KAWACH

Security Assessment Report

Dfyn V2

Version: 1.0

Date: 27 Apr 2023

Table of Contents

Licence

Disclaimer

This audit report is based on the findings of the audit conducted by our team, which focused on analyzing the Solidity smart contracts of Dfyn V2 Exchange.While we have taken all reasonable steps to ensure the accuracy and completeness of the report, we cannot guarantee that our findings are free from errors or omissions. The report should be used for informational purposes only and should not be construed as providing legal or investment advice. We have categorized our findings by level of severity, and provided recommendations to address the security issues we identified. It is our hope that this report will help inform decision making for the development team, and ultimately result in an improved level of security for the contract.

Introduction

Purpose of this report

Kawach has been engaged by **Dfyn Network** to perform a security audit of its contracts for the concentrated liquidity exchange.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behaviour.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebases Submitted for the Audit

The audit has been performed on the following GitHub repositories:

Repository	Commit hash
https://github.com/dfyn/dfyn-v2-contracts/tree/fix/vault-share	d28b3eaeac1d4f363a4be96502a2021d7e0e053b

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Overview

Methodology

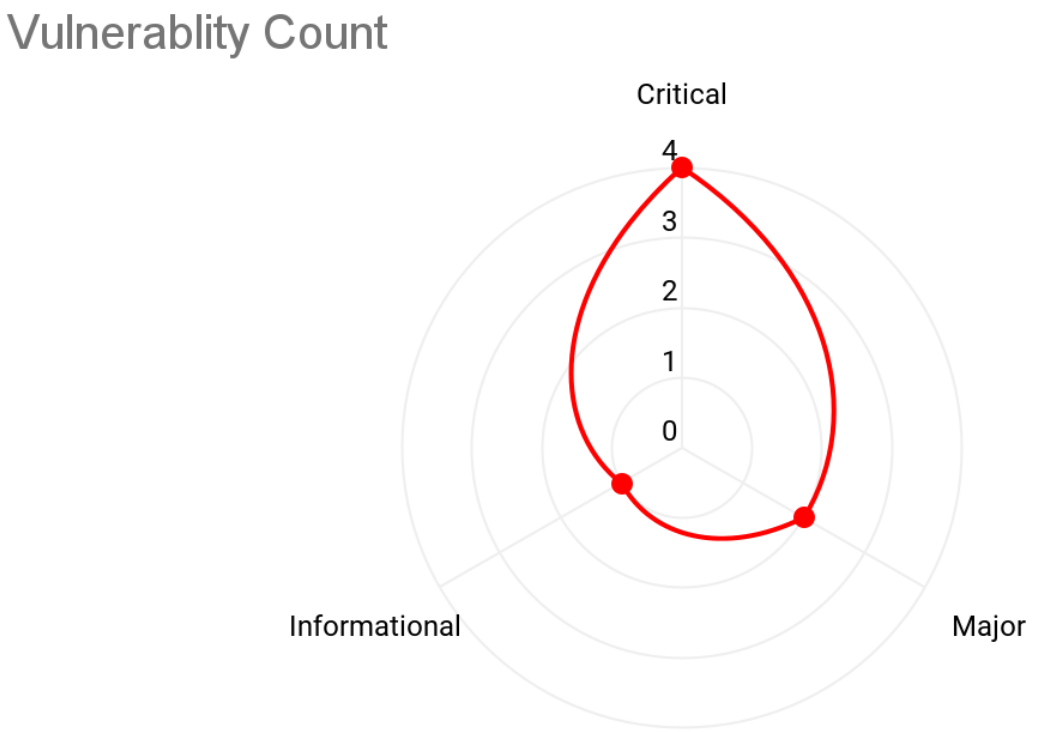
The audit has been performed in the following steps:

1. Gaining an understanding of the code base’s intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Dfyn V2's innovative Hybrid Model leverages concentrated liquidity AMM and limit order liquidity to offer a superior solution for decentralized on-chain limit orders. In this system, the limit order liquidity is superimposed on the concentrated liquidity curve to overcome the challenges of low liquidity and unpredictable slippage. By combining the advantages of both order types, the concentrated liquidity AMM allows the exchange to offer a high level of liquidity for popular trading pairs, while the limit order liquidity provides users with greater control over the execution price of their trades. This hybrid approach provides users with the best of both worlds, ensuring fast and reliable trade execution while giving them greater flexibility and control over their trades

Summary of Findings



Sr. No.	Description	Severity	Status
1	When calculating feeGrowth, the current tick must be based on the current price in order to be calculated correctly	Critical	Resolved
2	New users placing limit orders can claim previously filled limit orders placed by other users at that same price point	Critical	Resolved
3	After a partial limit order fill, minting can skip limit order liquidity for that price	Critical	Resolved
4	Balance check in mint(), burn(), and createLimitOrder() do not consider limitOrderFee variables	Critical	Resolved
5	Periphery contracts are not converting amount to shares	Major	Resolved
6	In _executeLimitOrder, the limitOrderFee variables are not correctly updated	Major	Resolved
7	Caching storage variables in local variables to save gas	Informational	Resolved

Detailed Findings

1. When calculating feeGrowth, the current tick must be based on the current price in order to be calculated correctly

Severity: Critical

Description

In the `rangeFeeGrowth()` function, the calculation for fee growth inside the current liquidity range should use the tick at the current price, but the function uses the `nearestTick` variable for that.

```
function rangeFeeGrowth(int24 lowerTick, int24 upperTick) public view returns (uint256
feeGrowthInside0, uint256 feeGrowthInside1) {
        int24 currentTick = nearestTick;


        Tick storage lower = ticks[lowerTick];
        Tick storage upper = ticks[upperTick];
        // Calculate fee growth below & above.
        uint256 _feeGrowthGlobal0 = feeGrowthGlobal0;
        uint256 _feeGrowthGlobal1 = feeGrowthGlobal1;
        uint256 feeGrowthBelow0;
        uint256 feeGrowthBelow1;
        uint256 feeGrowthAbove0;
        uint256 feeGrowthAbove1;


        if (lowerTick <= nearestTick) {
            feeGrowthBelow0 = lower.feeGrowthOutside0;
            feeGrowthBelow1 = lower.feeGrowthOutside1;
        } else {
...
```

Recommendation

Get the `currentTick` from the `price`

```
function rangeFeeGrowth(int24 lowerTick, int24 upperTick) public view returns (uint256
feeGrowthInside0, uint256 feeGrowthInside1) {
       int24 currentTick = TickMath.getTickAtSqrtRatio(price);



       Tick storage lower = ticks[lowerTick];
       Tick storage upper = ticks[upperTick];
       // Calculate fee growth below & above.
       uint256 _feeGrowthGlobal0 = feeGrowthGlobal0;
       uint256 _feeGrowthGlobal1 = feeGrowthGlobal1;
       uint256 feeGrowthBelow0;
       uint256 feeGrowthBelow1;
       uint256 feeGrowthAbove0;
       uint256 feeGrowthAbove1;



       if (lowerTick <= currentTick) {
           feeGrowthBelow0 = lower.feeGrowthOutside0;
           feeGrowthBelow1 = lower.feeGrowthOutside1;
       } else
...
```

Status

Resolved

2. New users placing limit orders can claim previously filled limit orders placed by other users at that same price point

Severity: Critical

Description

Currently, there is no check on who placed the limit order first or who can claim it first. As a result, new users who just placed a limit order at a specific price point may see that they can claim a filled order directly. To address this issue, you need to incorporate a method for determining who will claim the orders first.

Recommendation

To address the issue of new users being able to claim previously filled limit orders placed by other users at the same price point, we recommend incorporating a method for determining who will claim the orders first.

Therefore, we suggest that you use the limitOrder claimableGrowth logic just like it is used for feeGrowth.

Status

Resolved

3. After a partial limit order fill, minting can skip limit order liquidity for that price

Severity: Critical

Description

This issue occurs in the `swap()` function when it has depleted the concentrated liquidity at a specific tick, and sets the `cross` value to `true`.

Once `cross` is set to true, `currentPrice` is set to `nextTickPrice`, and `SwapExcecuter._executeLimitOrder()` is called. If a limit order is partially filled and `cross` is set to false, then `currentPrice` is the same as `nextTickPrice`, but `nearestTick` is still the old tick because crossing was not possible due to the lack of limit liquidity.

```
//checks limit order liquidity exist in next tick & executes if liquidity exist
          if (swapLocal.cross) {
              {
                  uint256 limitOrderLiquidity = zeroForOne
                      ? limitOrderTicks[cache.nextTickToCross].token1Liquidity
                      : limitOrderTicks[cache.nextTickToCross].token0Liquidity;
                  if (limitOrderLiquidity != 0) {
                      SwapExcecuter.ExecuteLimitResponse memory response =
SwapExcecuter._executeLimitOrder(
                          ExecuteLimitOrderParams({
                              sqrtpriceX96: swapLocal.nextTickPrice,
                              tick: cache.nextTickToCross,
                              zeroForOne: zeroForOne,
                              amountIn: cache.amountIn,
                              amountOut: cache.amountOut,
                              limitOrderAmountOut: cache.limitOrderAmountOut,
```

```
                            limitOrderAmountIn: cache.limitOrderAmountIn,
                            cross: swapLocal.cross,
                            token0LimitOrderFee: token0LimitOrderFee,
                            token1LimitOrderFee: token1LimitOrderFee,
                            exactIn: cache.exactIn,
                            limitOrderFee: limitOrderFee
                        }),
                        limitOrderTicks,
                        limitOrderLiquidity
                    );
                    // Set the state of cache and other variables in scope
                    cache.amountIn = response.amountIn;
                    cache.amountOut = response.amountOut;
                    swapLocal.cross = response.cross;
                    token0LimitOrderFee = response.token0LimitOrderFee;
                    token1LimitOrderFee = response.token1LimitOrderFee;
                    cache.limitOrderAmountOut = response.limitOrderAmountOut;
                    cache.limitOrderAmountIn = response.limitOrderAmountIn;
                    // reset amountIn and amountOut and update totalAmount
                    cache.totalAmount += cache.exactIn ? cache.amountOut : cache.amountIn;
                    (cache.amountOut, cache.amountIn) = cache.exactIn ? (uint256(0),
cache.amountIn) : (cache.amountOut, uint256(0));
                }
            }
            if (swapLocal.cross)
                (cache.currentLiquidity, cache.nextTickToCross) = Ticks.cross(
                    ticks,
                    Ticks.TickCrossRequest({nextTickToCross: cache.nextTickToCross,
currentLiquidity: cache.currentLiquidity}),
                    secondsGrowthGlobal,
                    cache.feeGrowthGlobalA,
                    cache.feeGrowthGlobalB,
                    zeroForOne,
                    tickSpacing
                );
        }
    }
    price = uint160(cache.currentPrice);
    int24 newNearestTick = zeroForOne ? cache.nextTickToCross :
ticks[cache.nextTickToCross].previousTick;
```

```
        if (nearestTick != newNearestTick) {
            nearestTick = newNearestTick;
            liquidity = uint128(cache.currentLiquidity);
        }
```

Afterwards, if you add liquidity to the range where your price at `lowerTick` is the same as the `currentPrice` tick, then you shift `nearestTick` to the `currentPrice` tick.

ConcentratedLiquidityPool.sol

```
function mint(MintParams memory mintParams) public lock returns (uint256 liquidityMinted) {
        Validators._ensureTickSpacing(mintParams.lower, mintParams.upper, tickSpacing);
        uint256 priceLower = uint256(TickMath.getSqrtRatioAtTick(mintParams.lower));


        uint256 priceUpper = uint256(TickMath.getSqrtRatioAtTick(mintParams.upper));
        uint256 currentPrice = uint256(price);

        ....
        (nearestTick, tickCount) = Ticks.insert(
            ticks,
            limitOrderTicks,
            feeGrowthGlobal0,
            feeGrowthGlobal1,
            secondsGrowthGlobal,
            mintParams.lowerOld,
            mintParams.lower,
            mintParams.upperOld,
            mintParams.upper,
            data
        );
```

Ticks.sol

```
function insert(
    mapping(int24 => IConcentratedLiquidityPoolStruct.Tick) storage ticks,
    mapping(int24 => IConcentratedLiquidityPoolStruct.LimitOrderTickData) storage limitOrderTicks,
    uint256 feeGrowthGlobal0,
    uint256 feeGrowthGlobal1,
```

```
    uint160 secondsGrowthGlobal,
    int24 lowerOld,
    int24 lower,
    int24 upperOld,
    int24 upper,
    IConcentratedLiquidityPoolStruct.InsertTickParams memory data
) public returns (int24, uint256) {
    require(lower < upper, "WRONG_ORDER");
    require(TickMath.MIN_TICK <= lower, "LOWER_RANGE");
    require(upper <= TickMath.MAX_TICK, "UPPER_RANGE");
    {
        ...
        ...
    if (data.nearestTick < upper && upper <= data.tickAtPrice) {
        unchecked {
            data.nearestTick = upper;
        }
    } else if (data.nearestTick < lower && lower <= data.tickAtPrice) {
        unchecked {
            data.nearestTick = lower;
        }
    }
    return (data.nearestTick, data.tickCount);
}
```

Because you shift `nearestTick`, it skips limit order liquidity and mimics a cross that would have occurred if the total limit order liquidity at that tick was used. This also does not adjust the global liquidity variable, which messes up the calculations and will underflow the `liquidity` variable at some point.

Recommendation

One solution is to keep another variable for `currentPrice`, which will point to the price at the previous or next tick. To do this, you can increment or decrement the price by 1, depending on the value of `zeroForOne`.

For instance, you can use a variable called `nearestPrice`.

```
function swap(bytes memory data, bytes calldata path) public lock returns (uint256 amountOut,
uint256 amountIn) {
        (bool zeroForOne, address recipient, bool unwrapVault, int256 quantity) = abi.decode(data,
```

```
(bool, address, bool, int256));
       SwapCache memory cache = SwapCache({
           protocolFee: 0,
           feeGrowthGlobalA: zeroForOne ? feeGrowthGlobal1 : feeGrowthGlobal0,
           feeGrowthGlobalB: zeroForOne ? feeGrowthGlobal0 : feeGrowthGlobal1,
           currentPrice: uint256(price),
           nearestPriceCached: uint256(nearestPrice),
           currentLiquidity: uint256(liquidity),
           amountIn: quantity > 0 ? uint256(quantity) : 0,
           amountOut: quantity > 0 ? 0 : uint256(-quantity),
           ...
           ...

           require(cache.nextTickToCross != TickMath.MIN_TICK && cache.nextTickToCross !=
TickMath.MAX_TICK, "E4");
           if (cache.currentLiquidity != 0) {
               (cache.amountOut, cache.currentPrice, swapLocal.cross, cache.amountIn,
swapLocal.fee) = SwapExcecuter
                   ._executeConcentrateLiquidity(
                       ConcStruct({
 ...
 ...
             cache.currentPrice = uint256(TickMath.getSqrtRatioAtTick(cache.nextTickToCross));
               swapLocal.cross = true;
           }
           cache.nearestPriceCached = cache.currentPrice;
           //checks limit order liquidity exist in next tick & excecutes if liquidity exist
           if (swapLocal.cross) {
               {
 ...
 ...
     (cache.amountOut, cache.amountIn) = cache.exactIn ? (uint256(0), cache.amountIn) :
(cache.amountOut, uint256(0));
                   }
               }
               if (swapLocal.cross) {
                   (cache.currentLiquidity, cache.nextTickToCross) = Ticks.cross(
                       ticks,
                       Ticks.TickCrossRequest({nextTickToCross: cache.nextTickToCross,
currentLiquidity: cache.currentLiquidity}),
```

```
                    secondsGrowthGlobal,
                    cache.feeGrowthGlobalA,
                    cache.feeGrowthGlobalB,
                    zeroForOne,
                    tickSpacing
                );
            } else {
                cache.nearestPriceCached = zeroForOne ? cache.currentPrice + 1 :
cache.currentPrice - 1;
            }
        }
    }
    price = uint160(cache.currentPrice);
    nearestPrice = uint160(cache.nearestPriceCached);
    int24 newNearestTick = zeroForOne ? cache.nextTickToCross :
ticks[cache.nextTickToCross].previousTick;


    if (nearestTick != newNearestTick) {
        nearestTick = newNearestTick;
        liquidity = uint128(cache.currentLiquidity);
    }
```

Use the `nearestPrice` as the `currentPrice` parameter in the `mint()` function.

```
function mint(MintParams memory mintParams) public lock returns (uint256 liquidityMinted) {
   Validators._ensureTickSpacing(mintParams.lower, mintParams.upper, tickSpacing);
   uint256 priceLower = uint256(TickMath.getSqrtRatioAtTick(mintParams.lower));


   uint256 priceUpper = uint256(TickMath.getSqrtRatioAtTick(mintParams.upper));
   uint256 currentPrice = uint256(nearestPrice);


   liquidityMinted = DyDxMath.getLiquidityForAmounts(
       priceLower,
       ...
       ...
```

Status

Resolved

4. Balance check in mint(), burn(), and createLimitOrder() do not consider limitOrderFee variables

Severity: Critical

Description

In the `mint()`, `burn()`, and `createLimitOrder()` functions, the manager contract is called via a callback which transfers the funds. After the callback, the balance of the pool is checked to verify that the correct amount has been transferred by the user.

According to our understanding, the balance of the contract is the sum of the reserve, limitOrderReserve, and limitOrderFee for that token. Therefore, the following equation should always hold:

```
reserve1 + limitOrderReserve1 + token1LimitOrderFee >= _balance(token1)
reserve0 + limitOrderReserve0 + token0LimitOrderFee >= _balance(token0)
```

However, the balance checks after the `mint()`, `burn()`, and `createLimitOrder()` functions do not consider the limitOrder fee:

```
if (reserve0 + limitOrderReserve0 > _balance(token0)) revert Token0Missing();
if (reserve1 + limitOrderReserve1 > _balance(token1)) revert Token1Missing();
```

As a result, a user can mint more than what they transferred.

Recommendation

Update the balance checking conditions to the following:

```
if (reserve0 + limitOrderReserve0 + token0LimitOrderFee > _balance(token0))
revert Token0Missing();
if (reserve1 + limitOrderReserve1 + token1LimitOrderFee > _balance(token1))
revert Token1Missing();
```

Status

Resolved

5. Periphery contracts are not converting amount to shares

Severity: Major

Description

The Pool and Vault contracts use shares to track balances. The Vault contract tracks depositors' balances as shares, which are then converted to actual amounts when they withdraw. These amounts include the profits generated by the vault through strategies and flash loans.

However, during testing we discovered that the Quoter and DfynSignal contracts are not converting input amounts to shares. As a result, if the Vault contract generates profits (i.e., `amount > shares` in the vault), swap transactions through DfynSignal can revert.

DfynSignal depositing into the vault without converting amount into shares

```
function payToken(address token, address payer, address recipient, uint256 value) internal {
        if (payer == address(this)) {
            IERC20(token).approve(address(vault), value);
        }
        if (token == wETH && address(this).balance >= value) {
            // pay with WETH
            vault.deposit{value: value}(address(0), address(this), recipient, value, 0);
        } else if (vault.balanceOf(token, payer) < value) {
            vault.deposit{value: 0}(token, payer, recipient, value, 0);
        } else {
            vault.transfer(token, payer, recipient, value);
        }
    }
```

Recommendation

We recommend converting the amount to shares in periphery contracts.

Convert the inputs to shares before calling the Pool contracts. This has to be changed in all the contracts which interact with the ConcentratedLiquidityPool contract.

For example, the `exactInputSingle` function should convert `amountIn` to shares and then call the Pool's `swap()` function.

```
function exactInputSingle(
        ExactInputSingleParamsDfyn memory params
    ) external payable override returns (uint256 amountOut) {
        bool hasAlreadyPaid;
        if (params.amountIn == Constants.CONTRACT_BALANCE) {
            hasAlreadyPaid = true;
            params.amountIn = IERC20(params.tokenIn).balanceOf(address(this));
        }
        params.amountIn = vault.toShare(params.tokenIn, params.amountIn, false);
        params.amountOutMinimum = vault.toShare(params.tokenOut, params.amountOutMinimum, false);
        address payer = hasAlreadyPaid ? address(this) : msg.sender;
        amountOut = exactInputInternal(
            params.amountIn,
            params.recipient,
            params.unWrapVault,
            SwapCallbackDataDfyn({path: abi.encodePacked(params.tokenIn, params.tokenOut), payer:
payer})
        );
        require(amountOut >= params.amountOutMinimum, 'Too little received');
    }
```

Status

Resolved

6. In _executeLimitOrder, the limitOrderFee variables are not correctly updated

Severity: Major

Description

The `_executeLimitOrder()` function inside the SwapExcecuter library executes the limit order logic and returns the output in an `ExecuteLimitResponse` struct.

```
function _executeLimitOrder(
    ILimitOrderStruct.ExecuteLimitOrderParams memory data,
```

```
    mapping(int24 => IConcentratedLiquidityPoolStruct.LimitOrderTickData) storage limitOrderTicks,
    uint256 limitOrderLiquidity
) internal returns (ExecuteLimitResponse memory response) {
    uint256 feeAmount;
    (
        feeAmount,
        response.amountIn,
        response.amountOut,
        response.limitOrderAmountOut,
        response.limitOrderAmountIn,
        response.cross
    ) = Ticks.excecuteLimitOrder(limitOrderTicks, data, limitOrderLiquidity);
    if (data.zeroForOne) {
        response.token0LimitOrderFee = data.token0LimitOrderFee + feeAmount;
    } else {
        response.token1LimitOrderFee = data.token1LimitOrderFee + feeAmount;
    }
}
```

Depending on the value of `data.zeroForOne`, either `response.token0LimitOrderFee` or `response.token1LimitOrderFee` is set, and the other token's limitOrderFee remains 0.

In the ConcentratedLiquidityPool contract, the `token0LimitOrderFee` and `token1LimitOrderFee` are updated after the call to `SwapExcecuter._executeLimitOrder`. However, one of them is overwritten as 0. This will mess up the calculation of the contract and future swaps will start to revert.

```
SwapExcecuter.ExecuteLimitResponse memory response = SwapExcecuter._executeLimitOrder(
                ExecuteLimitOrderParams({
                    sqrtpriceX96: swapLocal.nextTickPrice,
                    tick: cache.nextTickToCross,
                    zeroForOne: zeroForOne,
                    amountIn: cache.amountIn,
                    amountOut: cache.amountOut,
                    limitOrderAmountOut: cache.limitOrderAmountOut,
                    limitOrderAmountIn: cache.limitOrderAmountIn,
                    cross: swapLocal.cross,
                    token0LimitOrderFee: cache.token0LimitOrderFee,
                    token1LimitOrderFee: cache.token1LimitOrderFee,
```

```
                exactIn: cache.exactIn,
                limitOrderFee: limitOrderFee
            }),
            limitOrderTicks,
            limitOrderLiquidity
        );
        // Set the state of cache and other variables in scope
        cache.amountIn = response.amountIn;
        cache.amountOut = response.amountOut;
        swapLocal.cross = response.cross;
        token0LimitOrderFee = response.token0LimitOrderFee;
        token1LimitOrderFee = response.token1LimitOrderFee;
```

Recommendation

In the SwapExcecuter contract, set both `token0LimitOrderFee` and `token1LimitOrderFee` to the current `limitOrderFee` initially.

```
function _executeLimitOrder(
   ILimitOrderStruct.ExecuteLimitOrderParams memory data,
   mapping(int24 => IConcentratedLiquidityPoolStruct.LimitOrderTickData) storage limitOrderTicks,
   uint256 limitOrderLiquidity
) internal returns (ExecuteLimitResponse memory response) {
   ...
   ...
   response.token0LimitOrderFee = data.token0LimitOrderFee;
   response.token1LimitOrderFee = data.token1LimitOrderFee;
   if (data.zeroForOne) {
       response.token0LimitOrderFee += feeAmount;
   } else {
       response.token1LimitOrderFee += feeAmount;
   }
}
```

Status

Resolved

7. Caching storage variables in local variables to save gas

Severity: Informational

Description

Inside `swap()`, the `token0LimitOrderFee` and `token1LimitOrderFee` state variables are updated after the execution of limit orders, which occurs inside a while loop. It is not recommended to read from and write to storage slots inside loops, as they tend to be more expensive than memory variables.

Recommendation

Store the storage variables in the `cache` struct inside memory.

```
SwapCache memory cache = SwapCache({
            protocolFee: 0,
            feeGrowthGlobalA: zeroForOne ? feeGrowthGlobal1 : feeGrowthGlobal0,
            feeGrowthGlobalB: zeroForOne ? feeGrowthGlobal0 : feeGrowthGlobal1,
            ...
            ...
            token0LimitOrderFee: token0LimitOrderFee,
            token1LimitOrderFee: token1LimitOrderFee
        });
```

Then access and update the variables as `cache.token0LimitOrderFee` and `cache.token1LimitOrderFee` inside the loop. At the end of the loop again update the storage variables.

Status

Resolved

Conclusion

Overall, the audit revealed several security issues that need to be addressed to ensure the contract's security. The Dfyn team has worked on resolving all the reported issues with our recommended fixes.