Making Web3 Space Safe for Everyone

KALOS

DFX Finance v3

Security Assessment

Published on : 07 Nov. 2023
Version v1.1

Security Report Published by KALOS

v1.1 07 Nov. 2023

Auditor : Jade Han, Hun Lee

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	4	3	1	-
High	2	2	-	-
Medium	-	-	-	-
Low	1	1	-	-
Tips	2	2	-	-

TABLE OF CONTENTS

ABOUT US

Making Web3 Space Safer for Everyone

KALOS is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

We have secured the most well-known web3 services including 1inch, SushiSwap, Klaytn, Badger DAO, SuperRare, Netmarble, Klaytn and Chainsafe. We have secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges.

KALOS is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@kalos.xyz

Website: https://kalos.xyz

Executive Summary

Purpose of this report

This report was prepared to audit the security of the contracts developed by the DFX team. KALOS conducted the audit focusing on whether the system created by the DFX team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the contracts. In detail, we have focused on the following.

- Changes from v2 to v3
- input validation of the curve generation, which is now public
- sanity check of various formulas used
- the safety of various contracts
- Pool fund drainage

Codebase Submitted for the Audit

The code used in this audit can be found on GitHub (https://github.com/dfx-finance/protocol-v2/tree/universal)
The commit hash of the code used for this audit is `acdc5fbe8e96f8c800c6f546a7469f513da14c1c`.
The patched code is on GitHub (https://github.com/dfx-finance/protocol-v3/commit/91874d1be9a5bf5de054149b528206f86ded8610).

Audit Timeline

Date	Event
2023/08/21	Audit Initiation
2023/09/01	Delivery of v1.0 report.
2023/11/07	Delivery of v1.1 report.

Findings

KALOS found 4 Critical, 2 High, 0 Medium, and 1 Low severity issues. There are 2 Tips issues explained that would improve the code's usability or efficiency upon modification.

Severity	Issue	Status
Tips	Potential Storage Layout Collision Risk due to ReentrancyGuard in AssimilatorV2	(Resolved - v1.0)
Tips	Potential Read-only Reentrancy Vulnerability during LP Token Redemption	(Resolved - v1.0)
Critical	Loss of User or Pool Funds Due to Incorrect Amount during Wrapped Native Token Refunds	(Resolved - v1.0)
Low	No Consideration for FoT (Fee-on-Transfer) Tokens in Router Contract	(Resolved - v1.0)
Critical	Pool Fund Drainage Risk due to Rounding Error in targetSwap	(Resolved - v1.0)
Critical	Fund Loss for Liquidity Providers due to a wrong truncation for bug fix	(Resolved - v1.0)
Critical	Malicious Oracle Address Registration Risk within Curve Contract	(Acknowledged - v1.0)
High	Potential Loss Risk in Output Token Transfer Functions	(Resolved - v1.0)
High	balanceOf() in the loop leads to stealing the LP providers' tokens	(Resolved - v1.1)

OVERVIEW

Protocol overview

DFX Finance is a decentralized protocol for swapping assets like Uniswap and others.

In previous versions, the design was primarily focused on stable Pair swaps, but now it also supports swaps for Non-Stable Pairs. The LP pools consist of two tokens: the base token and the quote token. Using one or two of these pools and the Router contract, the user can swap between two tokens as they want.

The swap amounts are determined by the logic in Shell Protocol v1, which is a protocol for USD stablecoins (or, in general, assets pegged to the same token). Since DFX Finance utilizes various tokens, they do not use the Shell Protocol's formulas directly. To account for the values of each token appropriately, they use a price oracle from Chainlink. Shell Protocol's logic is used after normalizing the token amounts into numeraires.

We noted some various facts on Shell Protocol in our previous report's commentary section.

(https://github.com/dfx-finance/protocol-v2/blob/main/audits/2023-02-07-DFXv2-Kalos-Audit.pdf)

Another distinction with the Shell Protocol is that the deposit/withdrawal methods are more strict. In this version, the liquidity deposit and withdrawal are done so that the token's amounts are proportional to the current LP pool's balances. To make such deposits and withdrawals easier, a zapper contract can be used to do the necessary swaps at once.

Unlike the previous version, the updated contracts now allow any ERC20 token as a quote currency, facilitate swaps and liquidity operations using Native Token, centralize contract ownership in the config contract, and support Fees on Transfer tokens.

Scope

src

```
├── AssimilatorFactory.sol
├── Assimilators.sol
├── Config.sol
├── Curve.sol
├── CurveFactoryV2.sol
├── CurveMath.sol
├── Orchestrator.sol
├── ProportionalLiquidity.sol
├── Router.sol
├── Storage.sol
├── Structs.sol
├── Swaps.sol
├── ViewLiquidity.sol
├── Zap.sol
├── assimilators
│   └── AssimilatorV2.sol
├── interfaces
│   ├── IAssimilator.sol
│   ├── IAssimilatorFactory.sol
│   ├── IConfig.sol
│   ├── ICurve.sol
│   ├── ICurveFactory.sol
│   ├── IERC20Detailed.sol
│   ├── IFlashCallback.sol
│   ├── IOracle.sol
│   └── IWeth.sol
└── lib
    ├── ABDKMath64x64.sol
    ├── ABDKMathQuad.sol
    ├── FullMath.sol
    ├── NoDelegateCall.sol
    └── UnsafeMath64x64.sol
```

Access Controls

The contracts have the two following access control modifiers.

- ❖ onlyOwner()
- ❖ onlyCurveFactoryOrOwner()

onlyOwner() : The owner has a control over the curve's parameters, as well as safety measures such as pausing, freezing, pool caps, and pool guards.

- AssimilatorFactory.sol#setCurveFactory()
- AssimilatorFactory.sol#revokeAssimilator()
- Config.sol#setGlobalFrozen()
- Config.sol#toggleGlobalGuarded()
- Config.sol#setPoolGuarded()
- Config.sol#setGlobalGuardAmount()
- Config.sol#setPoolCap()
- Config.sol#setPoolGuardAmount()
- Config.sol#updateProtocolTreasury()
- Config.sol#updateProtocolFee()
- Curve.sol#setParams()
- Curve.sol#setAssimilator()
- Curve.sol#excludeDerivative()
- Curve.sol#setEmergency()
- Curve.sol#setFrozen()
- Curve.sol#transferOwnership()

onlyCurveFactoryOrOwner(): This is used only for setting up new Assimilators in AssimilatorFactory. Not only the CurveFactory, but the owner set by the curve parameter can also access.

- AssimilatorFactory.sol#newAssimilator()

The Owner can update crucial parameters related to swaps in DFX Finance, and this could cause risks that could potentially harm users.

FINDINGS

1. Potential Storage Layout Collision Risk due to ReentrancyGuard in AssimilatorV2

ID: DFX-2-01
Type: Storage Collision
File: src/assimilators/AssimilatorV2.sol

Severity: Tips
Difficulty: N/A

Issue

Previous AssimilatorV2 does not inherit ReentrancyGuard and does not modify the storage variables. There was no concern related to Storage Layout Collision.

```
contract AssimilatorV2 is IAssimilator, ReentrancyGuard {
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/assimilators/AssimilatorV2.sol#L28]

However, with this update, AssimilatorV2 Contract now inherits ReentrancyGuard.

```
        (bool _success, bytes memory returnData_) = _callee.delegatecall(_data);
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/Assimilators.sol#L35]

As AssimilatorV2's logic is called by delegatecall from the Curve contract, it introduces the possibility of encountering that issue. Although there are currently no functions using the nonReentrant modifier and no substantial issues, this issue should be fixed for the potential risk from the further update.

Recommendation

To remove the ReentrancyGuard from AssimilatorV2 and use it in the contract that adopts the logic of AssimilatorV2.

Fix Comment

This issue has been fixed by deleting the ReentrancyGuard .

2. Potential Read-only Reentrancy Vulnerability during LP Token Redemption

ID: DFX-2-02
Severity: Tips
Type: Reentrancy
Difficulty: Low
File: src/ProportionalLiquidity.sol

Issue

```
 function outputNumeraire(
    address _dst,
    int128 _amount,
    bool _toETH
) external payable override returns (uint256 amount_) {
    uint256 _rate = getRate();

    amount_ =
        (_amount.mulu(10 ** tokenDecimals) * 10 ** oracleDecimals) /
        _rate;
    if (_toETH) {
        IWETH(wETH).withdraw(amount_);
        (bool success, ) = payable(_dst).call{value: amount_}(""); // reentrancy here
        require(success, "Assimilator/Transfer ETH Failed");
    } else {
        token.safeTransfer(_dst, amount_);
    }
}
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/assimilators/AssimilatorV2.sol#L241-L258]

```
function proportionalWithdraw(
    Storage.Curve storage curve,
    uint256 _withdrawal,
    bool _toETH
) external returns (uint256[] memory) {
    uint256 _length = curve.assets.length;
    // ...
    for (uint256 i = 0; i < _length; i++) {
        if (
            _toETH &&
            (IAssimilator(curve.assets[i].addr).underlyingToken() ==
                IAssimilator(curve.assets[i].addr).getWeth())
        ) {
            withdrawals_[i] = Assimilators.outputNumeraire(
                curve.assets[i].addr,
                msg.sender,
```

```
                    _oBals[i].mul(_multiplier),
                    true
                );
        // ...
        }

        burn(curve, msg.sender, _withdrawal); // does not update yet

        return withdrawals_;
    }
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/ProportionalLiquidity.sol#L152-L192]

The code that withdraws deposited tokens can trigger the fallback function of the recipient when receiving Native Tokens. This scenario introduces the risk of Read-only Reentrancy vulnerability, as the assets are received before the LP Tokens are burned. This could lead to exploitable vulnerabilities like LP tokens' price manipulation of derivative financial products utilizing DFX Finance's LP Tokens.

Recommendation

Modify the `notEntered` variable (used for the reentrancy check) to a public visibility. Notify third-party developers through official documentation regarding the potential for read-only reentrancy events. Third-party developers can mitigate read-only reentrancy vulnerabilities by ensuring that the `Curve` Contract is active (i.e., `notEntered` is true) when evaluating the LP Token Valuation. We conclude that fully preventing this within the DFX Finance code is challenging. Notably, other platforms like `Curve` Finance and Balancer, where read-only reentrancy occurred, were unable to entirely prevent it. Their approach was to educate third-party developers about the possibility of read-only reentrancy and provide guidance on its mitigation.

Fix Comment

`notEntered` has been changed to public and can be read by external contracts.

3. Loss of User or Pool Funds Due to Incorrect Amount during Wrapped Native Token Refunds

ID: DFX-2-03
Severity: Critical
Type: Arithmetic
Difficulty: Low
File: src/Zap.sol

Issue

The depositETH() function wraps the received native tokens from users and transfers them to users, depositing them into the pool. After the depositing process, the remaining wrapped native tokens are received from the user, unwrapped, and refunded to the user. However, currently, when receiving the remaining wrapped native tokens from the user, it receives an incorrect amount from the user, and this results in a loss of the user's funds

In the code below, despite unwrapping the wrapped native tokens by the remainder amount, tokens are received from the user not based on the remainder amount, but on msg.value - deposits_[0].

```
    function depositETH(
        uint256 _deposit,
        uint256 _minQuoteAmount,
        uint256 _minBaseAmount,
        uint256 _maxQuoteAmount,
        uint256 _maxBaseAmount,
        uint256 _deadline
    )
        external
        payable
        // ...
        returns (uint256, uint256[] memory)
    {
        require(_deposit > 0, "Curve/deposit_below_zero");

        // (curvesMinted_,  deposits_)
        IWETH(wETH).deposit{value: msg.value}();
        IERC20(wETH).safeTransferFrom(address(this), msg.sender, msg.value);

        // ...
        ) = ProportionalLiquidity.proportionalDeposit(curve, _depositData);

        uint256 remainder = 0;
        if (IAssimilator(curve.assets[0].addr).underlyingToken() == wETH) {
            remainder = msg.value - deposits_[0];
        } else if (
            IAssimilator(curve.assets[1].addr).underlyingToken() == wETH
```

```
        ) {
            remainder = msg.value - deposits_[1];
        } else {
            revert("reverted here");
        }
        // now need to determine which is wETH
        if (remainder > 0) {
            IERC20(wETH).safeTransferFrom(
                msg.sender,
                address(this),
                msg.value - deposits_[0] // should be remainder
            );
            IWETH(wETH).withdraw(remainder);
            (bool success, ) = msg.sender.call{value: remainder}("");
            require(success, "Curve/ETH transfer failed");
        }
        return (curvesMinted_, deposits_);
    }
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/Curve.sol#L741-L798]

When `msg.value - deposits_[0]` is less than the `remainder`, pool funds would be lost. When it is greater, then the user's funds would be lost.

Recommendation

The code should be modified to use `remainder` instead of `msg.value - deposits_[0]`.

Fix Comment

This issue has been fixed.

4. No Consideration for FoT (Fee-on-Transfer) Tokens in OriginSwap

ID: DFX-2-04
Severity: Low
Type: Logic error/bug
Difficulty: Low
File: src/{Router, Curve}.sol

Issue

The Router Contract is designed to optimize Multi-Hop Swaps by combining multiple Curve Contracts. Within the code of the originSwapFromETH() and originSwap() functions in the Router Contract, the Curve's originSwap() function is utilized. It's important to note that this originSwap() function doesn't return the actual quantity of tokens transferred but an internally calculated output. The mentioned function becomes a concern when dealing with tokens that impose a fee upon invoking the transfer() function. When transferring the output obtained via Curve contract's originSwap() function, there might be an insufficient quantity, resulting in a revert if the token applies a fee. To address this potential issue, it's crucial to consider using the actual balance remaining in the Router Contract as the targetAmount instead of relying on the return value of originSwap() function.

```
// Router.sol
function originSwap(
        uint256 _originAmount,
        uint256 _minTargetAmount,
        address[] memory _path,
        uint256 _deadline
    ) public returns (uint256 targetAmount_) {
        uint256 pathLen = _path.length;
        address origin = _path[0];
        address target = _path[pathLen - 1];
        IERC20(origin).safeTransferFrom(
            msg.sender,
            address(this),
            _originAmount
        );
        for (uint i = 0; i < pathLen - 1; ++i) {
            address payable curve = CurveFactoryV2(factory).getCurve(
                _path[i],
                _path[i + 1]
            );
            uint256 originBalance = IERC20(_path[i]).balanceOf(address(this));
            IERC20(_path[i]).safeApprove(curve, originBalance);
            targetAmount_ = Curve(curve).originSwap(
                _path[i],
                _path[i + 1],
                originBalance,
                0,
```

```
                _deadline
            );
        }
        require(targetAmount_ >= _minTargetAmount, "Router/originswap-failure");
        IERC20(target).safeTransfer(msg.sender, targetAmount_);
    }

    function originSwapFromETH(
        uint256 _minTargetAmount,
        address[] memory _path,
        uint256 _deadline
    ) public payable returns (uint256 targetAmount_) {
        // wrap ETH to WETH
        IWETH(_wETH).deposit{value: msg.value}();
        uint256 pathLen = _path.length;
        address origin = _path[0];
        require(origin == _wETH, "router/invalid-path");
        address target = _path[pathLen - 1];
        for (uint i = 0; i < pathLen - 1; ++i) {
            address payable curve = CurveFactoryV2(factory).getCurve(
                _path[i],
                _path[i + 1]
            );
            uint256 originBalance = IERC20(_path[i]).balanceOf(address(this));
            IERC20(_path[i]).safeApprove(curve, originBalance);
            targetAmount_ = Curve(curve).originSwap(
                _path[i],
                _path[i + 1],
                originBalance,
                0,
                _deadline
            );
        }
        require(
            targetAmount_ >= _minTargetAmount,
            "Router/originswap-from-ETH-failure"
        );
        IERC20(target).safeTransfer(msg.sender, targetAmount_);
    }
```

[https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Router.sol]

Calling the originSwap function directly from the Curve contract without going through the Router contract can also cause problems. The code of the OriginSwap function in the Curve Contract is also relevant to this issue.

```
function originSwap(
    address _origin,
    address _target,
    uint256 _originAmount,
    uint256 _minTargetAmount,
    uint256 _deadline
)
    external
```

```
    deadline(_deadline)
    globallyTransactable
    transactable
    noDelegateCall
    isNotEmergency
    nonReentrant
    returns (uint256 targetAmount_)
{
    OriginSwapData memory _swapData;
    _swapData._origin = _origin;
    _swapData._target = _target;
    _swapData._originAmount = _originAmount;
    _swapData._recipient = msg.sender;
    _swapData._curveFactory = curveFactory;
    targetAmount_ = Swaps.originSwap(curve, _swapData, false);

    require(
        targetAmount_ >= _minTargetAmount,
        "Curve/below-min-target-amount"
    );
}
```

[https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Curve.sol]

Recommendation

It is recommended to compare the exact remaining balance in the Router Contract, once all desired user-initiated swaps have concluded, with the `_minTargetAmount`, and pass the balance as a parameter to the `safeTransfer()` function.
It is also recommended to compare the exact target token balance with the _minTargetAmount when the originSwap() function of the curve is almost finished.

Fix Comment

This issue has been fixed.

5. Pool Fund Drainage Risk due to Rounding Error during targetSwap

ID: DFX-2-05
Severity: Critical
Type: Arithmetic
Difficulty: Low
File: src/assimilators/AssimilatorV2.sol

Issue

The intakeNumeraire() function of the AssimilatorV2 contract receives tokens from users. To calculate the amount of tokens to receive, denoted as amount_, it multiplies 10 ** tokenDecimals and 10 ** oracleDecimals to the argument _amount and divides it by _rate. However, a rounding-down error occurs during these multiplication and division operations.

```
    function intakeNumeraire(
        int128 _amount
    ) external payable override returns (uint256 amount_) {
        uint256 _rate = getRate();

        amount_ =
            (_amount.mulu(10 ** tokenDecimals) * 10 ** oracleDecimals) /
            _rate;
        uint256 balanceBefore = token.balanceOf(address(this));

        token.safeTransferFrom(msg.sender, address(this), amount_);
        uint256 balanceAfter = token.balanceOf(address(this));
        uint256 diff = amount_ - (balanceAfter - balanceBefore);
        if (diff > 0) intakeMoreFromFoT(amount_, diff);
    }
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/assimilators/AssimilatorV2.sol#L125-L139]

For example, during mulu(), when tokenDecimals is 2 like EURS, the (0.001 wei).mulu(10 ** 2) results in 0 wei instead of 1 wei. This leads to a zero value to be divided by _rate. The amount_ becomes eventually the zero value.
In case of the division by _rate, due to the rounding-down error, the desired division result for amount_ is 0.99 wei, but it is calculated as 0 wei.
This vulnerability enables an attacker to swap tokens without source token payment. If the token being paid has a sufficiently high number of decimals, like 18, this might not be a concern. However, tokens like EURS with a decimal value of 2, resulting in low precision, can be drained from the EURS pool due to this issue. This risk applies not only to EURS tokens but also to tokens with low decimals like GUSD that might be used in the swap pool.

Recommendation

Change the precision of the multiplier for `mulu()` into the higher like `mulu(10 ** (tokenDecimals + oracleDecimals + 18))` and divide `rate * 1e18`.
Use rounding-up division in `intakeNumeraire()` when dividing `_amount` by `_rate`.
Also, to match the `viewRawAmount()` function with the output values of the `targetSwap()` function,
We recommend to modify the code of the `viewRawAmount()` function, as recommended for the `intakeNumeraire()` function.

Fix Comment

This issue has been fixed.

6. Potential Loss Risk in Output Token Transfer Functions

ID: DFX-2-06
Type: Arithmetic
File: src/assimilators/AssimilatorV2.sol

Severity: High
Difficulty: Low

Issue

In the functions `outputRawAndGetBalance()`, `outputRaw()`, and `outputNumeraire()`, there is a potential risk associated with token transfer outcomes during withdrawal and swap. In certain scenarios, the amount of tokens intended to be transferred to the receiver (or `_dst` in the given code) could result in a rounding-down error and a value of zero. Such an outcome would cause the asset loss of the intended receiver.

For example, in the `outputRawAndGetBalance` function, which is utilized in `targetSwap()` to transfer a specified amount of tokens, the concern is when the token's decimals is small and the token price from the oracle falls below one dollar. Due to these factors, the expression `(_amount * _rate)` might be less than `(10**oracleDecimals)`. Consequently, the entire division `((_amount * _rate) / 10 ** oracleDecimals)` could be zero.

Moreover, the Solidity and the Ethereum Virtual Machine do not provide for output in floating point. Thus, User Assets can be lost due to this feature.

Recommendation

To address this vulnerability and prevent potential loss for users, we recommend implementing a few changes to the codebase:

Add a require statement to the `outputRawAndGetBalance()`, `outputRaw()`, and `outputNumeraire()` functions to enforce the output amount is greater than zero.
Enforce the `Curve` Contract by adding a check in the `originSwap()`, `withdraw` and `emergencyWithdraw()` functions to check the minimum output amount.
Similarly, add the maximum input amount check to the targetSwap function,
Fix the `mulu()` operation in `outputNumeraire()` to be `.mulu(10 ** (tokenDecimals + oracleDecimals + 18)) / (_rate * 1e18)` instead of `.mulu(10 ** tokenDecimals)`

Fix Comment

This issue has been fixed.

7. Fund Loss for Liquidity Providers due to a wrong truncation for bug fix

ID: DFX-2-07
Severity: Critical
Type: Arithmetic
Difficulty: Low
File: src/ProportionalLiquidity.sol

Issue

The Curve contract mints LP tokens to liquidity providers when they deposit both Base Token and Quote Token.

The code added to prevent the Rounding Down Error in the original code is as follows:

```
// takes a numeraire amount, calculates the raw amount of eurs, transfers it in and returns
the corresponding raw amount
function intakeNumeraireLPRatio(
    uint256 _baseWeight,
    uint256 _minBaseAmount,
    uint256 _maxBaseAmount,
    uint256 _pairTokenWeight,
    uint256 _minpairTokenAmount,
    uint256 _maxpairTokenAmount,
    address _addr,
    int128 _amount
) external payable override returns (uint256 amount_) {
    ...
    amount_ = (_amount.mulu(10 ** tokenDecimals) * 1e6) / _rate;
    amount_ = amount_.add(1);
    ...
}
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/assimilators/AssimilatorV2.sol#L169]

it multiplies `10 ** tokenDecimals` and `10 ** oracleDecimals` to the argument `_amount` and divides it by `_rate`. However, a rounding-down error occurs during these multiplication and division operations.

For example, during `mulu()`, when `tokenDecimals` is 2 like EURS, the `(0.001 wei).mulu(10 ** 2)` results in 0 wei instead of 1 wei. This leads to a zero value to be divided by `_rate`. The `amount_` becomes eventually the zero value.

Thus, although 1 is later added, the deposited amount will inevitably be less than the amount initially intended to be deposited. However, the DFX Team thought a depositor was receiving more LP Tokens due to a miscalculation. It's not that the user gets more LP Tokens due to a miscalculation of the deposit amount, but rather that the user deposits less than the appropriate amount of tokens for the LP Tokens the user receives.

Due to the wrong bug fix mentioned earlier, the following truncation code was added to prevent vulnerabilities.

```
curves_ = curves_.div(1e17).mul(1e17)
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/ProportionalLiquidity.sol#L90]

This issue invalidates the mitigation against the ERC4626 inflation attack and raises the attack again.

To explain the side effects caused by the truncation code update, we've simplified the minting formula for clarity.

$$depositValue * (totalShells / normalizeLiquidity)$$

For example, the first depositor in the curve contract deposits a minimal amount of tokenA and tokenB, making the LP Token supply very small, and then transfers (donates) a significant amount to the curve contract later. If a victim deposits tokenA and tokenB later, the denominator in the above formula increases, resulting in an output of less than 1. Due to the code that truncates the output to below 1e17, the mint amount eventually becomes zero. As a result, the victim deposits tokenA and tokenB but doesn't receive any LP Tokens. Then, the attacker can withdraw all the victim's tokens and it leads to a loss of funds.

Recommendation

It is recommended to remove the `curves_ = curves_.div(1e17).mul(1e17)` code. While the DFX Team mentioned that this code was introduced to patch a previous vulnerability, removing it and ensuring proper rounding up in intakeNumeraireLPRatio will prevent the previous issue.
Below is the `intakeNumeraireLPRatio()` function we have fixed.

```
function intakeNumeraireLPRatio(
```

```
        uint256 _baseWeight,
        uint256 _minBaseAmount,
        uint256 _maxBaseAmount,
        uint256 _pairTokenWeight,
        uint256 _minpairTokenAmount,
        uint256 _maxpairTokenAmount,
        address _addr,
        int128 _amount
    ) external payable override returns (uint256 amount_) {
        ...
        amount_ = (_amount.mulu(10 ** tokenDecimals * 1e6)) / _rate;
        amount_ = amount_.add(1);
        ...
    }
```

Additionally, match the viewProportionalDeposit() function with the output values of the proportionalDeposit() function. We recommend modifying the code of the viewRawAmountLPRatio() function, as recommended for the intakeNumeraireLPRatio() function.

Fix Comment

This issue has been fixed by deleting the truncation.

8. Malicious Oracle Address Registration Risk within Curve Contract

ID: DFX-2-08
Type: Logic Error
File: src/CurveFactoryV2.sol

Severity: Critical
Difficulty: Low

Issue

In the Curve Contract's newCurve function, a potential security vulnerability was discovered concerning registering the malicious Oracle addresses. Specifically, the function allows for arbitrary registration of oracles through calling the newAssimilator(), which is a significant concern.

This function might allow a malicious actor to register an oracle that mimics Chainlink's interface but behaves maliciously.

A malicious actor could exploit this oversight to inject incorrect price data or engage in other malicious activities.

To protect against these threats, we emphasize the need for tighter controls on registering Oracle addresses.

Recommendation

We recommend taking the following measures to mitigate this risk:

1. Fetch the price feed contract addresses registered at https://data.chain.link/ethereum/mainnet and https://data.chain.link/polygon/mainnet off-chain.
2. Store and manage these fetched addresses in a separate registry contract.
3. When the newCurve() is called, enforce that only oracles listed in the registry contract are utilized.

In scenarios where Chainlink cannot provide prices, the DFX Team should consider creating a custom oracle that follows the Chainlink interface and then register it in the registry contract mentioned above. Implementing these steps will significantly reduce the risk of malicious oracles and enhance the system's overall security.

Fix Comment

The team replied that they will not employ whitelist oracles due to centralization and will not fix this issue. Therefore, they have to monitor the created pools that have malicious oracles and weird tokens like too low/high prices and decimals.

9. balanceOf() in the loop leads to stealing the LP providers' tokens

ID: DFX-2-09
Severity: High
Type: Logic Error
Difficulty: Low
File: src/assimilators/AssimilatorV2.sol

Issue

LP deposit doesn't work as intended since the increased token balance is used in the next iteration.

```
            for (uint256 i = 0; i < _length; i++) {
                IntakeNumLpRatioInfo memory info;
                info.baseWeight = _baseWeight;
                info.minBase = depositData.minBase;
                info.maxBase = depositData.maxBase;
                info.quoteWeight = _quoteWeight;
                info.minQuote = depositData.minQuote;
                info.maxQuote = depositData.maxQuote;
                info.amount = _oBals[i].mul(_multiplier).add(ONE_WEI);
                deposits_[i] = Assimilators.intakeNumeraireLPRatio(
                    curve.assets[i].addr,
                    info
                );
            }
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/ProportionalLiquidity.sol#L60-L73]

```
    function intakeNumeraireLPRatio(
        uint256 _baseWeight,
        uint256 _minBaseAmount,
        uint256 _maxBaseAmount,
        uint256 _pairTokenWeight,
        uint256 _minpairTokenAmount,
        uint256 _maxpairTokenAmount,
        address _addr,
        int128 _amount
    ) external payable override returns (uint256 amount_) {
        uint256 _tokenBal = token.balanceOf(_addr);

        if (_tokenBal <= 0) return 0;

        _tokenBal = _tokenBal.mul(10 ** (18 + pairTokenDecimals)).div(
            _baseWeight
        );

        uint256 _pairTokenBal = pairToken
            .balanceOf(_addr)
            .mul(10 ** (18 + tokenDecimals))
            .div(_pairTokenWeight);
```

```
        // Rate is in pair token decimals
        uint256 _rate = _pairTokenBal.mul(1e6).div(_tokenBal);

        amount_ = (_amount.mulu(10 ** tokenDecimals) * 1e6) / _rate;
```

[https://github.com/dfx-finance/protocol-v2/blob/acdc5fbe8e96f8c800c6f546a7469f513da14c1c/src/assimilators/AssimilatorV2.sol#L142-L168]

To get the amount of tokens to deposit when there already is some liquidity, `intakeNumeraireLPRatio()` is used. It can be shown via some computation that the amount of tokens to transfer correlates to `balance(_pairToken) / balance(_token)` of the LP.

This is problematic during the next iteration after the `_token` is transferred first. When computing the amount of the `_pairToken` to transfer, the computation will be done on the increased value of `balance(_token)`, which decreases the result. This means that the user depositing will receive the same number of LP tokens for less amount of `_token` currency than usual. By withdrawing the LP immediately, the user can effectively steal from the one who first deposited the tokens into the LP.

Recommendation

It is recommended to calculate `intakeNumeraireLPRatio()` based on the balance which is fetched before any actual token transfer.

Fix Comment

This issue has been fixed by passing the token balance as an argument to `intakeNumeraireLPRatio()`.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure codes, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	**Medium**	**High**
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	▪ Integer under/overflow vulnerability ▪ floating point and rounding accuracy
Access & Privilege Control	▪ Manager functions for emergency handle ▪ Crucial function and data access ▪ Count of calling important task, contract state change, intentional task delay
Denial of Service	▪ Unexpected revert handling ▪ Gas limit excess due to unpredictable implementation
Miner Manipulation	▪ Dependency on the block number or timestamp. ▪ Frontrunning
Reentrancy	▪Proper use of Check-Effect-Interact pattern. ▪Prevention of state change after external call ▪ Error handling and logging.
Low-level Call	▪ Code injection using delegatecall ▪ Inappropriate use of assembly code
Off-standard	▪ Deviate from standards that can be an obstacle of interoperability.
Input Validation	▪ Lack of validation on inputs.
Logic Error/Bug	▪ Unintended execution leads to error.
Documentation	▪Coherency between the documented spec and implementation
Visibility	▪ Variable and function visibility setting
Incorrect Interface	▪ Contract interface is properly implemented on code.

End of Document