GammaSwap Vault Security Review

Pashov Audit Group

Conducted by: unforgiven, ast3ros, Hals

December 30th 2024 - January 10th 2025

Contents

1. About Pashov Audit Group

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

2. Disclaimer

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

3. Introduction

A time-boxed security review of the **gammaswap/v1-vaults** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About GammaSwap Vault

V1-Vaults by GammaSwap provide delta-neutral liquidity strategies for Uniswap V3 pools by hedging positions with liquidity loans on GammaSwap. Users deposit funds into vaults, represented by ERC-20 tokens, while a manager oversees rebalancing, deposits, and withdrawals to maintain optimal liquidity. Profits or losses from the hedge are dynamically adjusted and reinvested, ensuring stability in the value of the designated token. The system uses separate funding contracts to streamline user interactions and minimize gas costs.

5. Risk Classification

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

5.1. Impact

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

5.2. Likelihood

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

5.3. Action required for severity levels

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

6. Security Assessment Summary

review commit hash - 08f90c7e9687586b016e6ccc7c50d4dee4ef295a

fixes review commit hash - dbfbc9c288147c36f784446c838a174a30d245c0

Scope

The following smart contracts were in scope of the audit:

- `FundingVaultERC20`
- `GammaVaultERC20`
- `FundingVault`
- `GSHedgeMath`
- `RangedPoolMath`
- `VaultStorage`
- `VaultCollateralManager`
- `VaultERC165Storage`
- `VaultLoanObserver`
- `BaseProcess`
- `DepositProcess`
- `HedgeProcess`
- `RebalanceProcess`
- `WithdrawProcess`
- `VaultAppStorage`
- `VaultRebalancer`
- `GammaVault`

7. Executive Summary

Over the course of the security review, unforgiven, ast3ros, Hals engaged with GammaSwap to review GammaSwap Vault. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	GammaSwap Vault
Repository	https://github.com/gammaswap/v1-vaults
Date	December 30th 2024 - January 10th 2025
Protocol Type	Delta-neutral liquidity management

Findings Count

Severity	**Amount**
Critical	1
High	2
Medium	5
Low	9
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[C-01]	disburseClaims() allows theft of unclaimed funds	Critical	Resolved
[H-01]	Incorrect swapping path in processWithdrawals()	High	Resolved
[H-02]	No slippage protection when interacting with AMM	High	Resolved
[M-01]	Uncapped funding period deposits could lead to DoS	Medium	Resolved
[M-02]	rebalancePosition() transaction is susceptible to frontrunning	Medium	Resolved
[M-03]	processWithdrawals() incorrectly updates of totalLiquidity	Medium	Resolved
[M-04]	Price manipulation risk in GammaVault collateral calculation	Medium	Resolved
[M-05]	Deposits will fail when pool's price reaches the min/max tick	Medium	Resolved
[L-01]	Signature malleability due to missing ECDSA checks	Low	Resolved
[L-02]	Potential precision loss when decoding square root prices for low price values	Low	Resolved
[L-03]	Withdrawn hedge profit is not reinvested into Uniswap V3 position	Low	Resolved
[L-04]	Rounding down in the shares calculation in favor of the user	Low	Resolved
[L-05]	Stale price ratios used when calculating deposit amounts after swaps	Low	Resolved

[L-06]	Missing transaction deadline validation	Low	Resolved
[L-07]	Using the wrong value when calling calcUpdatedCollateral()	Low	Resolved
[L-08]	Incorrect missingAmounts check	Low	Resolved
[L-09]	Small dust thresholds in rebalancing loop can cause token loss	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] `disburseClaims()` allows theft of unclaimed funds

Severity

Impact: High

Likelihood: High

Description

The FundingVault's `disburseClaims` function incorrectly assigns all current contract token balances to the current period, without accounting for unclaimed rewards from previous periods. This allows funds allocated to previous periods to be claimed by users in the current period.

We can see that the FundingVault enables users to make claims for the previous period. The claimable amount is determined by the lesser value between `_claimable` and the current balance of the claimToken.

```
function claim(uint256 _period) external override virtual lock {
        ...
        _claimable = GSMath.min(GammaSwapLibrary.balanceOf(claimToken, address
          (this)), _claimable);
        ...
        GammaSwapLibrary.safeTransfer(claimToken, _to, _claimable);
```

However, `disburseClaims` fails to track unclaimed amounts from previous periods. And it assigns the entire token balance to the current period without reservation for pending claims.

```
function disburseClaims() external override virtual isManager {
        ...
        totalClaimable[_period] = GammaSwapLibrary.balanceOf
        //(claimToken, address(this)); // @audit Incorrectly assigns all current balan
        ...
    }
```

Scenario:

- In period 2, Alice is eligible to claim 1000 tokens but hasn't claimed yet
- Period 3 begins, and new claim amount worth 500 tokens are added
- `disburseClaims` is called, setting totalClaimable[3] = 1500 (1000 + 500)
- Bob, who is eligible for only 100 tokens in period 3, claims first and receives more than his fair share from the total 1500 tokens
- When Alice tries to claim her 1000 tokens from period 2, insufficient funds remain. She may receive 0 if all the tokens are claimed in period 3.

Recommendations

Tracking separately `totalClaimable` in each period by parameters in `disburseClaims`:

```
- function disburseClaims() external override virtual isManager {
+ function disburseClaims
+ (uint256 newClaimable) external override virtual isManager {
    ...
-    totalClaimable[_period] = GammaSwapLibrary.balanceOf(claimToken, address
- (this));
+    totalClaimable[_period] = newClaimable;
    ...
}
```

8.2. High Findings

[H-01] Incorrect swapping path in `processWithdrawals()`

Severity

Impact: Medium

Likelihood: High

Description

When the `GammaVault` manager calls the `processWithdrawals()` , the `withdrawVault` transfers its `GammaVault` token holdings to the `GammaVault`, then the `GammaVault` calculates the corresponding liquidity to withdraw from Uniswap V3, adjusts the GammaSwap hedge proportionally to maintain the hedge ratio, and converts all funds (including any profits or losses) back into the `assetToken` , then these resulting funds are transferred to the `withdrawVault`.

The `withdrawVault` is expected to receive the `assetToken` (since the protocol currently supports only the `WETH/USDC` pair, the `assetToken` is `WETH`) and a swap is performed to convert the other token pair into the `assetToken` via the Uniswap router.

The `baseBalance` represents the other token pair (`USDC`), but the swap path is always set to `path1`, regardless of whether the base token is `token0` or `token1`:

```
function processWithdrawals(
        bytes memory path0,
        bytes memory path1
    ) external virtual override {
        //...
        uint256 baseBalance = GammaSwapLibrary.balanceOf(
                    isAssetToken0 ? token1 : token0,
                    address(this)
                );
        if (baseBalance > 0) {
                    IUniversalRouter(router).swapExactTokensForTokens(
                        baseBalance,
                        0,
                        path1, // @audit :  path is incorrect
                        address(this),
                        type(uint256).max
                    );
                    uint256 assetBalance = GammaSwapLibrary.balanceOf(
                        assetToken,
                        address(this)
                    );
                    GammaSwapLibrary.safeTransfer(
                        assetToken,
                        s.withdrawVault,
                        assetBalance
                    );
                }
        //...
    }
```

Since the protocol currently only supports `WETH/USDC`, where `WETH` is the `assetToken`, and the protocol is going to be initially deployed on Arbitrum; where `token1` will be USDC (as the address of USDC > address of WETH on Arbitrum and Base), so the `baseToken` path here will be correct (`path1`) and will not lead to any issues.

However, when the protocol is deployed on Ethereum, where `WETH` address > `USDC` address (`WETH` address: `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2`, `USDC` address `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48`), then `token0` will be `USDC` (`baseToken`), so the used path to sell `baseToken` will be incorrect, resulting in the swap silently fails, and the `withdrawVault` receiving less `assetToken` (`WETH`), which will negatively impact user claims.

This issue becomes more significant when other tokens pairs are adopted by the protocol, where swapping path to sell base token will not be always `path1`.

Recommendations

Update the code to dynamically select the correct path based on whether the `assetToken` is `token0` or `token1`:

```
function processWithdrawals(
        bytes memory path0,
        bytes memory path1
    ) external virtual override {
        //...
        uint256 baseBalance = GammaSwapLibrary.balanceOf(
                    isAssetToken0 ? token1 : token0,
                    address(this)
                );
        if (baseBalance > 0) {
                    IUniversalRouter(router).swapExactTokensForTokens(
                        baseBalance,
                        0,
-                       path1,
+                       isAssetToken0 ? path1 : path0,
                        address(this),
                        type(uint256).max
                    );
                    uint256 assetBalance = GammaSwapLibrary.balanceOf(
                        assetToken,
                        address(this)
                    );
                    GammaSwapLibrary.safeTransfer(
                        assetToken,
                        s.withdrawVault,
                        assetBalance
                    );
                }
        //...
    }
```

[H-02] No slippage protection when interacting with AMM

Severity

Impact: High

Likelihood: Medium

Description

- It was noticed that the `amount0Min` & `amount1Min` are set to zero during interactions with the AMM router (Uniswap) for swaps and liquidity additions. These parameters are intended to protect against slippage; setting them to zero exposes transactions to slippage, potentially resulting in receiving or adding smaller amounts than intended.

- This issue was identified across all interactions with the AMM (Uniswap), when swapping, adding liquidity and removing liquidity, where these different interactions has different impacts on the protocol:

_depositAsset() _increaseLiquidity() _addMissingAmountsToHedge() _decreaseLiquidity() _depositAsset() processWithdrawals() rebalanceCollateralToRatio()

Recommendations

Instead of hard-coding the minimum amounts to zero, introduce a state variable to control the slippage percentage (e.g., 80% of the swapped or added amounts, with a setter to update the slippage during high market volatility) and utilize it when interacting with the AMM.

Or, it's recommended to set a minLiquidity received and revert if the total liquidity received of the whole transaction is less than minLiquidity. The minLiquidity can be determined by simulation using current (no manipulated) price.

8.3. Medium Findings

[M-01] Uncapped funding period deposits could lead to DoS

Severity

Impact: Medium

Likelihood: Medium

Description

The `depositVault` allows unlimited deposits per funding period without any cap, which can lead to a DoS scenario during the execution of `DepositProcess.processDeposit()`, if too much liquidity is deposited into the `depositVault` during a single period, it can result in the vault attempting to hedge a larger position than what the GammaSwap pool can support, causing the `IGammaPool(gsPool).borrowLiquidity()` call to revert.

```
function _increaseHedge(
        uint256 _liquidity,
        bytes memory path0,
        bytes memory path1
    ) internal virtual {
     //...
       uint256 _hedgeLpTokens = GSHedgeMath.convertLiquidityToLPTokens(
            cfmm,
            _liquidity
        );

         IGammaPool(gsPool).borrowLiquidity(
            _gsTokenId,
            _hedgeLpTokens,
            new uint256[](0)
        ); // use externalRebalancing here or hedge with internal AMM

        _updateHedge(path0, path1, s.ratio0, s.ratio1);
    }
```

The `_hedgeLpTokens` amount is calculated using the ratio between the Uniswap pool's total liquidity and LP token supply:

```
function convertLiquidityToLPTokens(
        address _cfmm,
        uint256 _liquidity
    ) internal view returns (uint256) {
        (uint128 reserve0, uint128 reserve1, ) = ICPMM(_cfmm).getReserves();
        uint256 lastCFMMInvariant = GSMath.sqrt(reserve0 * reserve1);
        uint256 lastCFMMTotalSupply = GammaSwapLibrary.totalSupply(_cfmm);
        return (_liquidity * lastCFMMTotalSupply) / lastCFMMInvariant;
    }
```

and the `IGammaPool(gsPool).borrowLiquidity()` function will revert if the requested LP tokens exceed the pool's available balance (`s.LP_TOKEN_BALANCE`):

```
function borrowLiquidity(
  uint256tokenId,
  uint256lpTokens,
  uint256[]calldataratio
) external virtual override whenNotPaused(12
        return abi.decode(callStrategy(borrowStrategy, abi.encodeCall(
          callStrategy
        )
 }

//...
/// @dev See {IBorrowStrategy-_borrowLiquidity}.
 function _borrowLiquidity(
   uint256tokenId,
   uint256lpTokens,
   uint256[]calldataratio
 ) external virtual override lock returns(uint256 liquidityBorrowed, uint256[] memory
        // Revert if borrowing all CFMM LP tokens in pool
        if(lpTokens >= s.LP_TOKEN_BALANCE) revert ExcessiveBorrowing();
    //...
 }
```

Exceeding the GammaSwap pool's `s.LP_TOKEN_BALANCE` could be achieved in two ways:

1. The `processDeposit()` transaction can be frontrun by a malicious actor (as the protocol will be deployed on Ethereum) by depositing a large ETH amount (by a flash-loan), so that the required lp tokens needed to borrow to increase the hedge would be inflated to exceed the GammaSwap pool balance `s.LP_TOKEN_BALANCE`.
2. If users naturally deposit large amounts during the same funding period.

Recommendations

- Implement a deposit cap per period where the `depositVault` can't receive any further deposits from users when it's reached.
- Update the `convertLiquidityToLPTokens()` calculation to ensure the requested LP tokens (`_hedgeLpTokens`) do not exceed the GammaSwap pool's `s.LP_TOKEN_BALANCE`.

[M-02] `rebalancePosition()` transaction is susceptible to frontrunning

Severity

Impact: Medium

Likelihood: Medium

Description

Position collaterals are rebalanced to align with a desired ratio specified by the `GammaVault` manager when the manager calls `rebalancePosition`. The steps performed during the rebalancing process include:

1. Collecting fees from the Uniswap V3 position.
2. Decreasing liquidity to zero.
3. Closing the position.
4. Opening a new position with a revised hedge.

```
function rebalancePosition(
        IGammaVault.RebalancePositionData memory params
    ) external virtual override {
        _validatePaths(params.path0, params.path1);

        _collectFees(s.tokenId, address(this));
        _decreaseLiquidity(
            s.tokenId,
            s.totalLiquidity,
            params.amount0Min,
            params.amount1Min
        );
        _closeLP(s.tokenId);

        //...
    }
```

```
function _closeLP(uint256 _tokenId) internal {
        if (_tokenId > 0) {
            IUniswapV3PositionManager(nftPosMgr).burn(_tokenId);
            s.tokenId = 0;
        }
    }
```

The `IUniswapV3PositionManager(nftPosMgr).burn()` call requires the position to have zero liquidity; otherwise, the transaction will revert:

```
function burn
     (uint256 tokenId) external payable override isAuthorizedForToken(tokenId) {
        Position storage position = _positions[tokenId];
        require(
          position.liquidity==0&&position.tokensOwed0==0&&position.tokensOwed1==0,
          'Notcleared'
        );
        delete _positions[tokenId];
        _burn(tokenId);
    }
```

The `s.totalLiquidity` value is expected to be the total liquidity of the position, required for closing the position after withdrawal, however, using this cached value exposes the `rebalancePosition()` transaction to frontrunning, where a malicious user can frontrun the transaction by calling `NonfungiblePositionManager.increaseLiquidity()` and adding 1 wei of liquidity to the position.

This would cause the `burn()` call to revert, preventing the position from being closed since liquidity would no longer be zero.

Recommendations

Fetch the position liquidity directly from the Uniswap V3 position manager inside the `rebalancePosition()` :

```
function rebalancePosition(
        IGammaVault.RebalancePositionData memory params
    ) external virtual override {
        _validatePaths(params.path0, params.path1);
+   uint256 _totalLiquidity = _getTotalLiquidity(_tokenId);
        _collectFees(s.tokenId, address(this));
        _decreaseLiquidity(
            s.tokenId,
-           s.totalLiquidity,
+         _totalLiquidity,
            params.amount0Min,
            params.amount1Min
        );
        _closeLP(s.tokenId);

        //...
    }
```

[M-03] `processWithdrawals()` incorrectly updates of `totalLiquidity`

Severity

Impact: Medium

Likelihood: Medium

Description

- In `processWithdrawals()` function: when the `_withdrawShares = 0`, the `s.totalLiquidity` is increased by the liquidity of the collected-and-deposited fees of the position, where it assumes that the totalLiquidity of the position will not be decreased when `_reviseHedge()` is called:

```
function processWithdrawals(
        bytes memory path0,
        bytes memory path1
    ) external virtual override {
        //...
        uint256 _liquidity = _depositFees(_tokenId, path0, path1);
        _totalLiquidity += _liquidity;

        if (_withdrawShares == 0) {
            // if no withdrawals we only compound fees
            if (_liquidity > 0) {
                _reviseHedge
                //(_totalLiquidity, path0, path1); // liquidity only goes up here
            }
             s.totalLiquidity = _totalLiquidity;
            FundingVault(s.withdrawVault).disburseClaims();
            return;
        }

        //...
    }
```

Where:

```
function _reviseHedge(
        uint256 _totalLiquidity,
        bytes memory path0,
        bytes memory path1
    ) internal virtual {
        uint256 _newHedgeLiquidity = (_totalLiquidity * s.hedgeSize) / 1e18;

        uint256 _prevHedgeLiquidity = _loanData.liquidity;

        if (_newHedgeLiquidity > _prevHedgeLiquidity) {
            _increaseHedge(
                _newHedgeLiquidity - _prevHedgeLiquidity,
                path0,
                path1
            );
        } else if (_newHedgeLiquidity < _prevHedgeLiquidity) {
            _decreaseHedge(
                _prevHedgeLiquidity - _newHedgeLiquidity,
                path0,
                path1
            );
        }
    }
```

- Given that there's a setter for the `hedgeSize` (where it's used to calculate the `_newHedgeLiquidity`), then the `s.totalLiquidity` should be updated based on the liquidity position in case the `hedgeSize` was updated (also, in `_reviseHedge()`, the hedge liquidity is fetched based on the last updated data without checking if the hedge has accrued interest over time).

- So if the new `hedgeSize` is decreased and results in `_newHedgeLiquidity < _prevHedgeLiquidity` , then the hedge will be decreased, which might result in the hedge incurring losses, hence affecting the liquidity, which will affect the next process that's going to be executed after the withdrawal: if the next process is a deposit, then this incorrect `s.totalLiquidity` will result in minting wrong shares for that deposited period as the liquidity doesn't reflect the actual one (since it's the cashed one + liquidity from the collected-and-deposited fees of the LP position).

- Same issue in `DepositProcess.processDeposits()` function.

Recommendations

```
function processWithdrawals(
        bytes memory path0,
        bytes memory path1
    ) external virtual override {
        //...
        uint256 _liquidity = _depositFees(_tokenId, path0, path1);
        _totalLiquidity += _liquidity;

        if (_withdrawShares == 0) {
            // if no withdrawals we only compound fees
            if (_liquidity > 0) {
                _reviseHedge
                //(_totalLiquidity, path0, path1); // liquidity only goes up here
            }
-            s.totalLiquidity = _totalLiquidity;
+            s.totalLiquidity = _getTotalLiquidity(s.tokenId);
            FundingVault(s.withdrawVault).disburseClaims();
            return;
        }

        //...
    }
```

[M-04] Price manipulation risk in GammaVault collateral calculation

Severity

Impact: Medium

Likelihood: Medium

Description

The GammaVault contract calculates external collateral for GammaPool positions using the current spot price from Uniswap V3, rather than a more manipulation-resistant price source. This creates a potential vulnerability where an attacker could manipulate the spot price to affect collateral calculations and force unfair liquidations.

```
function _getCollateral(
      address_gammaPool,
      uint256_tokenId
    ) internal override virtual view returns(uint256 collateral
        uint160 sqrtPriceX96 = getCurrentPrice
        //(); // @audit collateral is depent on spot price
        uint160 sqrtPriceAX96 = RangedPoolMath.calcSqrtRatioAtTick(s.tickLower);
        uint160 sqrtPriceBX96 = RangedPoolMath.calcSqrtRatioAtTick(s.tickUpper);

        uint128[] memory tokensHeld = new uint128[](2);
        {
            (
              uint256amount0,
              uint256amount1
            ) = RangedPoolMath.getAmountsForLiquidity(sqrtPriceX96, sqrtPriceAX96,
            sqrtPriceBX96, uint128(s.totalLiquidity));
            tokensHeld[0] = uint128(amount0);
            tokensHeld[1] = uint128(amount1);
        }

        collateral = GSHedgeMath.convertAmountsToLiquidity
          (cfmm, tokensHeld, numOfDecimals0, numOfDecimals1, mathLib);
    }
```

Let's see an attack scenario:

- Attacker identifies a GammaVault position with: - Tight tick range in UniswapV3 (high concentration of liquidity) - High hedge ratio - Collateral value close to liquidation threshold
- Attacker executes a large swap to manipulate the UniswapV3 spot price
- This manipulation temporarily reduces the calculated collateral value via `_getCollateral`
- Attacker triggers liquidation of the hedge position
- Attacker profits by claiming the internal collateral (tokensHeld) at a discount

Recommendations

The collateral should use TWAP price instead of spot price to calculate the external collateral.

[M-05] Deposits will fail when pool's price reaches the min/max tick

Severity

Impact: Medium

Likelihood: Medium

Description

During the deposit, code calls `calculateTokenRatio()` to calculate the `ratio0` and `ratio1`. If the pool's prices reaches the upper or lower price then `ratio0` or `ratio1` would be 0:

```
} else if (price <= priceLower) {
            ratio0 = 0;
            ratio1 = decimals1;
        } else if (price >= priceUpper) {
            ratio0 = decimals0;
            ratio1 = 0;
        }
```

Later, code uses the `ratio0` and `ratio1` to calculate the max deposit amount by calling `calculateMaxDeposit()`. The issue is that if `ratio0` or `ratio1` were 0 then function `calculateMaxDeposit()` can revert because of division by zero:

```
function calculateMaxDeposit(
      boolisToken0,
      uint256amount0,
      uint256amount1,
      uint256ratio0,
      uint256ratio1
    )
        internal view returns(uint256 deposit0, uint256 deposit1) {
        if(isToken0) {
            deposit0 = amount0;
            deposit1 = ratio1 * amount0 / ratio0;
            if(deposit1 > amount1) {
                deposit0 = ratio0 * amount1 / ratio1;
                deposit1 = ratio1 * deposit0 / ratio0;
            }
        } else {
            deposit1 = amount1;
            deposit0 = ratio0 * amount1 / ratio1;
            if(deposit0 > amount0) {
                deposit1 = ratio1 * amount0 / ratio0;
                deposit0 = ratio0 * deposit1 / ratio1;
            }
        }
    }
```

As result when price is equal to min or max tick the deposit would revert always.

Recommendations

Handle the case when ratio0 or raio1 is 0 in the `calculateMaxDeposit()`

8.4. Low Findings

[L-01] Signature malleability due to missing ECDSA checks

The `permit` function in FundingVault20 uses `ecrecover` to validate signatures without implementing checks for signature malleability. It has no validation that `s` is in the lower half of the curve's order.

```
function permit(
      addressowner,
      addressspender,
      uint256value,
      uint256deadline,
      uint8_v,
      bytes32_r,
      bytes32_s
    )
        external
        override
    {
        ...
        address recoveredAddress = ecrecover
        //(digest, _v, _r, _s); // @audit s isn't verified
        ...
    }
```

While the nonce mechanism (`nonces[owner]++`) prevents signature replay attacks, the lack of proper ECDSA checks remains a deviation from best practices.

It's recommended to require the `s` value to be in the lower half order.
Reference: link

[L-02] Potential precision loss when decoding square root prices for low price values

When decoding sqrt prices in the decodeSqrtPrice function, the precision loss can occur for low price values, potentially affecting token ratio calculations during liquidity provisioning.

The decodeSqrtPrice function uses the following formula:

```
sqrtPrice = sqrtPriceX96 * GSMath.sqrt(decimals) / (2 ** 96);
```

The calculation can round down to 0 when: `sqrtPriceX96 < (2**96) / GSMath.sqrt(decimals)`

For token0 with 18 decimals:

- Rounds to 0 when `sqrtPriceX96` < 7.922816251426434e19 (approximately tick -414487)

For token0 with 6 decimals:

- Rounds to 0 when `sqrtPriceX96` < 7.922816251426434e25 (approximately tick -138163)

Since both these tick values are above `MIN_TICK` (-887272), the GammaVault could experience precision loss when providing liquidity across wide ranges that include low price points.

[L-03] Withdrawn hedge profit is not reinvested into Uniswap V3 position

Severity

Impact: Medium

Likelihood: Medium

Description

When the hedge is revised and needs to be reduced, after collateral rebalancing, if the hedge generated a profit after loan repayment, this profit is withdrawn to the `GammaVault` via `IGammaPool(gsPool).decreaseCollateral()`:

```
function _rebalanceAndRepayLiquidity(
    uint256_gsTokenId,
    uint256_liquidity,
    bytesmemory_path0,
    bytesmemory_path1
  ) internal {
    //...
        if(_missingAmounts[0] > 0 && _missingAmounts[1] > 0) {
            //... when the hedge incurs a loss
        } else {
            //... when the hedge incurs profit
            (,uint256[] memory paidAmounts) = IGammaPool(gsPool).repayLiquidity
              (_gsTokenId, _liquidity, 0, address(0));

            _loanData1 = getLoanData(_gsTokenId, false);

            _amounts = GSHedgeMath.calcWithdrawShare
              (_amounts, paidAmounts, _loanData1.tokensHeld);
            IGammaPool(gsPool).decreaseCollateral(_gsTokenId, _amounts, address
              (this), new uint256[](0));
        }
    //...
```

As per the documentation:

- the hedge profit should be **reinvested** into the Uniswap position if it is done during the **re-balancing** process.
- the hedge profit should be **distributed** to the users if it is done during the **withdrawal** process.

While the withdrawn profit is correctly distributed to the users during the withdrawal process, if the profit occurs during rebalancing, the withdrawn collateral is not reinvested into the Uniswap position, instead, it remains in the vault until the next process, resulting in a loss of LP fees for this un-invested collateral.

Recommendations

Deposit the collected hedge profit into the Uniswap position if it's collected during the rebalancing process.

[L-04] Rounding down in the `shares` calculation in favor of the user

In `FundingVault()`: when users call `withdraw()` to withdraw their deposited assets from the vault, the `shares` amount is calculated based on the requested `assets` to be withdrawn:

```
function withdraw(
        uint256 assets,
        address to,
        address from
    ) external virtual override lock returns (uint256 shares) {
        //...
        shares = _convertToShares(assets, _period);

        // Revert if redeeming 0 GS LP tokens
        if (shares == 0) revert ZeroShares();

        // Send CFMM LP tokens to receiver
        //(`to`) and burn corresponding GS LP tokens from msg.sender
        withdrawAssets(msg.sender, to, from, assets, shares, _period);
    }
```

Where:

```
function _convertToShares(
        uint256 assets,
        uint256 _period
    ) internal view virtual returns (uint256) {
        if (assets == 0) return 0;

        uint256 _totalSupply = periodTotalSupply[_period]; // Saves an extra
        // SLOAD if totalSupply is non-zero.
        uint256 _totalAssets = periodTotalAssets[_period];
        if (_totalSupply == 0 || _totalAssets == 0) {
            require(assets > MIN_SHARES, "FundingVault: MIN_SHARES");
            unchecked {
                return assets - MIN_SHARES;
            }
        }

        return ((assets * _totalSupply) / _totalAssets);
    }
```

- As can be noticed, the `_convertToShares()` function rounds down the calculated amounts of shares (that is going to be burnt to get the requested amount to be withdrawn), which favors the user instead of the vault, as users can burn less shares to get their deposited assets.

Update `_convertToShares()` function to round up the shares:

```
function _convertToShares(
        uint256 assets,
        uint256 _period
    ) internal view virtual returns (uint256) {
    //...

-       return ((assets * _totalSupply) / _totalAssets);
+       return (assets * _totalSupply + (_totalAssets - 1)) / _totalAssets;
    }
```

[L-05] Stale price ratios used when calculating deposit amounts after swaps

The `_depositAsset` function in BaseProcess.sol uses potentially stale price ratios when calculating optimal deposit amounts after performing token swaps. This can lead to suboptimal or incorrect deposit amounts being calculated.

The issue exists because:

- Initial price ratios are calculated
- A swap is performed that may change the pool's price
- The original (now stale) ratios are used to calculate deposit amounts via `calculateMaxDeposit`

It leads to the wrong `amount0` and `amount1` used for increasing liquidity.

```
function _depositAsset(
        uint160 sqrtPriceX96,
        uint160 sqrtPriceAX96,
        uint160 sqrtPriceBX96,
        bytes memory path0,
        bytes memory path1
    ) internal returns (uint256 _totalLiquidity) {
        uint256 amount0 = GammaSwapLibrary.balanceOf(token0, address(this));
        uint256 amount1 = GammaSwapLibrary.balanceOf(token1, address(this));
        while (amount0 > s.dust0 || amount1 > s.dust1) {
            (uint256 ratio0, uint256 ratio1) =
            RangedPoolMath.calculateTokenRatio(
              sqrtPriceX96,
              sqrtPriceAX96,
              sqrtPriceBX96,
              decimals0,
              decimals1
            );
            ...
            (amount0, amount1) = RangedPoolMath.calculateMaxDeposit
            //(isAssetToken0, amount0, amount1, ratio0, ratio1); // @audit using the s
            ...
        }
    }
```

It's recommended to calculate the new ratios after each swap to be used in `calculateMaxDeposit` function.

```
function _depositAsset(
        uint160 sqrtPriceX96,
        uint160 sqrtPriceAX96,
        uint160 sqrtPriceBX96,
        bytes memory path0,
        bytes memory path1
    ) internal returns (uint256 _totalLiquidity) {
        uint256 amount0 = GammaSwapLibrary.balanceOf(token0, address(this));
        uint256 amount1 = GammaSwapLibrary.balanceOf(token1, address(this));
+        (uint256 ratio0, uint256 ratio1) = RangedPoolMath.calculateTokenRatio
+ (sqrtPriceX96, sqrtPriceAX96, sqrtPriceBX96, decimals0, decimals1);
        while (amount0 > s.dust0 || amount1 > s.dust1) {
-            (uint256 ratio0, uint256 ratio1) =
-            RangedPoolMath.calculateTokenRatio
- (sqrtPriceX96, sqrtPriceAX96, sqrtPriceBX96, decimals0, decimals1);
            ...
+            (ratio0, ratio1) = RangedPoolMath.calculateTokenRatio
+ (sqrtPriceX96, sqrtPriceAX96, sqrtPriceBX96, decimals0, decimals1);
            (amount0, amount1) = RangedPoolMath.calculateMaxDeposit
            //(isAssetToken0, amount0, amount1, ratio0, ratio1); // @audit using the s
            ...
        }
    }
```

[L-06] Missing transaction deadline validation

The vault's liquidity management functions (`_increaseLiquidity` and `_decreaseLiquidity`) set transaction `deadlines` to `type(uint256).max`, effectively removing any timeout protection for liquidity operations. This creates several risks when during periods of network congestion, transactions could remain pending indefinitely. And delayed transactions could execute at unfavorable prices or conditions, particularly in volatile market conditions. For example, the transaction is executed when the prices of Uniswap V3 and CPMM diverge.

```
function _increaseLiquidity(
        uint256 _tokenId,
        uint256 amount0Desired,
        uint256 amount1Desired,
        uint256 amount0Min,
        uint256 amount1Min
    ) internal returns (uint128 liquidity, uint256 amount0, uint256 amount1) {
        if(_tokenId == 0) {

                ...
                deadline: type(uint256).max  // @audit No deadline protection
            });
            (
              s.tokenId,
              liquidity,
              amount0,
              amount1
            ) = IUniswapV3PositionManager(nftPosMgr
        } else {

                        IUniswapV3PositionManager.IncreaseLiquidityParams memory para
                .IncreaseLiquidityParams({
                ...
                    deadline: type(uint256).max // @audit No deadline protection
            });
            (liquidity, amount0, amount1) = IUniswapV3PositionManager
              (nftPosMgr).increaseLiquidity(params);
        }
    }

    function _decreaseLiquidity(
      uint256_tokenId,
      uint256liquidity,
      uint256amount0Min,
      uint256amount1Min
    )
        internal returns (uint256 amount0, uint256 amount1) {
        ...

                IUniswapV3PositionManager.DecreaseLiquidityParams memory params = IUn
            .DecreaseLiquidityParams({
                ...
                deadline: type(uint256).max // @audit No deadline protection
        ...
    }
```

Set the deadline by passing as an argument to the function.

[L-07] Using the wrong value when calling `calcUpdatedCollateral()`

In function `_rebalanceAndRepayLiquidity()`, the code executes a rebalance and then calculates the updated collateral amounts by calling `calcUpdatedCollateral()`:

```
if(_liquidity < _loanData0.liquidity) {
            uint256 remaining = _loanData0.liquidity - _liquidity;
            if(remaining <= 1e3) {
                _liquidity += 1e4; // Close entire position. 1e3 is MIN_BORROW
                // in GammaPool. 1e4 to avoid rounding issues
            }
        }

        uint128[] memory _collateral = GSHedgeMath.calcCollateralShare
          (_loanData0.tokensHeld, _liquidity, _loanData0.liquidity);
        uint128[] memory _tokensOwed = GSHedgeMath.calcTokensOwedShare
          (gsPool, _liquidity);

--snip--

        uint128[] memory _amounts = new uint128[](2);
        _amounts[0] = _collateral[0] + _collateral[0] / 1e4; // padding
        _amounts[1] = _collateral[1] + _collateral[1] / 1e4; // padding

                _amounts[0] = _amounts[0] > _loanData0.tokensHeld[0] ? _loanData0.tok

                _amounts[1] = _amounts[1] > _loanData0.tokensHeld[1] ? _loanData0.tok
        IGammaPoolExternal(gsPool).rebalanceExternally
          (_gsTokenId, _amounts, 0, hedgeRebalancer, abi.encode(data));

        IGammaPool.LoanData memory _loanData1 = getLoanData(_gsTokenId, false);
        _amounts = GSHedgeMath.calcUpdatedCollateral
          (_collateral, _loanData0.tokensHeld, _loanData1.tokensHeld);

       uint128[] memory _missingAmounts = GSHedgeMath.calcShortfall
         (_amounts, _tokensOwed);

       if(_missingAmounts[0] > 0 && _missingAmounts[1] > 0) {
--snip--
            IGammaPool(gsPool).repayLiquidity
              (_gsTokenId, _liquidity, 0, address(0));
        } else {
--snip--
            IGammaPool(gsPool).decreaseCollateral(_gsTokenId, _amounts, address
              (this), new uint256[](0));
        }
```

Because code increases `liquidity` by 1e4 when remaining liquidity is less than 1e3. So `liquidity` can be higher than `_loanData0.liquidity` and then `_collateral[]` can be higher than `_loanData0.tokensHeld`. The issue is that code use `_collateral[]` values when calling the `calcUpdatedCollateral()` and the value of the `_collateral[]` can be higher than `tokensHeld` amounts and it doesn't show the real amount code used to perform `rebalanceExternally()` which is `_amounts[]`. As a result when removed liquidity is almost all of the liquidity the recalculate `_amounts[]` will be slightly higher than their real value and it would cause `_missingAmounts[]` to be lower than real values. Code performs liquidity repayment or collateral decrease based on `_missingAmounts[]` values and if their values are wrong (lower than real value) code would perform decrease collateral while it should have performed repay-liquidity function and it would cause revert as there are not enough funds to repay the liquidity.

Consider using the value of `_amounts[]` when calling `calcUpdatedCollateral()`

[L-08] Incorrect `missingAmounts` check

When the hedge is revised and needs to be reduced, after collateral rebalancing, if the collateral held by the loan/hedge is insufficient to cover the debt repayment, the shortfall (`missingAmounts`) must be withdrawn from the Uniswap liquidity position to cover the loss. The protocol currently only supports WETH/USDC and will be initially deployed on Arbitrum, where:

- `token0` = WETH
- `token1` = USDC (as USDC's address > WETH's address on Arbitrum and Base).

```
function _rebalanceAndRepayLiquidity(
        uint256 _gsTokenId,
        uint256 _liquidity,
        bytes memory _path0,
        bytes memory _path1
    ) internal {

        //...
    if (_missingAmounts[0] > 0 && _missingAmounts[1] > 0) {

            _missingAmounts[0] = _missingAmounts[0] < 1e12
                ? 1e12
                : _missingAmounts[0];

            _missingAmounts[1] = _missingAmounts[1] < 1e4
                ? 1e4
                : _missingAmounts[1];

            _addMissingAmountsToHedge(
                _gsTokenId,
                _missingAmounts,
                _path0,
                _path1
            );

             IGammaPool(gsPool).repayLiquidity(
                _gsTokenId,
                _liquidity,
                0,
                address(0)
            );
        } else {
            //..
        }
        //...
    }
```

As can be noticed; the minimum thresholds for `missingAmounts` are hardcoded as: `token0` (WETH) → 1e12 `token1` (USDC) → 1e4

However, when the protocol is deployed on Ethereum, where `WETH` address > `USDC` address (`WETH` address: `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2`, `USDC` address `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48`),so: `token0` = USDC (6 decimals) `token1` = WETH (18 decimals)

where the hardcoded check 1e12 for `missingAmounts[0]` would translate to 1 million USDC (1_000_000e6), which is too high and unrealistic for the missing amount check.

- So the minimum amount comparision will be huge and incorrect resulting in an incoorect liquidity removal and over-hedging.

Recommendation:

Upate minimums of `missingAmounts` to be dynamically calculated based on the decimals of `token0` and `token1` instead of hardcoding these values.

[L-09] Small dust thresholds in rebalancing loop can cause token loss

The VaultRebalancer contract uses fixed dust thresholds `1e3` to determine when to break out of the rebalancing loop, without accounting for token decimals. This creates an issue that leads to token loss.

For tokens with high decimals like WETH (18 decimals), 1e3 Wei of WETH is an extremely small amount (0.000000000000001 WETH) that will likely result in a zero output when swapped. This leads to token loss as the contract repeatedly sends dust amounts that can't be meaningfully exchanged.

It leads to loss of tokens due to dust amounts being swapped with zero output. It is repeated multiple times at each external rebalancing. The accumulated value over time can be significant.

```
function rebalanceCollateralToRatio
      (address[] memory tokens, uint256[] memory ratio,
        ...

        for(uint256 i = 0; i < 10;) {
        ...
            if(bR0 < aR1 && (aR1 - bR0 > 1e10)) { // sell A
                ...

                if(sellQtyA < 1e3) break;

                IUniversalRouter(router).swapExactTokensForTokens(
                    sellQtyA, 0, path0, address(this), type(uint256).max
                );
            } else if(bR0 > aR1 && (bR0 - aR1 > 1e10)) { // sell B
                ...
                if(sellQtyB < 1e3) break;

                IUniversalRouter(router).swapExactTokensForTokens(
                    sellQtyB, 0, path1, address(this), type(uint256).max
                );
            } else {
                break;
            }

            unchecked {
                ++i;
            }
        }
    }
```

For example: it can be observed in a test case where WETH (amountIn) is swapped to USDC (amountOut) after 10 loops instead of stopping earlier. From the swap 5, the amountOut is 0:

1. _swap - amountIn: 98069305391033220 _swap - amountOut: 318441758
2. _swap - amountIn: 98069305391033220 _swap - amountOut: 318441758
3. _swap - amountIn: 46046461845172 _swap - amountOut: 149517
4. _swap - amountIn: 21720088027 _swap - amountOut: 70
5. _swap - amountIn: 86364762 _swap - amountOut: 0
6. _swap - amountIn: 40540410 _swap - amountOut: 0
7. _swap - amountIn: 19030039 _swap - amountOut: 0
8. _swap - amountIn: 8932874 _swap - amountOut: 0
9. _swap - amountIn: 4193173 _swap - amountOut: 0
10. _swap - amountIn: 1968314 _swap - amountOut: 0

It's recommended to scale thresholds based on token decimals.