QuillAudits

Audit Report
March, 2023

For

Dfyn

Table of Content

Executive Summary

Project Name	Dfyn
Overview	Dfyn concentrated liquidity contracts are designed to allow for deployment of token pairs pools, provision of liquidity with ticks feature, and ensure swaps. It also integrates the system of limit order to moderate the buys and sells that interests a user set ticks. Libraries were used to aid arithmetic and mathematical computation in contracts.
Timeline	Dec 15,2023 - 6 March, 2023
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Vault, Pool, Limit Order contracts and its associated libraries.
GithubLink	*https://github.com/dfyn/V2-Contracts/tree/audit/contracts*
Commit Hash	*1bb84da5eb1720085a94d97cbf055ca71ae052f9*
Fixed in	*https://github.com/dfyn/dfyn-v2-contracts/tree/6d963298984dde5f042cfcaa479491ca53cb38fe*

14
Issues Found

🟥 High 🟨 Medium

🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	**1**	**2**	**4**	**4**
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	**1**	0	**2**

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open
Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved
These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged
Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved
Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Using block.timestamp
- ✓ Multiple Sends
- ✓ Using SHA3
- ✓ Using suicide
- ✓ Using throw
- ✓ Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Vault.sol

High Severity Issues

A1. Stuck funds in the Vault can be withdrawn by anyone

A user doesn't need to send any ether (or WETH) for depositing in the Vault, if someone has already deposited some Ether in the Vault or if there is some Ether stuck in the Vault.

For example, let's say the Vault has 0.5 Ether. And a user wants to deposit 0.25 WETH. Then even if he just calls the deposit() function, he will be able to deposit 0.25 WETH, because it is the Vault that actually deposits this amount. An attacker can exploit this by setting his own address in the from and to parameters of this function. And withdrawing this so-called "deposited" amount.

Recommendation

Consider fixing issue A.2 issue in order to fix this issue. Add specific checks to ensure that additional Ether is not sent accidentally to the contract.

Status

Acknowledged

Medium Severity Issues

A2. Funds stuck issue in the Vault

The funds deposited in the vault can get stuck in the following scenarios.

Scenario 1: Consider the following scenario-
- A person wants to deposit 1000 * 10e-18 WETH. For this the amount parameter the user sets the amount parameter for the deposit() function as 1000.
- But then while calling the deposit function() of the Vault, he specifies 1 Ether as value(as it is a payable function). This would result in 1000 * 10e-18 WETH getting deposited successfully, but the rest of the amount of Ether (1 Ether - 1000 wei) will get stuck in the Vault. The depositor will no longer be able to withdraw this stuck amount.

Scenario 2: This issue can occur in the following scenario as well-
- The user specifies amount: 10^18 and share: 1000 as parameters, while specifying 1 ether as value while calling the payable deposit() function.
- This will result in the user depositing 1 ether completely while only 1000 wei or 1000^10-e18 WETH will only be withdrawable. The rest of the ETH will get stuck in the vault.

Recommendation

In order to prevent this issue, consider refunding the remaining amount immediately to the person depositing in the Vault

Status

Acknowledged

Low Severity Issues

No issues were found

Informational Issues

No issues were found

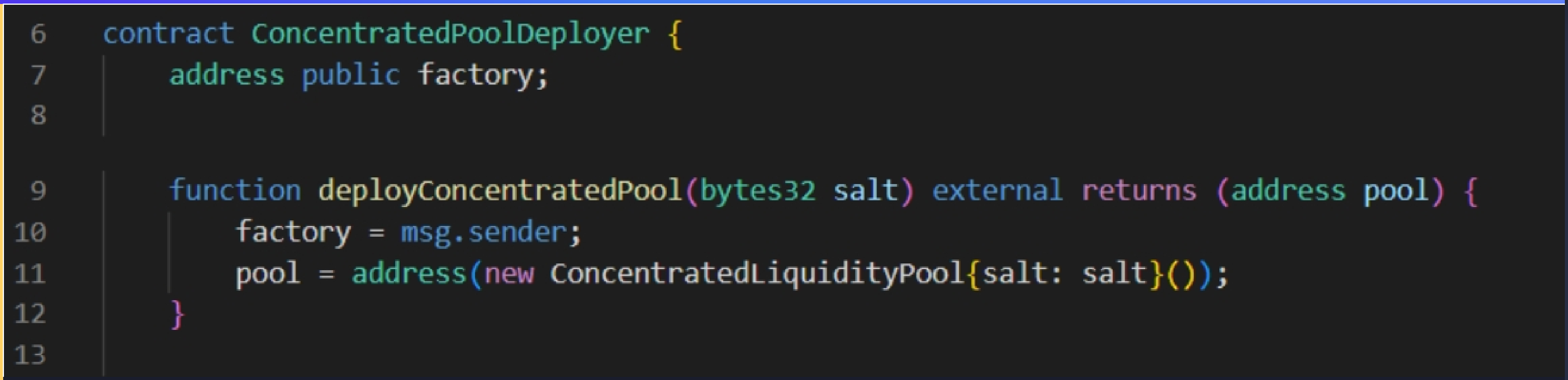
B. ConcentratedPoolDeployer.sol

High Severity Issues

No issues were found

Medium Severity Issues

B1. The factory variable can be overwritten



```
contract ConcentratedPoolDeployer {
    address public factory;

    function deployConcentratedPool(bytes32 salt) external returns (address pool) {
        factory = msg.sender;
        pool = address(new ConcentratedLiquidityPool{salt: salt}());
    }

```

Anyone can call deployConcentratedPool(). Due to this, the value in the variable factory can be overwritten.

For example, let's say a user deploys a pool using deployPool() function from ConcentratedLiquidityPoolFactory. This would set the factory variable as the address of the ConcentratedLiquidityPoolFactory contract.

Now if an attacker calls the deployConcentratedPool() function directly (instead of calling deployPool() in ConcentratedLiquidityPoolFactory) to deploy another pool, the factory variable will be overwritten with the address of the attacker. This can be detrimental if other contracts rely on this factory variable for fetching the factory contract's address and can be subject to attacks such as denial of service.

Recommendation

Consider adding necessary access controls and checks in the contract in order to prevent this issue. Also, consider reviewing the business and operational logic.

Status

Acknowledged

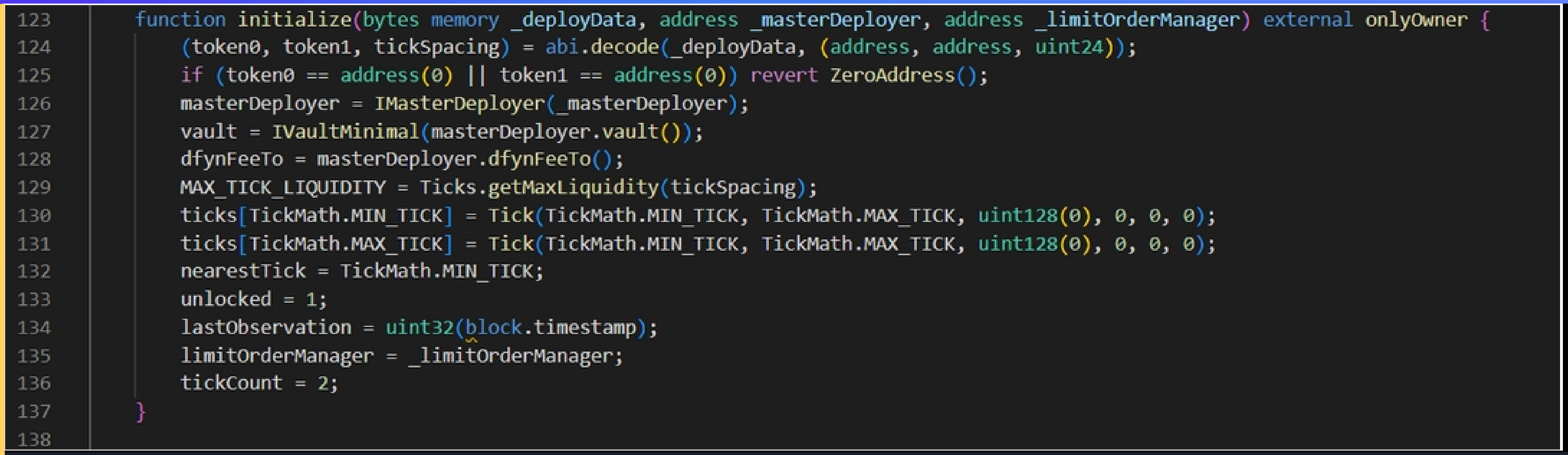
C. ConcentratedLiquidityPool.sol

High Severity Issues

No issues were found

Medium Severity Issues

C1. initialize can be called twice



```
    function initialize(bytes memory _deployData, address _masterDeployer, address _limitOrderManager) external onlyOwner {
        (token0, token1, tickSpacing) = abi.decode(_deployData, (address, address, uint24));
        if (token0 == address(0) || token1 == address(0)) revert ZeroAddress();
        masterDeployer = IMasterDeployer(_masterDeployer);
        vault = IVaultMinimal(masterDeployer.vault());
        dfynFeeTo = masterDeployer.dfynFeeTo();
        MAX_TICK_LIQUIDITY = Ticks.getMaxLiquidity(tickSpacing);
        ticks[TickMath.MIN_TICK] = Tick(TickMath.MIN_TICK, TickMath.MAX_TICK, uint128(0), 0, 0, 0);
        ticks[TickMath.MAX_TICK] = Tick(TickMath.MIN_TICK, TickMath.MAX_TICK, uint128(0), 0, 0, 0);
        nearestTick = TickMath.MIN_TICK;
        unlocked = 1;
        lastObservation = uint32(block.timestamp);
        limitOrderManager = _limitOrderManager;
        tickCount = 2;
    }
```

initialize can be called twice by the owner. This would result in changing of deployData, masterDeployer and limitOrderManager.

This could result further in unintended or undiscovered bugs such as changing the token addresses via deployData parameter and further exploitation.

Recommendation

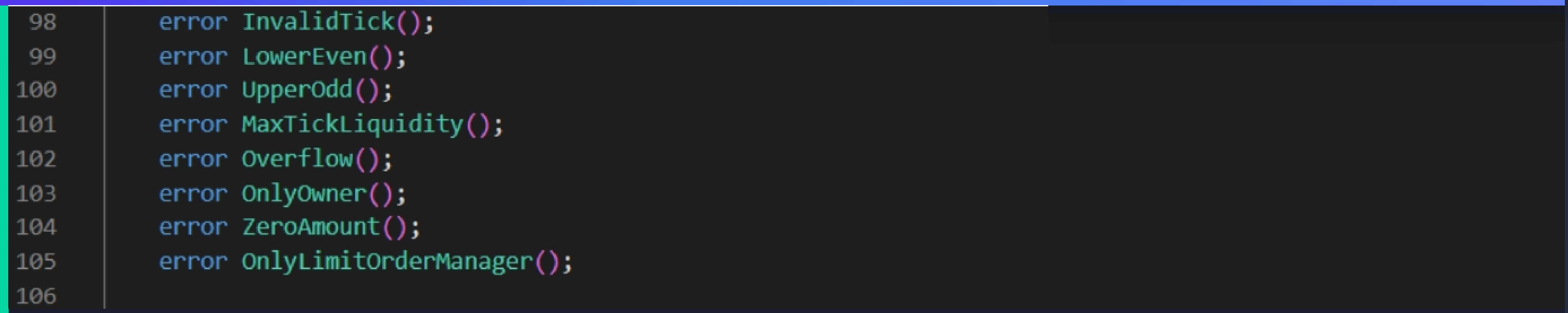
It is advised to restrict calling of the initialize function to a single call and disallowing from calling this function more than once.

Status

Resolved

Low Severity Issues

C2. Redundant errors



```
    error InvalidTick();
    error LowerEven();
    error UpperOdd();
    error MaxTickLiquidity();
    error Overflow();
    error OnlyOwner();
    error ZeroAmount();
    error OnlyLimitOrderManager();
```

The errors InvalidTick(), InvalidToken(), LowerEven() and UpperOdd() are never used in the ConcentratedLiquidityPool contract. (It is being used in Validators.sol and not ConcentratedLiquidityPool.sol so it can be removed).

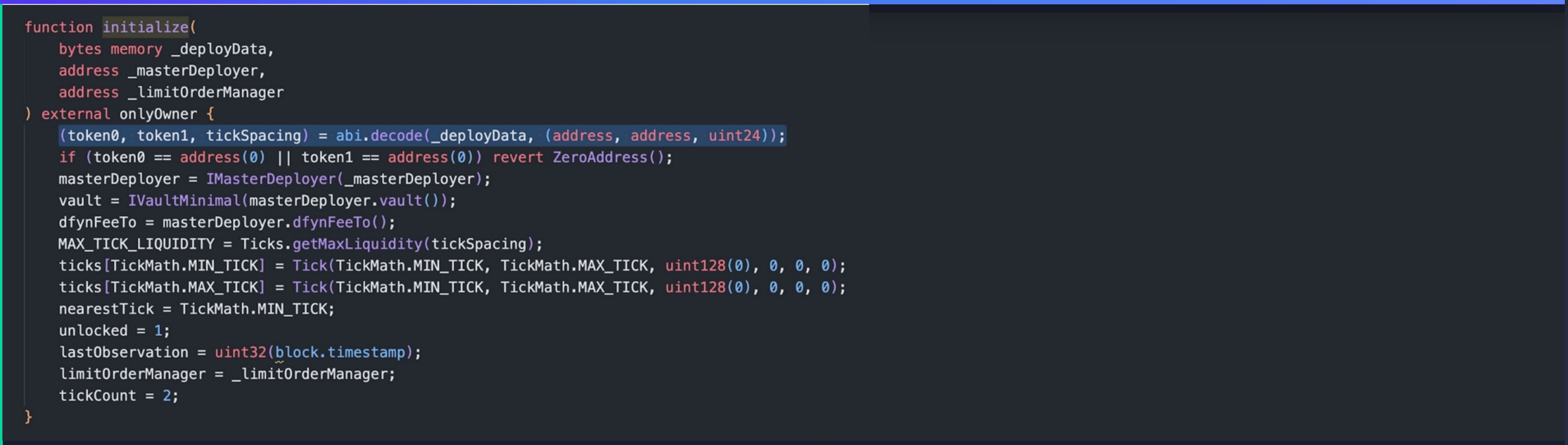
Recommendation

It is advised to remove the above-mentioned errors as they are redundant.

Status

Acknowledged

C3. Missing sanity checks in inputs of initialize function



```
function initialize(
    bytes memory _deployData,
    address _masterDeployer,
    address _limitOrderManager
) external onlyOwner {
    (token0, token1, tickSpacing) = abi.decode(_deployData, (address, address, uint24));
    if (token0 == address(0) || token1 == address(0)) revert ZeroAddress();
    masterDeployer = IMasterDeployer(_masterDeployer);
    vault = IVaultMinimal(masterDeployer.vault());
    dfynFeeTo = masterDeployer.dfynFeeTo();
    MAX_TICK_LIQUIDITY = Ticks.getMaxLiquidity(tickSpacing);
    ticks[TickMath.MIN_TICK] = Tick(TickMath.MIN_TICK, TickMath.MAX_TICK, uint128(0), 0, 0, 0);
    ticks[TickMath.MAX_TICK] = Tick(TickMath.MIN_TICK, TickMath.MAX_TICK, uint128(0), 0, 0, 0);
    nearestTick = TickMath.MIN_TICK;
    unlocked = 1;
    lastObservation = uint32(block.timestamp);
    limitOrderManager = _limitOrderManager;
    tickCount = 2;
}
```

The initialize function of concetratedliquiditypool contract takes three inputs, _deployData, _masterDeployer and, _limitOrderManager. The bytes input _decodeData is decoded and the output is set to values token0, token1, tickSpacing. However, there are no checks on the output of decoded data. For instance, if by mistake the owner initializes it with same values of token0 and token1, it will accept the values without performing any kind of checks on the returned output.

Recommendation

Consider adding a require check that ensures that token1 and token0 are not the same addresses.

Status

Acknowledged

Informational Issues

C4. Functions declared with Public Visibility but not used within the Contract should be made external

Some functions in the contract are declared with the public visibility and are not called within the function. For gas optimization purposes, it is recommended to label these functions as external. Input parameters provided to external functions are read from calldata unlike public functions that automatically copy to memory which invariably cost more gas.

Recommendation

Change visibility label from public to external to save gas.

Status

Acknowledged

C5. State variable that can be declared immutable

State variables that get initialized in the constructor and don't change their value throughout the code, should be declared immutable to save gas. Here, the following variables could be declared immutable:

- owner

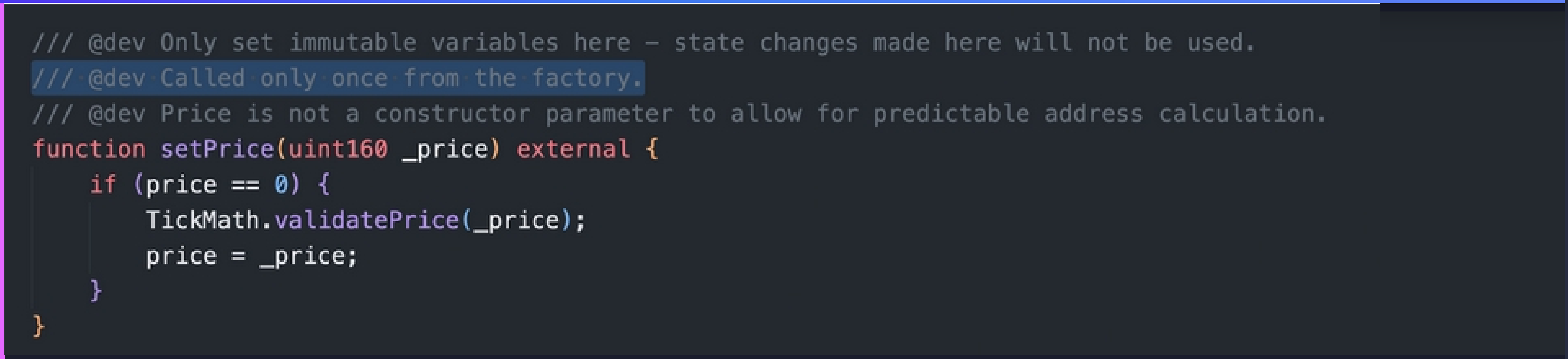
Recommendation

Consider declaring these variables as immutable.

Status

Acknowledged

C6. Contradicting code implementation with the comments.



```
/// @dev Only set immutable variables here - state changes made here will not be used.
/// @dev Called only once from the factory.
/// @dev Price is not a constructor parameter to allow for predictable address calculation.
function setPrice(uint160 _price) external {
    if (price == 0) {
        TickMath.validatePrice(_price);
        price = _price;
    }
}
```

The setPrice function in concentratedLiquidityPool contract is used to set the price of price variable and is only called once. In the comments above the code, it is mentioned that it should only be called by the factory contract. However, in the actual implementation, it is an external function and there is no check on whether it is being called from factory contact or not.

Recommendation

Review the logic of the contract and if required, add a modifier that ensures that the function can only be called by factory contract only once. It won't be a major threat because at the time of deployment of the pool contract from the factory contract, the setPrice function is immediately being called. However, if the logic requires it to be called only from the factory contract, necessary changes should be made. Similar issue is with the mint function. It is mentioned in the comments that it should be called from the CL pool manager contract. However, there are no access controls in place to ensure that it is being called from the CL pool manager contract. It can be called directly. We do acknowledge that this might be a development practice to be callable externally. Hence, we recommend developers to validate the issue of conflicting comments and make changes if required.

Status

Resolved

D. ConcentratedLiquidityPoolFactory.sol

High Severity Issues

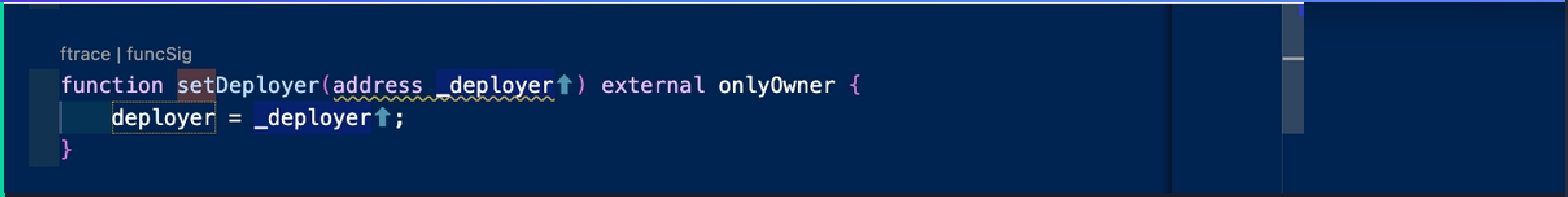
No issues were found

Medium Severity Issues

No issues were found

Low Severity Issues

D1. Missing Events for Significant Actions



```
ftrace | funcSig
function setDeployer(address _deployer) external onlyOwner {
    deployer = _deployer;
}
```

When some significant actions such as setting or changing of the concentrated liquidity pool deployer, or other privileged actions, it is recommended to emit parameters from this action to enable easy tracking and filtering of data.

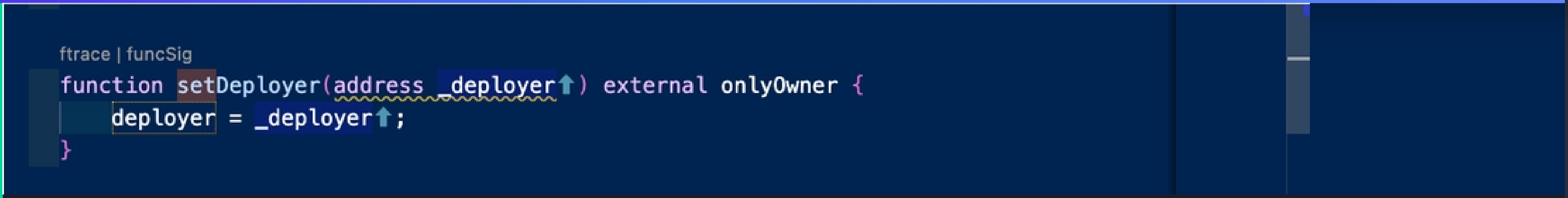
Recommendation

it is recommended to emit events for significant action like setting the concentrated liquidity pool deployer for a factory.

Status

Acknowledged

D2. Missing Zero Address Check



```
ftrace | funcSig
function setDeployer(address _deployer) external onlyOwner {
    deployer = _deployer;
}
```

The setDeployer() function receives an address variable and updates immediately the deployer state variable to the parameter passed to the function. With the exemption of necessary checks to prevent setting the null address as the deployer which could cause a possible lapse in the system flow, the present function will allow a malicious owner to set the deployer to the null address in order to break the system. The present function design will allow setting an existing deployer to be the new deployer, which of its own, is a waste of gas.

Recommendation

add checks to prevent setting the null address as the deployer and also setting the existing deployer to be the new deployer.

Status

Acknowledged

Informational Issues

No issues were found

E. ConcentratedLiquidityPoolManager.sol

High Severity Issues

No issues were found

Medium Severity Issues

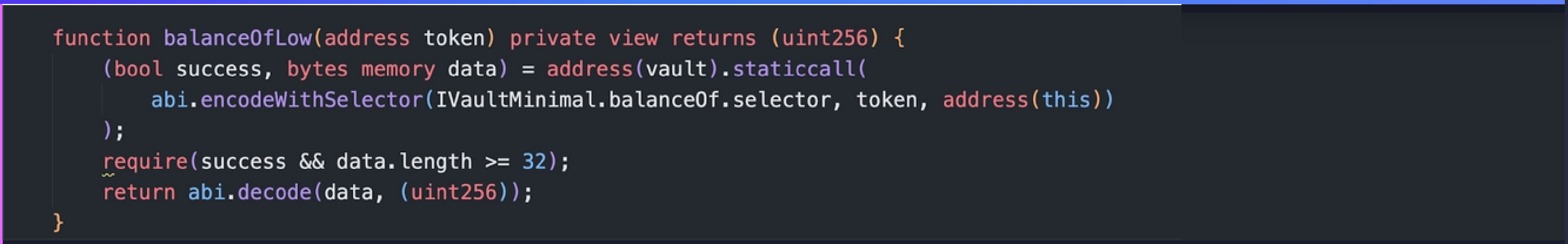
No issues were found

Low Severity Issues

No issues were found

Informational Issues

E1. Incorrect naming convention



```
function balanceOfLow(address token) private view returns (uint256) {
    (bool success, bytes memory data) = address(vault).staticcall(
        abi.encodeWithSelector(IVaultMinimal.balanceOf.selector, token, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
}
```

In solidity developers usually follow a standard naming convention pattern it not only makes the code more readable but also helps in quickly understand the code. While the whole dfyn code base follows the standard solidity coding pracitces. There is a private function in ConcentratedLiquidityPoolManager.sol contract named balanceOfLow. According to standard naming convention paractices, all private functs begin with an _ (underscore prefix). Hence it is recommended to append an underscore to the name of a private function.

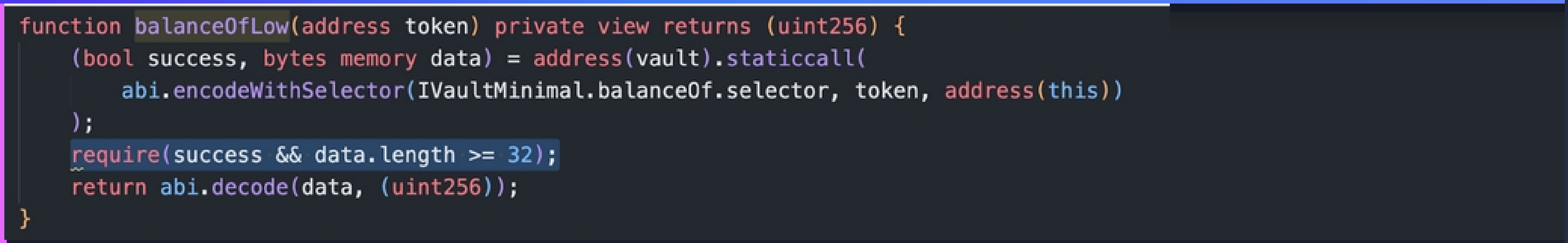
Recommendation

Consider renaming balanceOfLow to _balanceOfLow.

Status

Acknowledged

E2. Require statements without reason string



```
function balanceOfLow(address token) private view returns (uint256) {
    (bool success, bytes memory data) = address(vault).staticcall(
        abi.encodeWithSelector(IVaultMinimal.balanceOf.selector, token, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
}
```

The balanceOfLow has a require statement. However, the reason string (error message) is missing. It is a best practice to have a human-readable revert reason, unique across the contract. Hence, whenever a transaction fails, the proper human-readable error is displayed.

Recommendation

Consider adding a reason string to the require statement.

Status

Acknowledged

F. ConcentratedLiquidityPoolHelper.sol

High Severity Issues

No issues were found

Medium Severity Issues

No issues were found

Low Severity Issues

No issues were found

Informational Issues

No issues were found

G. LimitOrderManager.sol

High Severity Issues

No issues were found

Medium Severity Issues

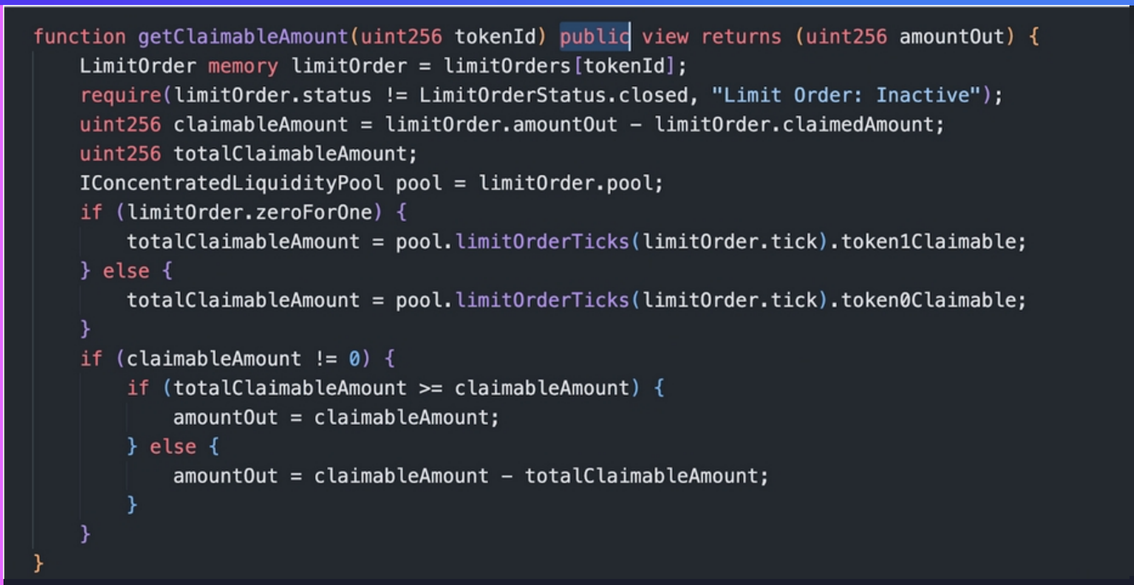
No issues were found

Low Severity Issues

No issues were found

Informational Issues

G1. Functions declared with Public Visibility but not used within the Contract should be made external



```
function getClaimableAmount(uint256 tokenId) public view returns (uint256 amountOut) {
    LimitOrder memory limitOrder = limitOrders[tokenId];
    require(limitOrder.status != LimitOrderStatus.closed, "Limit Order: Inactive");
    uint256 claimableAmount = limitOrder.amountOut - limitOrder.claimedAmount;
    uint256 totalClaimableAmount;
    IConcentratedLiquidityPool pool = limitOrder.pool;
    if (limitOrder.zeroForOne) {
        totalClaimableAmount = pool.limitOrderTicks(limitOrder.tick).token1Claimable;
    } else {
        totalClaimableAmount = pool.limitOrderTicks(limitOrder.tick).token0Claimable;
    }
    if (claimableAmount != 0) {
        if (totalClaimableAmount >= claimableAmount) {
            amountOut = claimableAmount;
        } else {
            amountOut = claimableAmount - totalClaimableAmount;
        }
    }
}
```

The contract has a view function named getClaimableAmount which is used to get the value of AmoutOut based on tokenID provided. The visibility of the function is public. However, it is not being called in the contract. It is only called externally. Hence it is recommended to declare such functions as external in order to save gas.

Recommendation

Change visibility label from public to external to save gas.

Status

Resolved

H. LimitOrderToken.sol

High Severity Issues

No issues were found

Medium Severity Issues

No issues were found

Low Severity Issues

No issues were found

Functional Testing

Some of the tests performed are mentioned below:

- ✓ Should get addresses that are given certain privileges.
- ✓ Should revert when deployPool function is called from masterDeployer with a non-whitelisted factory address.
- ✓ Should successfully remove whitelisted factories and prevent deploying pools with these factories.
- ✓ Should successfully deploy pools for different pairs of tokens with accurate deployData that follows token order.
- ✓ Should revert when setDfynFee is set beyond the MAX_FEE.
- ✓ Should revert when a non-owner attempts to whitelist master contracts.
- ✓ Should successfully allow the contract owner to whitelist master contracts.
- ✓ Should revert when the caller of setMasterContractApproval is not the same as the user.
- ✓ Should revert when the caller of setMasterContractApproval is a clone of the master contract.
- ✓ Should revert when the address passed to setMasterContractApproval is not whitelisted.
- ✓ Should successfully call the setMasterContractApproval function when the address is already whitelisted.
- ✓ Should deposit varieties of tokens into the vault.
- ✓ Should revert when users with less amount transfer out of the vault.
- ✓ Should get the token reserves after providing liquidity into the concentrated liquidity pool.
- ✓ Should get the value of liquidity after providing into the concentrated liquidity pool.
- ✓ Should make the vault give necessary approvals to some contract as the concentrated liquidity pool manager, limit order manager and the router.
- ✓ Should ascertain the update of position and the range fee growth on every mint and collect functions call from the concentrated liquidity pool manager.
- ✓ Should burn tick position nft minted and reset position when removing from the concentrated liquidity pool.
- ✓ Should get the value of exactInputSingle and exactOutputSingle after a swap via a router.
- ✓ Should create and increase the limitOrderId when the createLimitOrder function is called.
- ✓ Should send expected claimable amount from calling the claimLimitOrder either to the user or the vault.
- ✓ Should ensure that only active limit order can be canceled by the cancelLimitOrder function.
- ✓ Should burn limitOrderId when calling the cancelLimitOrder.

Closing Summary

In this report, we have considered the security of the Dfyn. We performed our audit according to the procedure described above.

Some issues of High,Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Dfyn Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Dfyn Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+	**$16B**	**700K**
Audits Completed	Secured	Lines of Code Audited

Follow Our Journey

Audit Report
March, 2023

For

Dfyn

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com