code4rena

Gemnify

Smart Contract Security Assessment

Version 2.0

Audit dates: Jul 03 — Jul 26, 2024

Audited by: cccz
peakbolt

Contents

1. Introduction
   - 1.1 About Zenith
   - 1.2 Disclaimer
   - 1.3 Risk Classification
2. Executive Summary
   - 2.1 About Gemnify
   - 2.2 Scope
   - 2.3 Audit Timeline
   - 2.4 Issues Found
3. Findings Summary
4. Findings
   - 4.1 High Risk
   - 4.2 Medium Risk
   - 4.3 Low Risk
   - 4.4 Informational

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Gemnify

Gemnify is an innovative decentralized perpetual exchange designed for leverage trading on relatively pegged assets including USDT, DAI, USDe and GHO. The platform operates on the Arbitrum network, offering users a secure environment for engaging in leverage trading.

2.2 Scope

Repository	GMX-For-NFT/gemnify-contract
Commit Hash	4810eb86d3391ba9e2eab26a725e9c7f63a3fa9e
Mitigation Hash	31814224ff65d5c7d9b986371e08146394a56b43

2.3 Audit Timeline

DATE	EVENT
Jul 03, 2024	Audit start
Jul 26, 2024	Audit end
Oct 25, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	6
Medium Risk	10
Low Risk	3
Informational	2
Total Issues	21

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	The average price was calculated incorrectly.	Resolved

H-2	Incorrect impactPool increase when decreasing positions	Resolved
H-3	When calling applyDeltaToPositionImpactPool() in reduceCollateral(), delta's decimals is incorrect	Resolved
H-4	ExecuteBuyUSDG() and ExcuteSwap() tracked incorrect usdg amounts and poolAmounts.	Resolved
H-5	`claimFundingFees()` should use `GenericLogic.TransferOut()`	Resolved
H-6	`fundingFeeAmount` is incorrectly scaled to 18 decimals instead of 30 decimals	Resolved
M-1	User could lose out on positive impact fee when decreasing position	Resolved
M-2	`minProfitTime` could cause unintended effects in scenarios	Resolved
M-3	ExecuteSwap() is better to collect fee before applying priceImpact	Acknowledged
M-4	ExecuteSellUSDG() should also apply validateBufferAmount()	Acknowledged
M-5	Incorrect feeBP calculation in ExecuteSellUSDG	Resolved
M-6	nextDiffUsdToken in SwapPriceImpactLogic._getPriceImpactUsd() is inaccurate	Resolved
M-7	Incorrect calculation in `getRealisedPnl()` affects `ULP` price	Resolved
M-8	Lack of slippage protection for position execution price after price impact	Resolved
M-9	`setFundingFactor()` fails to update funding state	Resolved
M-10	Protocol fails to update cumulative borrowing rate in certain scenarios	Resolved
L-1	USDG minted in ExecuteBuyUSDG() and the return value may be different, which will make slippage control	Acknowledged

	invalid.	
L-2	getSupplyPriceImpactUsd() is better to use priceMid instead of priceMax	Acknowledged
L-3	`getNextFundingAmountPerSize()` calls `getFundingAmountPerSizeDelta()` with incorrect parameters	Resolved
I-1	Some checks in validateLiquidation() that could be improved	Resolved
I-2	validateTokens() should add additional validation	Resolved

4. Findings

4.1 High Risk

A total of 6 high risk findings were identified.

[H-1] The average price was calculated incorrectly.

Severity: High	Status: Resolved

Context:

- PositionLogic.sol#L420-L438
- PositionLogic.sol#L841-L877
- PositionLogic.sol#L879-L915
- PositionLogic.sol#L939-L956

Description:

The protocol was calculating the position average price and the global average price incorrectly due to mixing raw price and priceImpact price and not taking priceImpactUsd into account. Since priceImpact price only applies to the increased size, rawPrice should be used when calculating the delta of the profit or loss of the position. And priceImpactUsd should be directly considered as the profit or loss of the position, so priceImpactUsd should be considered in the delta.

Recommendation:

```
    function getNextAveragePrice(
        address _indexToken,
        uint256 _size,
        uint256 _averagePrice,
        bool _isLong,
        uint256 _nextPrice,
        uint256 _sizeDelta,
        uint256 _lastIncreasedTime
    ) internal view returns (uint256) {
        (bool hasProfit, uint256 delta) = getDelta(_indexToken, _size,
_averagePrice, _isLong, _lastIncreasedTime);
        uint256 nextSize = _size + _sizeDelta;
        uint256 divisor;
+       uint256 priceImpactUsd;
```

```
+       uint256 rawPrice = _isLong ?
GenericLogic.getMaxPrice(_indexToken) :
GenericLogic.getMinPrice(_indexToken);
+       if (_nextPrice > rawPrice) {
+           priceImpactUsd = (_nextPrice - rawPrice) * (_sizeDelta /
_nextPrice);
+       } else {
+           priceImpactUsd = (rawPrice - _nextPrice) * (_sizeDelta /
_nextPrice);
+       }
+
        if (_isLong) {
            divisor = hasProfit ? nextSize + delta : nextSize - delta;
+           if(_nextPrice > rawPrice) { divisor - priceImpactUsd};
+           else {divisor + priceImpactUsd};
        } else {
            divisor = hasProfit ? nextSize - delta : nextSize + delta;
+           if(_nextPrice > rawPrice) { divisor + priceImpactUsd};
+           else {divisor - priceImpactUsd};
        }
-       return (_nextPrice * nextSize) / divisor;
+       return (rawPrice * nextSize) / divisor;
    }
```

```
    function getNextGlobalLongData(
        address _account,
        address _collateralToken,
        address _indexToken,
        uint256 _nextPrice,
        uint256 _sizeDelta,
        bool _isIncrease
    ) internal view returns (uint256) {
        DataTypes.PositionStorage storage ps =
StorageSlot.getVaultPositionStorage();

        int256 realisedPnl = getRealisedPnl(_account, _collateralToken,
_indexToken, _sizeDelta, _isIncrease, true);
        uint256 globalLongSize = ps.globalLongSizes[_indexToken];
        uint256 averagePrice = ps.globalLongAveragePrices[_indexToken];
+       uint256 rawPrice = GenericLogic.getMinPrice(_indexToken);
+       uint256 priceDelta = averagePrice > rawPrice ? averagePrice -
rawPrice : rawPrice - averagePrice;
-       uint256 priceDelta = averagePrice > _nextPrice ? averagePrice -
_nextPrice : _nextPrice - averagePrice;
...
```

```
         uint256 nextAveragePrice =
-            getNextGlobalAveragePrice(averagePrice, _nextPrice, nextSize,
delta, realisedPnl, true);
+            getNextGlobalAveragePrice(averagePrice, _nextPrice, nextSize,
delta, realisedPnl, true, _isIncrease);
```

```
    function getNextGlobalShortData(
        address _account,
        address _collateralToken,
        address _indexToken,
        uint256 _nextPrice,
        uint256 _sizeDelta,
        bool _isIncrease
    ) internal view returns (uint256) {
        DataTypes.PositionStorage storage ps =
StorageSlot.getVaultPositionStorage();

        int256 realisedPnl = getRealisedPnl(_account, _collateralToken,
_indexToken, _sizeDelta, _isIncrease, false);
        uint256 globalShortSize = ps.globalShortSizes[_indexToken];
        uint256 averagePrice = ps.globalShortAveragePrices[_indexToken];
+       uint256 rawPrice = GenericLogic.getMaxPrice(_indexToken);
+       uint256 priceDelta = averagePrice > rawPrice ? averagePrice -
rawPrice : rawPrice - averagePrice;
-       uint256 priceDelta = averagePrice > _nextPrice ? averagePrice -
_nextPrice : _nextPrice - averagePrice;
...
        uint256 nextAveragePrice =
-           getNextGlobalAveragePrice(averagePrice, _nextPrice, nextSize,
delta, realisedPnl, false);
+           getNextGlobalAveragePrice(averagePrice, _nextPrice, nextSize,
delta, realisedPnl, false, _isIncrease);
```

```
    function getNextGlobalAveragePrice(
        uint256 _averagePrice,
        uint256 _nextPrice,
        uint256 _nextSize,
        uint256 _delta,
        int256 _realisedPnl,
        bool _isLong,
+       bool _isIncrease
    ) internal pure returns (uint256) {
+       uint256 rawPrice = _isIncrease ? ( _isLong?
GenericLogic.getMaxPrice(_indexToken) :
```

```
GenericLogic.getMinPrice(_indexToken))  : (_isLong ?
GenericLogic.getMinPrice(_indexToken) :
GenericLogic.getMaxPrice(_indexToken));
-       (bool hasProfit, uint256 nextDelta) = getNextDelta(_delta,
_averagePrice, _nextPrice, _realisedPnl, _isLong);
+       (bool hasProfit, uint256 nextDelta) = getNextDelta(_delta,
_averagePrice, rawPrice, _realisedPnl, _isLong);
        uint256 divisor;
+       uint256 priceImpactUsd;
+       if (_nextPrice > rawPrice) {
+           priceImpactUsd = (_nextPrice - rawPrice) * (_sizeDelta /
_nextPrice);
+       } else {
+           priceImpactUsd = (rawPrice - _nextPrice) * (_sizeDelta /
_nextPrice);
+       }
+
        if (_isLong) {
            divisor = hasProfit ? _nextSize + nextDelta : _nextSize -
nextDelta;
+           if(_nextPrice > rawPrice) { divisor - priceImpactUsd};
+           else {divisor + priceImpactUsd};
        } else {
            divisor = hasProfit ? _nextSize - nextDelta : _nextSize +
nextDelta;
+           if(_nextPrice > rawPrice) { divisor + priceImpactUsd};
+           else {divisor - priceImpactUsd};
        }
-       return (_nextPrice * _nextSize) / divisor;
+       return (rawPrice * _nextSize) / divisor;
    }
```

Gemnify: The fix has been implemented with the following commit

Zenith: Verified

[H-2] Incorrect impactPool increase when decreasing positions

Severity: High	Status: Resolved

Context:

- PositionLogic.sol#L642-L661

Description: When decreasing position, if there is a negative priceImpact, will try to increase the impactPool with the funds withdrawn by the user. The problem here is that even if the funds withdrawn by the user are not enough, the impactPool will be increased in full, which results in the possibility that the user can increase 1000 USD impactPool by 1 USD.

Recommendation:

```
if (GenericLogic.isPriceImpactEnabled(params.indexToken, false)) {
            uint256 priceMax =
GenericLogic.getMaxPrice(params.indexToken);
            uint256 priceMin =
GenericLogic.getMinPrice(params.indexToken);
            (int256 priceImpactUsd,) =
getExecutionPriceForDecrease(params, priceMax, priceMin);
            if (priceImpactUsd > 0) {
                cache.usdOutAfterFee += priceImpactUsd.toUint256();
            } else {
                if (cache.usdOutAfterFee > (-priceImpactUsd).toUint256())
{
                    cache.usdOutAfterFee -= (-
priceImpactUsd).toUint256();
                } else {
+                   priceImpactUsd = -cache.usdOutAfterFee.toInt256();
                    cache.usdOutAfterFee = 0;
                }
            }
            // if there is a positive impact, the impact pool amount
should be reduced
            // if there is a negative impact, the impact pool amount
should be increased
            PositionPriceImpactLogic.applyDeltaToPositionImpactPool(
                params.indexToken, -(priceImpactUsd *
Constants.PRICE_PRECISION.toInt256() / priceMin.toInt256())
            );
        }
```

Gemnify: Fix has been implemented with the following commit

Zenith: Verified

[H-3] When calling applyDeltaToPositionImpactPool() in reduceCollateral(), delta's decimals is incorrect

Severity: High	Status: Resolved

Context:

- PositionLogic.sol#L658-L660

Description:

When converting priceImpactUsd to priceImpactAmount, it is necessary to multiply by PRICE_PRECISION to normalize the decimals. But this is not done when calling applyDeltaToPositionImpactPool() in reduceCollateral(), this causes the passed delta parameter to be much smaller (1e30 times) than the correct value, making the update of positionImpactPoolAmounts incorrect.

Recommendation:

```
            PositionPriceImpactLogic.applyDeltaToPositionImpactPool(
-               params.indexToken, -(priceImpactUsd / priceMin.toInt256())
+               params.indexToken, -(priceImpactUsd * Constants.PRICE_PRECISION.toInt256() / priceMin.toInt256())
            );
```

Gemnify: Fix has been implemented with the following commit

Zenith: Verified

[H-4] ExecuteBuyUSDG() and ExcuteSwap() tracked incorrect usdg amounts and poolAmounts.

Severity: High	Status: Resolved

Context:

- SupplyLogic.sol#L67-L102
- SwapLogic.sol#L124-L136

Description:

Based on discussions with sponsors, tracking of usdg amounts and poolamounts needs to be based on the following principles.

1. balance reward only affect the return value, as the ulp amount minted to the user in ulpManager
2. balance reward does not affect poolAmounts, but collectFee will reduce poolAmounts(also reduce usdgAmount)
3. priceImpact affects poolAmounts and minted usdg. negative priceImapct stores some tokens into swapImpactPoolAmounts, which decreases poolAmounts and minted usdg, positive priceImapct takes some tokens out of the swapImpactPoolAmounts, which increases poolAmounts and minted usdg.

However ExecuteBuyUSDG() and ExcuteSwap() do not follow them, one direct impact is that the usdg amount minted is incorrect.

Recommendation:

```
    function ExecuteBuyUSDG(address _token, address _receiver) external
returns (uint256) {
        DataTypes.AddressStorage storage addrs =
StorageSlot.getVaultAddressStorage();
        ValidationLogic.validateManager();
        ValidationLogic.validateWhitelistedToken(_token);

        uint256 tokenAmount;

        tokenAmount = GenericLogic.transferIn(_token);

        ValidationLogic.validate(tokenAmount > 0,
Errors.VAULT_INVALID_TOKEN_AMOUNT);


BorrowingFeeLogic.updateCumulativeBorrowingRate(addrs.collateralToken,
```

```
addrs.collateralToken);

        uint256 price = GenericLogic.getMinPrice(_token);
        uint256 usdgAmount = (tokenAmount * price) /
Constants.PRICE_PRECISION;

        usdgAmount = GenericLogic.adjustForDecimals(usdgAmount, _token,
addrs.usdg);

        ValidationLogic.validate(usdgAmount > 0,
Errors.VAULT_INVALID_USDG_AMOUNT);

        int256 feeBasisPoints =
GenericLogic.getBuyUsdgFeeBasisPoints(_token, usdgAmount);

-       uint256 amountAfterFees = GenericLogic.collectSwapFees(_token,
tokenAmount, feeBasisPoints);
+       uint256 amountAfterFees0 = GenericLogic.collectSwapFees(_token,
tokenAmount, feeBasisPoints);

        // price impact
        int256 priceImpactUsd;
+       uint256 amountAfterFees = amountAfterFees0;
+       uint256 priceImpactUsdg;
        if (GenericLogic.isPriceImpactEnabled(_token, true)) {
            uint256 priceMax = GenericLogic.getMaxPrice(_token);
            priceImpactUsd =
SwapPriceImpactLogic.getSupplyPriceImpactUsd(
                SwapPriceImpactLogic.GetSupplyPriceImpactUsdParams({
                    token: _token,
                    price: priceMax,
                    usdDelta: ((usdgAmount * Constants.PRICE_PRECISION) /
10 ** Constants.USDG_DECIMALS).toInt256()
                })
            );
            int256 impactAmount =
SwapPriceImpactLogic.applySwapImpactWithCap(_token, priceMax, price,
priceImpactUsd);
            if (priceImpactUsd > 0) {
                uint256 positiveImpactAmount =
GenericLogic.adjustFor30Decimals(impactAmount.toUint256(), _token);
-               amountAfterFees += positiveImpactAmount;
+               priceImpactUsdg = impactAmount.toUint256() * price /
Constants.PRICE_PRECISION;
+               priceImpactUsdg =
GenericLogic.adjustFor30Decimals(priceImpactUsdg, addr.usgd);
```

```
+                amountAfterFees = amountAfterFees0 +
positiveImpactAmount;
             }
             if (priceImpactUsd < 0) {
                 uint256 negativeImpactAmount =
GenericLogic.adjustFor30Decimals((-impactAmount).toUint256(), _token);
-                amountAfterFees -= negativeImpactAmount;
+                priceImpactUsdg = (-impactAmount).toUint256() * price /
Constants.PRICE_PRECISION;
+                priceImpactUsdg =
GenericLogic.adjustFor30Decimals(priceImpactUsdg, addr.usgd);
+                amountAfterFees = amountAfterFees0 -
negativeImpactAmount;
             }
         }

         uint256 mintAmount = (amountAfterFees * price) /
Constants.PRICE_PRECISION;
         mintAmount = GenericLogic.adjustForDecimals(mintAmount, _token,
addrs.usdg);
+        if (amountAfterFees0 > tokenAmount) { // reward
+            if(amountAfterFees > amountAfterFees0) { // positive
priceImpact
+                GenericLogic.increaseUsdgAmount(_token, usdgAmount +
priceImpactUsdg);
+                GenericLogic.increasePoolAmount(_token, tokenAmount +
amountAfterFees - amountAfterFees0);
+                IUSDG(addrs.usdg).mint(_receiver, usdgAmount +
priceImpactUsdg);
+            } else {                                  // negative
priceImpact
+                GenericLogic.increaseUsdgAmount(_token, usdgAmount -
priceImpactUsdg);
+                GenericLogic.increasePoolAmount(_token, tokenAmount +
amountAfterFees - amountAfterFees0); // reduce poolAmounts
+                IUSDG(addrs.usdg).mint(_receiver, usdgAmount -
priceImpactUsdg);
+            }
+        } else {                                  // collect fee
+            uint256 mintAmount0 = (amountAfterFees0 * price) /
Constants.PRICE_PRECISION; // collectFee reduce usdgAmount
+            mintAmount0 = GenericLogic.adjustForDecimals(mintAmount0,
_token, addrs.usdg);
+            if(amountAfterFees > amountAfterFees0) { // positive
priceImpact
```

```
+                GenericLogic.increaseUsdgAmount(_token, mintAmount0 +
priceImpactUsdg);
+                GenericLogic.increasePoolAmount(_token, amountAfterFees);
+                IUSDG(addrs.usdg).mint(_receiver, mintAmount0 +
priceImpactUsdg);
+            } else {                                    // negative
priceImpact
+                GenericLogic.increaseUsdgAmount(_token, mintAmount0 -
priceImpactUsdg);
+                GenericLogic.increasePoolAmount(_token, amountAfterFees);
// reduce poolAmounts
+                IUSDG(addrs.usdg).mint(_receiver, mintAmount0 -
priceImpactUsdg);
+            }
+        }
-        if (amountAfterFees > tokenAmount) {
-            GenericLogic.increaseUsdgAmount(_token, usdgAmount);
-            GenericLogic.increasePoolAmount(_token, tokenAmount);
-            IUSDG(addrs.usdg).mint(_receiver, usdgAmount);
-        } else {
-            GenericLogic.increaseUsdgAmount(_token, mintAmount);
-            GenericLogic.increasePoolAmount(_token, amountAfterFees);
-            IUSDG(addrs.usdg).mint(_receiver, mintAmount);
-        }

        emit BuyUSDG(_receiver, _token, tokenAmount, mintAmount,
feeBasisPoints, priceImpactUsd);

        return mintAmount;
    }
```

```
    function ExecuteSwap(address _tokenIn, address _tokenOut, uint256
_amountIn, address _receiver)
        external
        returns (uint256)
    {
        PriceImpactCache memory cache;

        DataTypes.TokenConfigSotrage storage ts =
StorageSlot.getVaultTokenConfigStorage();
        DataTypes.PermissionStorage storage ps =
StorageSlot.getVaultPermissionStorage();
        DataTypes.AddressStorage storage addrs =
StorageSlot.getVaultAddressStorage();
```

```
        ValidationLogic.validateSwapParams(ps.isSwapEnabled, _tokenIn,
_tokenOut, _amountIn, ts.whitelistedTokens);


BorrowingFeeLogic.updateCumulativeBorrowingRate(addrs.collateralToken,
addrs.collateralToken);

        cache.tokenIn = _tokenIn;
        cache.tokenOut = _tokenOut;
        cache.priceInMax = GenericLogic.getMaxPrice(_tokenIn);
        cache.priceInMin = GenericLogic.getMinPrice(_tokenIn);
        cache.priceOutMax = GenericLogic.getMaxPrice(_tokenOut);
        cache.priceOutMin = GenericLogic.getMinPrice(_tokenOut);

        // adjust usdgAmounts by the same usdgAmount as debt is shifted
between the assets
        uint256 usdgAmount = (_amountIn * cache.priceInMin) /
Constants.PRICE_PRECISION;
        usdgAmount = GenericLogic.adjustForDecimals(usdgAmount, _tokenIn,
addrs.usdg);
        // price impact tokenIn
        if (GenericLogic.isPriceImpactEnabled(_tokenIn, true)) {
            cache.priceImpactUsdTokenIn =
SwapPriceImpactLogic.getSupplyPriceImpactUsd(
                SwapPriceImpactLogic.GetSupplyPriceImpactUsdParams({
                    token: cache.tokenIn,
                    price: cache.priceInMax,
                    usdDelta: ((usdgAmount * Constants.PRICE_PRECISION) /
10 ** Constants.USDG_DECIMALS).toInt256()
                })
            );
            int256 impactAmountTokenIn =
SwapPriceImpactLogic.applySwapImpactWithCap(
                cache.tokenIn, cache.priceInMax, cache.priceInMin,
cache.priceImpactUsdTokenIn
            );
            if (cache.priceImpactUsdTokenIn > 0) {
                uint256 positiveImpactAmountTokenIn =

GenericLogic.adjustFor30Decimals(impactAmountTokenIn.toUint256(),
cache.tokenIn);
                _amountIn += positiveImpactAmountTokenIn;
            }

            if (cache.priceImpactUsdTokenIn < 0) {
                uint256 negativeImpactAmount =
```

```
                    GenericLogic.adjustFor30Decimals((-
impactAmountTokenIn).toUint256(), cache.tokenIn);
                _amountIn -= negativeImpactAmount;
            }
        }

        uint256 amountOut = (_amountIn * cache.priceInMin) /
cache.priceOutMax;

        amountOut = GenericLogic.adjustForDecimals(amountOut, _tokenIn,
_tokenOut);

        int256 feeBasisPoints = getSwapFeeBasisPoints(_tokenIn,
_tokenOut, usdgAmount);
-       uint256 amountOutAfterFees =
GenericLogic.collectSwapFees(_tokenOut, amountOut, feeBasisPoints);
+       uint256 amountOutAfterFees0 =
GenericLogic.collectSwapFees(_tokenOut, amountOut, feeBasisPoints);
+       uint256 amountOutAfterFees = amountOutAfterFees0;
        // price impact tokenOut
        if (GenericLogic.isPriceImpactEnabled(_tokenOut, true)) {
            cache.priceImpactUsdTokenOut =
SwapPriceImpactLogic.getSupplyPriceImpactUsd(
                SwapPriceImpactLogic.GetSupplyPriceImpactUsdParams({
                    token: cache.tokenOut,
                    price: cache.priceOutMax,
                    usdDelta: -(((usdgAmount * Constants.PRICE_PRECISION)
/ 10 ** Constants.USDG_DECIMALS).toInt256())
                })
            );
            int256 impactAmountOut =
SwapPriceImpactLogic.applySwapImpactWithCap(
                cache.tokenOut, cache.priceOutMax, cache.priceOutMin,
cache.priceImpactUsdTokenOut
            );
            if (cache.priceImpactUsdTokenOut > 0) {
                uint256 positiveImpactAmount =

GenericLogic.adjustFor30Decimals(impactAmountOut.toUint256(),
cache.tokenOut);
-               amountOutAfterFees += positiveImpactAmount;
+               amountOutAfterFees = amountOutAfterFees0 +
positiveImpactAmount;
            }
            if (cache.priceImpactUsdTokenOut < 0) {
                uint256 negativeImpactAmount =
```

```
                    GenericLogic.adjustFor30Decimals((-
impactAmountOut).toUint256(), cache.tokenOut);
-               amountOutAfterFees -= negativeImpactAmount;
+               amountOutAfterFees = amountOutAfterFees0 -
negativeImpactAmount;
            }
        }

        GenericLogic.increaseUsdgAmount(_tokenIn, usdgAmount);
        GenericLogic.increasePoolAmount(_tokenIn, _amountIn);
+       GenericLogic.decreaseUsdgAmount(_tokenOut, usdgAmount);
+       if(amountOutAfterFees0 > amountOut)
+           GenericLogic.decreasePoolAmount(_tokenOut,
amountOutAfterFees0, false);
+       else
+           GenericLogic.decreasePoolAmount(_tokenOut, amountOut, false);
-       if (amountOutAfterFees > amountOut) {
-           uint256 usdgAmountAfterFee = (amountOutAfterFees *
cache.priceOutMin) / Constants.PRICE_PRECISION;

-           usdgAmountAfterFee =
GenericLogic.adjustForDecimals(usdgAmount, _tokenOut, addrs.usdg);
-           GenericLogic.decreaseUsdgAmount(_tokenOut,
usdgAmountAfterFee);
-           GenericLogic.decreasePoolAmount(_tokenOut,
amountOutAfterFees, false);
-       } else {
-           GenericLogic.decreaseUsdgAmount(_tokenOut, usdgAmount);
-           GenericLogic.decreasePoolAmount(_tokenOut, amountOut, false);
-       }

        ValidationLogic.validateBufferAmount(_tokenOut);

        GenericLogic.transferOut(_tokenOut, amountOutAfterFees,
_receiver);

        emit Swap(_receiver, _tokenIn, _tokenOut, _amountIn, amountOut,
amountOutAfterFees, feeBasisPoints);

        return amountOutAfterFees;
    }
```

Gemnify: Fix has been implemented with the following commit

Zenith: Verified.

[H-5] `claimFundingFees()` should use `GenericLogic.TransferOut()`

Severity: High	Status: Resolved

Context:

- Vault.sol#L540

Description: In `Vault.claimFundingFees()`, the `claimableFundingAmount` is transferred out using `safeTransfer()`. This is incorrect as it will not update `ps.tokenBalances[]` with the new token balance, causing the next `GenericLogic.transferIn()` to return the wrong amount received.

```
    function claimFundingFees() external nonReentrant returns (uint256) {
        DataTypes.FeeStorage storage fs =
StorageSlot.getVaultFeeStorage();
        DataTypes.AddressStorage storage addrs =
StorageSlot.getVaultAddressStorage();

        uint256 claimableFundingAmount =
fs.claimableFundingAmount[msg.sender];

        if (claimableFundingAmount > 0) {

IERC20Upgradeable(addrs.collateralToken).safeTransfer(msg.sender,
claimableFundingAmount);

            fs.claimableFundingAmount[msg.sender] = 0;

            emit ClaimFundingFee(msg.sender, addrs.collateralToken,
claimableFundingAmount);
        }
        return claimableFundingAmount;
    }
```

Recommendation: This should use `GenericLogic.transferOut()` instead of `safeTransfer()` so that it will update `ps.tokenBalances[_token]`.

Gemnify: Recommendation implemented with the following commit

Zenith: Verified

[H-6] `fundingFeeAmount` is incorrectly scaled to 18 decimals instead of 30 decimals

Severity: High	Status: Resolved

Context:

- FundingFeeLogic.sol#L194-L198

Description:

Note: this issue is discovered by sponsor during the audit period and then discussed with the auditors.

`FundingFeeLogic.getFundingFees()` returns the new `fundingFeeAmount` and `claimableAmount` since the last funding fee payment.

As `claimableAmount` is the amount of funding fee that can be claimed by the receiving side, it is scaled and stored as `COLLATERAL_PRECISION` (18 decimals), to facilitate the actual transfer during claiming.

However, the scaling to 18 decimals was performed in `getFundingAmountPerSizeDelta()`, which also affects the `fundingFeeAmount` and cause it to be incorrectly scaled to 18 decimals instead of 30 decimals.

Due to this issue, the amount of funding fee paid will be much lesser than the amount of funding fee claimed, causing the protocol to incur a loss over time.

```
    function getFundingAmountPerSizeDelta(
        uint256 fundingUsd,
        uint256 openInterest,
        uint256 tokenPrice,
        bool roundUpMagnitude
    ) internal pure returns (uint256) {
        if (fundingUsd == 0 || openInterest == 0) {
            return 0;
        }
        uint256 fundingUsdPerSize = Precision.mulDiv(
            fundingUsd, Precision.FLOAT_PRECISION *
Precision.FLOAT_PRECISION_SQRT, openInterest, roundUpMagnitude
        );

        if (roundUpMagnitude) {
>>>         return Calc.roundUpDivision(fundingUsdPerSize *
Constants.COllATERAL_PRECISION, tokenPrice);
        } else {
```

```
>>>         return (fundingUsdPerSize * Constants.COllATERAL_PRECISION) /
tokenPrice;
        }
    }
```

Recommendation: Within `FundingFeeLogic.getFundingAmountPerSizeDelta()`, removing the scaling to `COLLATERAL_PRECISION` as follows,

```
    function getFundingAmountPerSizeDelta(
        ...

        if (roundUpMagnitude) {
-           return Calc.roundUpDivision(fundingUsdPerSize *
Constants.COllATERAL_PRECISION, tokenPrice);
+           return Calc.roundUpDivision(fundingUsdPerSize, tokenPrice);
        } else {
-           return (fundingUsdPerSize * Constants.COllATERAL_PRECISION)
/ tokenPrice;
+           return fundingUsdPerSize / tokenPrice;


        }
    }
```

And within `Vault.getFundingAmount()`, remove the division by `Precision.FLOAT_PRECISION` as follows,

```
    function getFundingAmount(
        uint256 latestFundingAmountPerSize,
        uint256 positionFundingAmountPerSize,
        uint256 positionSizeInUsd,
        bool roundUpMagnitude
    ) internal pure returns (uint256) {
        uint256 fundingDiffFactor = (latestFundingAmountPerSize -
positionFundingAmountPerSize);

        return Precision.mulDiv(
            positionSizeInUsd,
            fundingDiffFactor,
-            Precision.FLOAT_PRECISION * Precision.FLOAT_PRECISION_SQRT,
+           Precision.FLOAT_PRECISION_SQRT,
            roundUpMagnitude
        );
```

```
    }
```

Finally within `Vault.sol` scale `claimableFundingAmount` to token decimals as follows,

```
    function claimFundingFees() external nonReentrant returns (uint256) {
        ...
        if (claimableFundingAmount > 0) {
+           claimableFundingAmount =
GenericLogic.usdToTokenMin(addrs.collateralToken,
claimableFundingAmount);
```

Gemnify: Recommendation implemented with the following commit

Zenith: Verified.

4.2 Medium Risk

A total of 10 medium risk findings were identified.

[M-1] User could lose out on positive impact fee when decreasing position

Severity: Medium	Status: Resolved

Context:

- PositionLogic.sol#L318-L322

Description: In `_decreasePosition()`, the user will receive a token amount based on the value of `cache.usdOutAfterFee`, which may include realized profit, removed collateral, and any positive impact fees. The transfer of tokens to the user will only take place when `cache.usdOut > 0`, indicating the presence of realized profit and/or removed collateral.

```
        if (cache.usdOut > 0) {
            cache.amountOutAfterFees =
GenericLogic.usdToTokenMin(params.collateralToken, cache.usdOutAfterFee);
            GenericLogic.transferOut(params.collateralToken,
cache.amountOutAfterFees, params.receiver);
            return cache.amountOutAfterFees;
        }
```

However, it is possible for `usdOut == 0` while `usdOutAfterFee > 0`.

- `usdOut == 0` could happen if the position is decreased without removing collateral and there is no profit.
- `usdOutAfterFee > 0` could occur when the position decrease resulted in a positive impact.

That means the user will lose out on the positive impact fee when these happen, as currently, the transfer only happens on `usdOut > 0`.

Recommendation: To resolve this, the change can be made as follows:

```
- if (cache.usdOut > 0) {
+ if (cache.usdOutAfterFee > 0) {
```

Gemnify: Recommendation implemented in commit

Zenith: Verified

[M-2] `minProfitTime` could cause unintended effects in scenarios

Severity: Medium	Status: Resolved

Context:

- PositionLogic.sol#L462-L468

Description:

In `PositionLogic.sol`, a min profit cap is imposed when the time of last position creation/increase is within the `minProfitTime`. That is in place to prevent traders from frontrunning on-chain price updates as those prices are obtained from the other exchanges.

```
        // if the minProfitTime has passed then there will be no min
profit threshold
        // the min profit threshold helps to prevent front-running issues
        uint256 minBps =
            block.timestamp > _lastIncreasedTime + fs.minProfitTime ? 0 :
ts.minProfitBasisPoints[_indexToken];
        if (hasProfit && delta * Constants.PERCENTAGE_FACTOR <= _size *
minBps) {
            delta = 0;
        }
```

However, this min profit cap could cause unintended effects in some scenarios,

- A position could have a profit above the min-profit-threshold (i.e. resulting in a positive delta), but after increasing the position size to a certain amount could cause it to drop below min-profit-threshold (causing delta = 0). This is because min-profit-threshold is a % of the position size.
- nextAveragePrice() will be incorrect if the position is still within the min profit time as the profit will be zero and not considered into the nextAveragePrice().
- Profit is also not considered in the RealisedPnl() if the position is still within min profit time period. That will then affect getNextGlobalLongData().
- When only adding collateral with params.sizeDelta == 0, position.lastIncreaseTime is reset to current time. This puts the position in the min profit time period, which prevents the user from taking profit until it has passed. This is not necessary as the user is not increasing the position size.

Recommendation:

To resolve this, it is recommended to either remove the code for `minProfitTime` or set `minProfitBasisPoints` to zero.

As for the issue of frontrunning that the `minProfitTime` was supposed to mitigation, it could instead be resolved by setting at least 1 second delay between a user's intention to open a position and when the position is actually opened. This was implemented by gmx as proposed here https://gov.gmx.io/t/remove-the-min-price-movement-rule/157/18.

Gemnify: Recommendation implemented in commit

Zenith: Verified

[M-3] ExecuteSwap() is better to collect fee before applying priceImpact

Severity: Medium	Status: Acknowledged

Context:

- SwapLogic.sol#L93-L99

Description:

The base to collect fee in ExecuteSellUSDG()/ExecuteBuyUSDG() is the token amount without applying priceImapct. However in ExecuteSwap(), the _amountIn below is applied with priceImapct, i.e. the base to collect the fee in swap is the token amount with priceImapct applied.

```
        uint256 amountOut = (_amountIn * cache.priceInMin) /
cache.priceOutMax;

        amountOut = GenericLogic.adjustForDecimals(amountOut, _tokenIn,
_tokenOut);

        int256 feeBasisPoints = getSwapFeeBasisPoints(_tokenIn,
_tokenOut, usdgAmount);
        uint256 amountOutAfterFees =
GenericLogic.collectSwapFees(_tokenOut, amountOut, feeBasisPoints);
```

Since the tokens in priceImpactPool have been charged fee, this will cause these tokens to be charged fee repeatedly in ExecuteSwap().

Recommendation:

Consider charging fee before applying priceImpact in ExecuteSwap().

Gemnify: Acknowledged

Zenith: Will remain the same due to design.

[M-4] ExecuteSellUSDG() should also apply validateBufferAmount()

Severity: Medium	Status: Acknowledged

Context: ValidationLogic.sol#L55-L58

Description:

The protocol allows owner to set bufferAmounts to reserve available liquidity for leveraged positions.

```
    // bufferAmounts allows specification of an amount to exclude from
swaps
    // this can be used to ensure a certain amount of liquidity is
available for leverage positions
    mapping (address => uint256) public override bufferAmounts;
```

The bufferAmounts check requires poolAmounts >= bufferAmounts after the token swap, ensuring that no more tokens are swapped out. However, the problem here is that sellUSDG() also swaps tokens out, and users can execute buyUSDG() -> sellUSDG() to perform token swaps. Since there is no bufferAmounts check in sellUSDG(), it is possible that after the tokens are swapped out, poolAmounts < bufferAmounts, thus bypassing the bufferAmounts check.

Recommendation: Consider applying validateBufferAmount() in ExecuteSellUSDG

Gemnify: Acknowledged as a won't fix.

[M-5] Incorrect feeBP calculation in ExecuteSellUSDG

Severity: Medium	Status: Resolved

Context:

- SupplyLogic.sol#L132-L135
- GenericLogic.sol#L80-L105

Description:

getFeeBasisPoints() predicts the usdgAmount change direction and then determines the applied feeBPS.

The correct sequence is that the user deposits the asset, getFeeBasisPoints predicts the usdgAmount change direction and determines the feeBPS, and then changes the usdgAmount to apply the change. ExecuteBuyUSDG() and ExecuteSwap() both do this. However in ExecuteSellUSDG() it decreases the usdgAmount first, i.e. it applies the change first, and then calls getFeeBasisPoints to get the feeBPS.

For example, if usdgAmount[USDT] = 30000 and the user sells 10000 USDG for USDT, it should apply the feeBPS when usdgAmount goes from 30000 to 20000, but because of the early decrease of usdgAmount, it applies the feeBPS when usdgAmount goes from 20000 to 10000.

Recommendation: Consider getting feeBP first and then decreasing usdgAmount.

Gemnify: Fix implemented with the following commit

Zenith: The fix has been reviewed and approved.

[M-6] nextDiffUsdToken in SwapPriceImpactLogic._getPriceImpactUsd() is inaccurate

Severity: Medium	Status: Resolved

Context:

- SwapPriceImpactLogic.sol#L47-L75

Description:

When SwapPriceImpactLogic._getPriceImpactUsd() calculates nextDiffUsdToken, the actual targetAmountToken will not match the current targetAmountToken due to the minting of the new usdg, so getNextTargetUsdgAmount() should be used here to calculate the new targetAmountToken, that is, nextTargetAmountToken.

And in the judgment of isSameSideRebalance, nextTargetAmountToken should also be used to compare with nextPoolUsdForToken.

Also, note that when swapping, when swap tokenOut, targetAmountToken and nextTargetAmountToken need to be flipped.

Recommendation:

```
        uint256 targetAmountToken =
            (GenericLogic.getTargetUsdgAmount(token) *
Precision.FLOAT_PRECISION) / 10 ** Constants.USDG_DECIMALS;
+       uint256 nextTargetAmountToken =
GenericLogic.getNextTargetUsdgAmount(token, params.usdDelta.abs() / 1e12,
params.usdDelta > 0)
        uint256 initialDiffUsdToken =
Calc.diff(poolParams.poolUsdForToken, targetAmountToken);
-       uint256 nextDiffUsdToken =
Calc.diff(poolParams.nextPoolUsdForToken, targetAmountToken);
+       uint256 nextDiffUsdToken =
Calc.diff(poolParams.nextPoolUsdForToken, nextTargetAmountToken);

        // check whether an improvement in balance comes from causing the
balance to switch sides
        // for example, if there is $2000 of ETH and $1000 of USDC in the
pool
        // adding $1999 USDC into the pool will reduce absolute balance
from $1000 to $999 but it does not
        // help rebalance the pool much, the isSameSideRebalance value
helps avoid gaming using this case
```

```
        bool isSameSideRebalance =
-           (poolParams.poolUsdForToken <= targetAmountToken) ==
(poolParams.nextPoolUsdForToken <= targetAmountToken);
+           (poolParams.poolUsdForToken <= targetAmountToken) ==
(poolParams.nextPoolUsdForToken <= nextTargetAmountToken);
```

Gemnify: The fix has been implemented with the following commit

Zenith: Verified

[M-7] Incorrect calculation in `getRealisedPnl()` affects `ULP` price

Severity: Medium	Status: Resolved

Context:

- PositionLogic.sol#L931-L932

Description:

`PositionLogic.getRealisedPnl()` returns the realized profit/loss for the calculation of global long/short average price.

However, it fails to pass in the `isLong` parameter into `getDelta()` and instead use the `false` value even when it is for long positions.

This causes an incorrect calculation of the realized profit/loss for `poolInfo.globalLongAveragePrice`, which is used for `aum` calculation. The impact of this is that `ULP` price will be inaccurate as it is based on `aum` divided by `ULP` supply.

```
    function getRealisedPnl(
        address _account,
        address _collateralToken,
        address _indexToken,
        uint256 _sizeDelta,
        bool _isIncrease,
        bool _isLong
    ) internal view returns (int256) {
        if (_isIncrease) {
            return 0;
        }

        DataTypes.Position memory position = getPosition(_account,
_collateralToken, _indexToken, _isLong);

        (bool hasProfit, uint256 delta) =
>>>         getDelta(_indexToken, position.size, position.averagePrice,
false, position.lastIncreasedTime);
        // get the proportional change in pnl
        uint256 adjustedDelta = (_sizeDelta * delta) / position.size;

        return hasProfit ? int256(adjustedDelta) : -
int256(adjustedDelta);
    }
```

Recommendation: Pass in the `isLong` parameter for the call to `getDelta()` within `getRealisedPnl()`.

```
    function getRealisedPnl(
        address _account,
        address _collateralToken,
        address _indexToken,
        uint256 _sizeDelta,
        bool _isIncrease,
        bool _isLong
    ) internal view returns (int256) {
        if (_isIncrease) {
            return 0;
        }

        DataTypes.Position memory position = getPosition(_account,
_collateralToken, _indexToken, _isLong);

        (bool hasProfit, uint256 delta) =
-           getDelta(_indexToken, position.size, position.averagePrice,
false, position.lastIncreasedTime);
+           getDelta(_indexToken, position.size, position.averagePrice,
_isLong, position.lastIncreasedTime);
        // get the proportional change in pnl
        uint256 adjustedDelta = (_sizeDelta * delta) / position.size;

        return hasProfit ? int256(adjustedDelta) : -
int256(adjustedDelta);
    }
```

Gemnify: Recommendation implemented in commit

Zenith: Verified.

[M-8] Lack of slippage protection for position execution price after price impact

Severity: Medium	Status: Resolved

Context:

- PositionLogic.sol#L129-L142
- PositionLogic.sol#L641-L661
- BasePositionManager.sol#L80-L84
- BasePositionManager.sol#L102-L106
- OrderBook.sol#L505-L519

Description:

To provide slippage protection, `PositionRouter.sol` checks the `markPrice` against `request.acceptablePrice` to ensure that the execution price is not worse than the user's acceptable price. Similarly, `OrderBook.sol` also checks the `currentPrice` has passed the `triggerPrice`.

However, within `PositionLogic.sol`, both `increasePosition()` and `decreasePosition()` are subjected to price impact, which means that the execution price `cache.entryPrice` after price impact could be worse than what the user expected.

This could cause the user to open a position with unfavorable price, making an unexpected loss. Such a scenario could occur due to race condition or frontrunning that causes the user to encounter a significant negative price impact.

```
    function increasePosition(DataTypes.IncreasePositionParams memory
params) external {
        ...
        // get priceImpact
        if (GenericLogic.isPriceImpactEnabled(params.indexToken, false))
{
            (cache.priceImpactUsd, cache.priceImpactAmount,
cache.entryPrice) = getExecutionPriceForIncrease(
                params,
                GenericLogic.getMaxPrice(params.indexToken),
                GenericLogic.getMinPrice(params.indexToken),
                position.averagePrice,
                ps.reservedAmounts[params.indexToken]
            );

            // if there is a positive impact, the impact pool amount
should be reduced
```

```
            // if there is a negative impact, the impact pool amount
should be increased

PositionPriceImpactLogic.applyDeltaToPositionImpactPool(params.indexToken
, -cache.priceImpactAmount);
        }
```

Recommendation: This issue can be fixed by applying a slippage protection through a check of `cache.entryPrice` against `request.acceptablePrice` for both increasing and decreasing of positions.

Similarly, for limit orders using `OrderBook` it should add an `request.acceptablePrice` that allows the check against `cache.entryPrice`.

Gemnify: Price impact feature removed from implementation in commit

Zenith: Verified - resolved by removing price impact feature.

[M-9] `setFundingFactor()` fails to update funding state

Severity: Medium	Status: Resolved

Context:

- ConfigureLogic.sol#L149-L165

Description:

`ConfigureLogic.setFundingFactor()` should call `FundingFeeLogic.updateFundingState()` to update the funding state with the previous funding factor before updating it.

Otherwise this issue will cause the funding state update to use the incorrect factor for the period before funding factor update, causing users to overpay or underpay the funding fees.

```
    function setFundingFactor(
        address[] memory _tokens,
        uint256[] memory _fundingFactors,
        uint256[] memory _fundingExponentFactors
    ) external {
        require(
            _tokens.length == _fundingFactors.length && _tokens.length ==
_fundingExponentFactors.length,
            "inconsistent length"
        );

        DataTypes.FeeStorage storage fs =
StorageSlot.getVaultFeeStorage();

        for (uint256 i = 0; i < _tokens.length; i++) {
            fs.fundingFactors[_tokens[i]] = _fundingFactors[i];
            fs.fundingExponentFactors[_tokens[i]] =
_fundingExponentFactors[i];
        }
    }
```

Recommendation: Call `FundingFeeLogic.updateFundingState(_tokens[i])` before updating the factors as follows:

```
    function setFundingFactor(
        address[] memory _tokens,
        uint256[] memory _fundingFactors,
```

```
        uint256[] memory _fundingExponentFactors
    ) external {
        require(
            _tokens.length == _fundingFactors.length && _tokens.length ==
_fundingExponentFactors.length,
            "inconsistent length"
        );

        DataTypes.FeeStorage storage fs =
StorageSlot.getVaultFeeStorage();

        for (uint256 i = 0; i < _tokens.length; i++) {
+            FundingFeeLogic.updateFundingState(_tokens[i]);
            fs.fundingFactors[_tokens[i]] = _fundingFactors[i];
            fs.fundingExponentFactors[_tokens[i]] =
_fundingExponentFactors[i];
        }
    }
```

Gemnify: Recommendation implemented with the following commit

Zenith: Verified.

[M-10] Protocol fails to update cumulative borrowing rate in certain scenarios

Severity: Medium	Status: Resolved

Context:

- ConfigureLogic.sol#L33-L52
- SupplyLogic.sol#L165-L172
- BorrowingFeeLogic.sol#L51-L71

Description:

When borrowing rate changes due to new settings or new values, it should trigger `updateCumulativeBorrowingRate()` to update the borrowing rate accumulator based on the previous borrowing rate first.

Otherwise, it would cause users to be overcharged or undercharged as the borrowing rate will be updated with an incorrect rate for the previous period.

The issue present in the following code, where the borrowing rate update are missing,

1. `ConfigureLogic.setBorrowingRate()` will set the new borrowing rate but it fails to call `updateCumulativeBorrowingRate()`.

```
    function setBorrowingRate(
        uint256 _borrowingInterval,
        uint256 _borrowingRateFactor,
        uint256 _stableBorrowingRateFactor
    ) external {
        DataTypes.FeeStorage storage fs =
StorageSlot.getVaultFeeStorage();
        ValidationLogic.validate(
            _borrowingInterval >= Constants.MIN_BORROWING_RATE_INTERVAL,
Errors.VAULT_INVALID_BORROWING_INTERVALE
        );
        ValidationLogic.validate(
            _borrowingRateFactor <= Constants.MAX_BORROWING_RATE_FACTOR,
Errors.VAULT_INVALID_BORROWING_RATE_FACTOR
        );
        ValidationLogic.validate(
            _stableBorrowingRateFactor <=
Constants.MAX_BORROWING_RATE_FACTOR,
            Errors.VAULT_INVALID_STABLE_BORROWING_RATE_FACTOR
        );
        fs.borrowingInterval = _borrowingInterval;
        fs.borrowingRateFactor = _borrowingRateFactor;
```

```
        fs.stableBorrowingRateFactor = _stableBorrowingRateFactor;
    }
```

2. `SupplyLogic.directPoolDeposit()`will need to call `updateCumulativeBorrowingRate()` since it will increase pool amount, which is used for determining the `aum`, a component of the borrowing rate calculatin.

```
    function directPoolDeposit(address _token) external {
        DataTypes.TokenConfigSotrage storage ts =
StorageSlot.getVaultTokenConfigStorage();
        ValidationLogic.validate(ts.whitelistedTokens[_token],
Errors.VAULT_TOKEN_NOT_WHITELISTED);
        uint256 tokenAmount = GenericLogic.transferIn(_token);
        ValidationLogic.validate(tokenAmount > 0,
Errors.VAULT_INVALID_TOKEN_AMOUNT);
        GenericLogic.increasePoolAmount(_token, tokenAmount);
        emit DirectPoolDeposit(_token, tokenAmount);
    }
```

3. Any changes to `aum` should also trigger `updateCumulativeBorrowingRate()`, as it will affect the borrowing rate.

```
   function getNextBorrowingRate(address _token) internal view returns
(uint256) {
       DataTypes.FeeStorage storage fs =
StorageSlot.getVaultFeeStorage();
       DataTypes.PositionStorage storage ps =
StorageSlot.getVaultPositionStorage();
       DataTypes.TokenConfigSotrage storage ts =
StorageSlot.getVaultTokenConfigStorage();
       DataTypes.AddressStorage storage addrs =
StorageSlot.getVaultAddressStorage();
       if (fs.lastBorrowingTimes[_token] + fs.borrowingInterval >
block.timestamp) {
           return 0;
       }

       uint256 intervals = (block.timestamp -
fs.lastBorrowingTimes[_token]) / (fs.borrowingInterval);

       uint256 _borrowingRateFactor = ts.stableTokens[_token] ?
fs.stableBorrowingRateFactor : fs.borrowingRateFactor;
```

```
        uint256 aum = IUlpManager(addrs.ulpManager).getAum(true);

        uint256 price = GenericLogic.getMinPrice(_token);
        uint256 decimals = ts.tokenDecimals[_token];
        uint256 reservedUsd = (price * ps.reservedAmounts[_token]) / 10 **
decimals;

        return (_borrowingRateFactor * reservedUsd * intervals) / aum;
    }
```

Recommendation:

1. Call `BorrowingFeeLogic.updateCumulativeBorrowingRate()` to update the borrowing rate accumulator based on the previous borrowing rate, before updating with the new borrowing rate.
2. Call `updateCumulativeBorrowingRate()` within `directPoolDeposit()`.
3. To continue to have aum for OI limit, you will have to settle for an approximate solution, since price for aum is controlled by external factor.

- For example, you can have a bot to call `updateCumulativeBorrowingRate()` periodically, where the borrowing rate is updated based on latest aum. The more frequent you call it, the more accurate it will be (e.g. every 1 sec vs every 1 hr).

Gemnify: Recommendation implemented with the following commit

Zenith: Verified

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] USDG minted in ExecuteBuyUSDG() and the return value may be different, which will make slippage control invalid.

Severity: Low	Status: Acknowledged

Context:

- SupplyLogic.sol#L88-L103

Description:

The return value of ExecuteBuyUSDG() is not the actual usdg amount minted, it returns the usdg amount suitable for ulpManager to use for minting to the user.

```
        uint256 usdgAmount = vault.buyUSDG(_token, address(this));
        require(usdgAmount >= _minUsdg, "UlpManager: insufficient USDG
output");
        uint256 mintAmount;

        if (aumInUsdg != 0 && ulpSupply != 0) {
            mintAmount = (usdgAmount * ulpSupply) / aumInUsdg;
        } else {
            mintAmount = usdgAmount;
        }
        require(mintAmount >= _minUlp, "UlpManager: insufficient ULP
output");
```

However, in Router, the return value should represent the actual usdg amount minted to the user, otherwise the Router's slippage control will invalid.

```
    function _vaultSwap(address _tokenIn, address _tokenOut, uint256
_minOut, address _receiver)
        private
        returns (uint256)
    {
        uint256 amountOut;

        if (_tokenOut == usdg) {
            // buyUSDG
```

```
            amountOut = IVault(vault).buyUSDG(_tokenIn, _receiver);
        ...

        require(amountOut >= _minOut, "Router: insufficient amountOut");
        return amountOut;
```

Recommendation:

It is recommended to return two values so that the actual usdg amount minted is the second one and check it in the Router.

Gemnify: It is recommended to return two values so that the actual usdg amount minted is the second one and check it in the Router.

Zenith: Acknowledged - no fix required here due to design.

[L-2] getSupplyPriceImpactUsd() is better to use priceMid instead of priceMax

Severity: Low	Status: Acknowledged

Context:

- SupplyLogic.sol#L70-L76
- SupplyLogic.sol#L121-L127
- SupplyLogic.sol#L193-L195
- SwapLogic.sol#L70-L75
- SwapLogic.sol#L102-L108
- SwapLogic.sol#L183-L198

Description:

In getSupplyPriceImpactUsd(), params.price is used to calculate poolUsd and nextPoolUsd, which are used to predict the direction of token movement and determine the value of priceImpact.

The problem here is that priceMax is always used when calling getSupplyPriceImpactUsd, which actually overestimates the value of tokens in the pool, making the prediction inaccurate.

We can consider using (priceMax+priceMin)/2 as params.price. (Also, priceMax is used in PositionPriceImpact, but since priceMax is not currently involved in the calculation, it is fine.

Recommendation:

Consider using (priceMax+priceMin)/2 as params.price when calling getSupplyPriceImpactUsd().

Gemnify: Fix implemented with the following commit

Zenith: Verified

[L-3] `getNextFundingAmountPerSize()` calls `getFundingAmountPerSizeDelta()` with incorrect parameters

Severity: Low	Status: Resolved

Context:

- FundingFeeLogic.sol#L118-L123

Description:

In `FundingFeeLogic.sol`, the function `getNextFundingAmountPerSize()` is used by `updateFundingState()` to calculate the funding fee amount.

However, the parameters `shortTokenPrice` and `cache.longOpenInterest` are incorrectly passed in the wrong order for Long funding fee.

```
    function getNextFundingAmountPerSize(address _indexToken)
        internal
        view
        returns (GetNextFundingAmountPerSizeResult memory)
    {
        ...
        if (result.longsPayShorts) {
            result.fundingFeeAmountPerSizeDelta.long =
getFundingAmountPerSizeDelta(
                cache.fundingUsd,
>>>             shortTokenPrice,
>>>             cache.longOpenInterest,
                true // roundUpMagnitude
            );
```

This issue only causes a wrong order of the computation in `getFundingAmountPerSizeDelta()` that still derive the same result, as both `shortTokenPrice` and `cache.longOpenInterest` are used as denominators as shown below. Despite that, it is still recommended to fix this issue to prevent future bugs.

```
    function getFundingAmountPerSizeDelta(
        uint256 fundingUsd,
        uint256 openInterest,
        uint256 tokenPrice,
        bool roundUpMagnitude
    ) internal pure returns (uint256) {
```

```
        if (fundingUsd == 0 || openInterest == 0) {
            return 0;
        }
        uint256 fundingUsdPerSize = Precision.mulDiv(
            fundingUsd, Precision.FLOAT_PRECISION *
Precision.FLOAT_PRECISION_SQRT, openInterest, roundUpMagnitude
        );

        if (roundUpMagnitude) {
            return Calc.roundUpDivision(fundingUsdPerSize *
Constants.COllATERAL_PRECISION, tokenPrice);
        } else {
            return (fundingUsdPerSize * Constants.COllATERAL_PRECISION) /
tokenPrice;
        }
    }
```

Recommendation: For `getNextFundingAmountPerSize()`, update the parameters for the call to `getFundingAmountPerSizeDelta()` as follows:

```
        if (result.longsPayShorts) {
            result.fundingFeeAmountPerSizeDelta.long =
getFundingAmountPerSizeDelta(
                cache.fundingUsd,
-               shortTokenPrice,
-               cache.longOpenInterest,
+               cache.longOpenInterest,
+               shortTokenPrice,
                true // roundUpMagnitude
            );
```

Gemnify: Recommendation implemented in the following commit

Zenith: Verified

4.4 Informational

A total of 2 informational findings were identified.

[I-1] Some checks in validateLiquidation() that could be improved

Severity: Informational	Status: Resolved

Context:

- PositionLogic.sol#L514-L574

Description: validateLiquidation() is used to check whether a position can be liquidated. Some of the checks can be improved.

Recommendation:

1. For bad debts (collateral is insufficient to cover the loss), the collateral should cover losses first, i.e., let marginFees be 0.

```
        if (!hasProfit && position.collateral < delta) {
            if (_raise) {
                revert("Vault: losses exceed collateral");
            }
-           return (1, marginFees);
+           return (1, 0);
        }
```

2. remainingCollateral should take into account pending profits, thus avoiding users being liquidated due to insufficient collateral (but profitable positions)

```
        uint256 remainingCollateral = position.collateral;
        if (!hasProfit) {
            remainingCollateral = position.collateral - delta;
-       }
+       } else {
+           remainingCollateral = position.collateral + delta;
+       }
```

Gemnify: Fixed with the following commit

Zenith: Verified 1 and 2 will remain the same due to design.

[I-2] validateTokens() should add additional validation

Severity: Informational	Status: Resolved

Context:

- ValidationLogic.sol#L67-L76

Description:

validateTokens() should add two additional validations:

1. _collateralToken must be USDC
2. When long, _indexToken must be a whitelisted token

Recommendation:

```
    function validateTokens(address _collateralToken, address
_indexToken, bool _isLong) internal view {
        DataTypes.TokenConfigSotrage storage ts =
StorageSlot.getVaultTokenConfigStorage();
+       validate(_collateralToken == addrs.collateralToken,
Errors.VAULT_COLLATERAL_TOKEN_MUST_BE_STABLE_TOKEN);
        validate(ts.whitelistedTokens[_collateralToken],
Errors.VAULT_COLLATERAL_TOKEN_NOT_WHITELISTED);
        validate(ts.stableTokens[_collateralToken],
Errors.VAULT_COLLATERAL_TOKEN_MUST_BE_STABLE_TOKEN);
        validate(!ts.stableTokens[_indexToken],
Errors.VAULT_INDEX_TOKEN_MUST_NOT_BE_STABLE_TOKEN);
        if (!_isLong) {
            validate(ts.shortableTokens[_indexToken],
Errors.VAULT_INDEX_TOKEN_NOT_SHORTABLE);
-       }
+       } else {
+           validate(ts.whitelistedTokens[_indexToken],
Errors.VAULT_INDEX_TOKEN_NOT_WHITELISTED);
+       }
    }
```

Gemnify: Fixed with the following commit

Zenith: Verified