateral delta from both the current and previous deutilizations, while **$.pendingDecreaseCollateral** does not account for the collateral delta from the current deutilization. As a result, this update will either excessively reduce or incorrectly clear **$.pendingDecreaseCollateral**, leading to the same impact as in Case 1.

Remediation:

Update **$.pendingDecreaseCollateral** in the **_afterDecreasePosition** function instead of resetting it to 0 before making the request, as shown below:

```
if (responseParams.collateralDeltaAmount > 0) {
    // the case when deutilizing for withdrawals and rebalancing Up
    if (!shouldPause) {
        //The response decreased the sufficient size and collateral of the
position, including pendingDecreaseCollateral
        $.pendingDecreaseCollateral = 0;
    } else {
        //Calculate the corresponding collateral requested when the response is
insufficient
        uint256 currentRequestedCollateral =
(requestParams.collateralDeltaAmount - $.pendingDecreaseCollateral)
            .mulDiv(responseParams.sizeDeltaInTokens,
requestParams.sizeDeltaInTokens);

        //If the response lacks sufficient collateral for the current request,
add the deficit to pendingDecreaseCollateral
        //Otherwise, decrease pendingDecreaseCollateral by the remaining
collateral.
        if (responseParams.collateralDeltaAmount < currentRequestedCollateral)
{
            $.pendingDecreaseCollateral += currentRequestedCollateral -
responseParams.collateralDeltaAmount;
        } else {
            (, $.pendingDecreaseCollateral) =
$.pendingDecreaseCollateral.trySub(
                responseParams.collateralDeltaAmount -
currentRequestedCollateral
            );
        }
    }
    _asset.safeTransferFrom(_msgSender(), address(this),
responseParams.collateralDeltaAmount);
}
```

LOGLAB-21 | Not all pendingDecreaseCollateral is utilized due to the max limit

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

src/strategy/BasisStrategy.sol#L601-L607 , src/strategy/BasisStrategy.sol#L752-L758

Description:

In scenarios where the strategy is partially deutilizing, the storage variable **pendingDecreaseCollateral** is added to **collateralDeltaToDecrease** on line 601 of the **BasisStrategy.spotSellCallback()** function. This represents the amount of collateral to be reduced from the position.

Following this, the **_adjustPosition()** function is called to validate the position adjustment parameters before proceeding. Within **_adjustPosition()**, the **collateralDeltaAmount** is clamped — if its value exceeds a specific boundary **max**, it is capped at that **max**.

This behavior introduces an issue when **collateralDeltaAmount > max**, as not all of the **pendingDecreaseCollateral** is utilized. The remaining amount, calculated as **collateralDeltaAmount - max**, will not be removed from the position.

```
collateralDeltaToDecrease += _pendingDecreaseCollateral;
uint256 limitDecreaseCollateral = _hedgeManager.limitDecreaseCollateral();
if (collateralDeltaToDecrease < limitDecreaseCollateral) {
    $.pendingDecreaseCollateral = collateralDeltaToDecrease;
} else {
    collateralDeltaAmount = collateralDeltaToDecrease;
}
```

```
if (collateralDeltaAmount > 0) {
    uint256 min;
    uint256 max;
    if (isIncrease) (min, max) = $.hedgeManager.increaseCollateralMinMax();
    else (min, max) = $.hedgeManager.decreaseCollateralMinMax();
    collateralDeltaAmount = _clamp(min, collateralDeltaAmount, max);
}
```

Remediation:

When the **collateralDeltaAmount** is bigger than **max**, the **pendingDecreaseCollateral** should be assigned with **collateralDeltaAmount - max**.

LOGLAB-19 | Unable to execute the final withdrawal due to utilizedAssets() not being zero

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

src/vault/LogarithmVault.sol#L758-L761

Description:

The function **LogarithmVault._isWithdrawRequestExecuted()** determines whether a specific withdrawal request can be executed. It returns a boolean variable, **isExecuted**, to indicate if the user's request has been fulfilled. A user is allowed to execute the withdrawal only when this variable evaluates to **true**.

For the last withdrawal request, the value of **isExecuted** is determined by checking whether the **utilizedAssets()** value of the strategy contract is zero. If it is, the user is permitted to withdraw their tokens. However, an attacker can manipulate the state by front-running the execution and causing **utilizedAssets()** to remain non-zero, thereby preventing the user from completing their withdrawal.

To understand this, we examine the implementation of the function **BasisStrategy.utilizedAssets()**:

```
function utilizedAssets() public view returns (uint256) {
    BasisStrategyStorage storage $ = _getBasisStrategyStorage();
    return
        $.spotManager.getAssetValue() +
        $.hedgeManager.positionNetBalance() +
        assetsToWithdraw();
}

function assetsToWithdraw() public view returns (uint256) {
    return IERC20(asset()).balanceOf(address(this));
}
```

The **assetsToWithdraw()** function calculates its value based on the strategy contract's token balance. An attacker can exploit this by transferring 1 wei of the asset token directly to the strategy contract, making the **utilizedAssets()** return a value greater than zero. As a result, **isExecuted** will evaluate to **false**, blocking the final withdrawal request.

Another way to exploit this involves manipulating the **positionNetBalance()** value. An attacker can transfer some collateral token directly to the **OffChainPositionManager** contract, possibly causing **positionNetBalance()** to return a non-zero value. Here's the relevant implementation:

```
function positionNetBalance() public view virtual override returns (uint256) {
    OffChainPositionManagerStorage storage $ =
_getOffChainPositionManagerStorage();
    PositionState memory state = $.positionStates[$.currentRound];

    uint256 initialNetBalance = state.netBalance
                              + $.pendingCollateralIncrease
                              + idleCollateralAmount();
    ...
}

function idleCollateralAmount() public view returns (uint256) {
    return IERC20(collateralToken()).balanceOf(address(this));
}
```

In both cases, by transferring minimal tokens to the respective contracts, the attacker can artificially inflate the **utilizedAssets()** value, thereby preventing the execution of the last withdrawal request.

```
if (isLast) {
    // last withdraw is claimable when utilized assets is 0
    isExecuted = IStrategy(strategy()).utilizedAssets() == 0;
} else {
```

Remediation:

Consider clearing the idle assets within the strategy and hedge manager contract before checking if the **utilizedAssets()** is equal to zero.

LOGLAB-14 | Redundant and ineffective staleness check implementation

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/oracle/LogarithmOracle.sol#L110-115

Description:

The **LogarithmOracle** uses Chainlink data feeds to fetch assets prices. This oracle exposes the **latestRoundData** function to return pricing data.
There is a staleness check, the first part of which is redundant.

```
uint256 heartbeatDuration =
_getLogarithmOracleStorage().heartbeatDurations[address(priceFeed)];
if (block.timestamp > timestamp && block.timestamp - timestamp >
heartbeatDuration) {
    revert Errors.PriceFeedNotUpdated(asset, timestamp, heartbeatDuration);
}
```

The condition **block.timestamp - timestamp > heartbeatDuration** is sufficient to check for staleness. The additional condition **block.timestamp > timestamp** is redundant and adds no value. Furthermore, in unlikely cases where the price feed occasionally returns incorrect data (e.g., a **timestamp** value greater than **block.timestamp**), the staleness check does not revert as expected. This could result in inconsistent or invalid behavior, as the check fails to handle such edge cases properly.

Remediation:

Consider deleting redundant check.

LOGLAB-9 | The change in priority after requestWithdraw may block the claiming of the withdrawal request

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/vault/LogarithmVault.sol#L408-L41, src/vault/LogarithmVault.sol#L762-L764

Description:

If the priority is changed after a withdraw request is made, the withdrawal might not be able to be claimed.
The issue is that the withdrawRequest saves the **accRequestedWithdrawAssets** without noting whether the **owner** was prioritized. Upon claiming, **accRequestedWithdrawAssets** was used in conjunction with **isPrioritized**, which may not match **accRequestedWithdrawAssets**.

For example, let's assume the **prioritizedAccRequestedWithdrawAssets** is 1e18, while **accRequestedWithdrawAssets** is 10e18.

- Lines 408 - 419: If an **owner** requested withdrawal before they are prioritized, the **accRequestedWithdrawAssets** of their withdrawRequest will be set as **10e18 + assetsToRequest**.
- Lines 762 - 764: Afterwards, the **owner** was prioritized. The withdrawRequest's **accRequestedWithdrawAssets** will be compared against **$.prioritizedProcessedWithdrawAssets**. In this case it would be 1e18, therefore the withdrawal will not be able to be claimed.

It is reversable; if the **owner** should be not anymore prioritized, they can then claim their withdrawal.

```
if (isPrioritized(owner)) {
    _accRequestedWithdrawAssets = $.prioritizedAccRequestedWithdrawAssets +
assetsToRequest;
    $.prioritizedAccRequestedWithdrawAssets = _accRequestedWithdrawAssets;
} else {
    _accRequestedWithdrawAssets = $.accRequestedWithdrawAssets +
assetsToRequest;
    $.accRequestedWithdrawAssets = _accRequestedWithdrawAssets;
}
```

```
bytes32 withdrawKey = getWithdrawKey(owner, _useNonce(owner));
$.withdrawRequests[withdrawKey] = WithdrawRequest({
    requestedAssets: assetsToRequest,
    accRequestedWithdrawAssets: _accRequestedWithdrawAssets,
```

```
isExecuted = isPrioritizedAccount
    ? accRequestedWithdrawAssetsOfRequest <=
$.prioritizedProcessedWithdrawAssets
    : accRequestedWithdrawAssetsOfRequest <= $.processedWithdrawAssets;
```

Remediation:

Consider saving the priority at the time of requestWithdraw.

LOGLAB-11 | LogarithmVault.sol::maxMint is returning super.maxDeposit instead of super.maxMint

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

src/vault/LogarithmVault.sol#L652

Description:

The **LogarithmVault::maxMint** function at line 652 incorrectly calls **super.maxDeposit()**. Instead, it should convert the user's maximum deposit into shares by calling **super.maxMint()** as intended.

```
function maxMint(address receiver) public view virtual override returns
(uint256) {
    if (paused() || isShutdown()) {
        return 0;
    } else {
        return super.maxDeposit(receiver);
    }
}
```

Remediation:

Consider returning **super.maxMint(receiver)** in the maxMint function.

LOGLAB-7 | Missing Asset/Product Check When Setting New Strategy

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/vault/LogarithmVault.sol#L232-L248

Description:

The **setStrategy** function in the vault allows setting a new strategy without asserting the new asset/product stay unchanged. This means a new strategy can be applied to the vault without validating if its asset matches the vault's current asset, potentially leading to incompatibilities.

```
function setStrategy(address _strategy) external onlyOwner {
    LogarithmVaultStorage storage $ = _getLogarithmVaultStorage();

    IERC20 _asset = IERC20(asset());
    address prevStrategy = strategy();

    if (prevStrategy != address(0)) {
        IStrategy(prevStrategy).stop();
        _asset.approve(prevStrategy, 0);
    }

    require(_strategy != address(0));
    $.strategy = _strategy;
    _asset.approve(_strategy, type(uint256).max);

    emit StrategyUpdated(_msgSender(), _strategy);
}
```

Remediation:

The **setStrategy** function should validate that the new strategy's asset matches the vault's asset and perhaps check that the new strategy's product is consistent with the previous strategy's product.

LOGLAB-4 | Invalidation of setLimitDecreaseCollateral Validation Logic when setting new setCollateralMinMax

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/hedge/offchain/OffChainConfig.sol#L63-L79

Description:

limitDecreaseCollateral() is designed to optimize gas usage and cost efficiency by enforcing a minimum threshold for decreasing collateral. **decreaseCollateralMinMax()** sets the allowable minimum and maximum collateral that can be decreased. However, the interaction between the two setter functions could introduce a inconsistency.

Function to Set Limit (**setLimitDecreaseCollateral**):

```
    function setLimitDecreaseCollateral(uint256 limit) external onlyOwner {
          OffChainConfigStorage storage $ = _getOffChainConfigStorage();
          require(limit > $.decreaseCollateralMinMax[0]);
          $.limitDecreaseCollateral = limit;
      }
```

Function to Set Min-Max (**setCollateralMinMax**):

```
    function setCollateralMinMax(
          uint256 increaseCollateralMin,
          uint256 increaseCollateralMax,
          uint256 decreaseCollateralMin,
          uint256 decreaseCollateralMax
      ) external onlyOwner {
          require(increaseCollateralMin < increaseCollateralMax &&
decreaseCollateralMin < decreaseCollateralMax);
          OffChainConfigStorage storage $ = _getOffChainConfigStorage();
          $.increaseCollateralMinMax = [increaseCollateralMin,
increaseCollateralMax];
          $.decreaseCollateralMinMax = [decreaseCollateralMin,
decreaseCollateralMax];
      }
```

Root Cause

The issue exists due to the order of operations between the two functions:

1. **setCollateralMinMax** is called, setting a valid range for **decreaseCollateralMinMax**.
2. **setLimitDecreaseCollateral** is called, setting a **limitDecreaseCollateral** value that is acceptable **require(limit > $.decreaseCollateralMinMax[0])**.
3. **setCollateralMinMax** is called again, changing the **decreaseCollateralMinMax** range to new values. This invalidates the assumption that **limitDecreaseCollateral** is above the allowable **decreaseCollateralMin**, as the new **decreaseCollateralMinMax[0]** could now be greater than or lesser than **limitDecreaseCollateral**.

```
function setCollateralMinMax(
    uint256 increaseCollateralMin,
    uint256 increaseCollateralMax,
    uint256 decreaseCollateralMin,
    uint256 decreaseCollateralMax
) external onlyOwner {
    require(increaseCollateralMin < increaseCollateralMax &&
decreaseCollateralMin < decreaseCollateralMax);
    OffChainConfigStorage storage $ = _getOffChainConfigStorage();
    $.increaseCollateralMinMax = [increaseCollateralMin,
increaseCollateralMax];
    $.decreaseCollateralMinMax = [decreaseCollateralMin,
decreaseCollateralMax];
}


function setLimitDecreaseCollateral(uint256 limit) external onlyOwner {
    OffChainConfigStorage storage $ = _getOffChainConfigStorage();
    require(limit > $.decreaseCollateralMinMax[0]);
    $.limitDecreaseCollateral = limit;
}
```

Remediation:

Validate existing limit in **setCollateralMinMax** by adding a check in **setCollateralMinMax** to ensure that the current **limitDecreaseCollateral** remains valid with the updated **decreaseCollateralMinMax**.

LOGLAB-1 | Missing disableinitializers() to prevent uninitialized contracts

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

All contracts in the project that use OpenZeppelin's **Initializable** don't call **_disableInitializers** in the constructor per the OpenZeppelin documentation:

Writing Upgradeable Contracts - OpenZeppelin Docs

Proxies - OpenZeppelin Docs

Contract implementations could be initialized when this should not be possible.

For example: src/strategy/BasisStrategy.sol

```
    function initialize(
        address _config,
        address _product,
        address _vault,
        address _oracle,
        address _operator,
        uint256 _targetLeverage,
        uint256 _minLeverage,
        uint256 _maxLeverage,
        uint256 _safeMarginLeverage
@>  ) external initializer {
        __Ownable_init(_msgSender());

  [.....]
```

Remediation:

We recommend using the constructor, which calls the **_disableInitializers()** function to block initialize calls in the implementation:

```
constructor() {
 _disableInitializers();
}
```

LOGLAB-20 | BasisStrategy::_afterIncreasePosition may send asset to vault without LogarithmVault.processingPendingWithdrawRequests

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

src/vault/LogarithmVault.sol#L659-L666

Description:

The exit fee is calculated based on the amount of assets to be deutilized for the redeem/withdraw. It means only the asset amount above the **idleAssets** is subject to exit fee.

Also note that if asset is sent to the vault, it should call the **LogarithmVault.processPendingWithdrawRequests()** to ensure that the sent asset is processed for the pending withdraw. So, if there is pending withdrawal, the vault does not consider the sent asset to be 'idle'.

In the **afterAdjustPosition** called by the hedgeManager, if the adjust was to utilize, but the requested collateral delta amount is bigger than the response collateral delta amount (exceeding threshold), the deviation is sent back to the vault:

```
                    $.asset.safeTransferFrom(hedgeManager(), vault(), uint256(-
collateralDeviation));
```

Even though this will make the **BasisStrategy** to pause, one can still request withdraw on the vault. If somebody is calling **requestWithdraw** after this particular case of the **afterAdjustPosition**, the exit fee might consider this collateral deviation as idle asset and incorrectly calculate the exit fee. On top of it, this asset sent by hedgeManager via BasisStrategy will be withdrawn without being queued or processed.

```
    function maxWithdraw(address owner) public view virtual override returns
(uint256) {
        if (paused()) {
            return 0;
        }
        uint256 assets = super.maxWithdraw(owner);
        uint256 withdrawableAssets = idleAssets();
        return assets > withdrawableAssets ? withdrawableAssets : assets;
    }
```

```
// BasisStrategy::_afterIncreasePosition
if (requestParams.collateralDeltaAmount > 0) {
  (bool exceedsThreshold, int256 collateralDeviation) = _checkDeviation(
      responseParams.collateralDeltaAmount,
requestParams.collateralDeltaAmount, _responseDeviationThreshold
  );
  if (exceedsThreshold) {
      if (collateralDeviation < 0) {
          shouldPause = true;
          $.asset.safeTransferFrom(hedgeManager(), vault(), uint256(-
collateralDeviation));
      }
  }
```

Remediation:

Consider calling **LogarithmVault.processPendingWithdrawRequests()** after transferring asset to vault.

LOGLAB-8 | Use Custom Errors

Fixed

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Description:

Some contract's are currently handling errors by using native Solidity **revert** statements instead of custom errors.

For example:

```
function uniswapV3SwapCallback(int256 amount0Delta, int256 amount1Delta, bytes
calldata data) external {
        require(amount0Delta > 0 || amount1Delta > 0); // swaps entirely within
0-liquidity regions are not supported
```

Additionally, add custom errors to the existing revert cases that lack descriptions.

For example:

```
function _setManualSwapPath(address[] calldata _assetToProductSwapPath, address
_asset, address _product) private {
  --- SNIP ----
    if (length % 2 == 0 || _assetToProductSwapPath[0] != _asset ||
_assetToProductSwapPath[length - 1] != _product)
    {
        // length should be odd
        // the first element should be asset
        // the last element should be product
        revert();
    }
}
```

Remediation:

Replace **require** statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

For example:

```
++ error ZeroLiquiditySwapNotSupported();

function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external {
--  require(amount0Delta > 0 || amount1Delta > 0); // swaps entirely within 0-liquidity regions are not supported
++  if (amount0Delta <= 0 && amount1Delta <= 0) {
++        revert ZeroLiquiditySwapNotSupported();
++   }

}
```

LOGLAB-3 | Use Ownable2StepUpgradeable for all contracts

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

In the project, only the **src/oracle/LogarithmOracle.sol** and **src/hedge/gmx/GmxGasStation.sol** contracts utilize the **Ownable2StepUpgradeable** contract, while the rest rely on **OwnableUpgradeable**.

To maintain consistency and adhere to best practices, it is recommended to implement the **Ownable2StepUpgradeable** pattern across all contracts. The two-step ownership transfer provides an added layer of security, reducing the risk of accidental or malicious ownership changes.

Remediation:

Consider using the **Ownable2StepUpgradeable** across all the contracts to maintain the consistency.

LOGLAB-2 | Constant variables should be marked as private

Fixed

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

src/vault/ManagedVault.sol#L27-L30, src/oracle/LogarithmOracle.sol#L18

Description:

In the following locations, there are constant variables that are declared **public**. However, setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

```
    /// @notice The maximum value of management fee that can be configured.
    uint256 public constant MAX_MANAGEMENT_FEE = 5e16; // 5%
    /// @notice The maximum value of performance fee that can be configured.
    uint256 public constant MAX_PERFORMANCE_FEE = 5e17; // 50%
```

```
    uint256 public constant FLOAT_PRECISION = 1e30;
```

Remediation:

The mentioned variables should be marked as **private** instead of **public**.


hexens
x
basisOS