DFX Finance

Security Assessment

May 3, 2021

Prepared For:
Kendrick Tan | *DFX Finance*
kendrick@dfx.finance

Henry Chan | *DFX Finance*
henry@dfx.finance

Kevin Zhang | *DFX Finance*
kevin@dfx.finance

Adrian Li | *DFX Finance*
adrian@dfx.finance

Prepared By:
Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Changelog:

May 3, 2020:	Initial report draft
May 19, 2021:	Added fix log
May 27, 2021:	Revised language

Executive Summary

From April 12 to April 30, 2021, DFX Finance engaged Trail of Bits to review the security of its smart contracts. Trail of Bits conducted this assessment working from the following repository and commit hash:

- Repository: `dfx-finance/protocol`
- Commit Hash: `906bd5274dcd07c458e6bbd6f13adced873ac952`

In the first week, we began to familiarize ourselves with the platform, employed [`slither`](slither) to perform static analysis across the codebase, and started triaging relevant output to guide our manual review. As part of our scoping process, we identified critical components and data flows, which helped us to plan out subsequent aspects of our assessment. Our manual review efforts included analysis of token/interface compliance, high-level arithmetic issues, the platform's adherence to best practices when performing low-level calls and use of Chainlink APIs, and general code correctness. This resulted in four findings ranging from low to informational severity and a fifth of undetermined severity.

During the second week, we expanded our review of the codebase, covering aspects such as swap-related arithmetic and data validation within the invariant enforcement functions, in addition to partially reviewing fee calculations and assessing assimilators for general code correctness. These efforts led to five additional findings: four ranging from high to informational severity and one of undetermined severity. We also began setting up a fuzzing harness, which we used to validate codebase properties during the next week.

In the final week, we deepened our review of arithmetic and the state machine, writing related property and unit tests and considering additional attack vectors such as front-running and compromise by external providers. This resulted in four additional findings ranging from high to low severity and two more of undetermined severity.

The DFX Finance smart contracts represent a work in progress with multiple planned iterations. Trail of Bits recommends that DFX Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Improve code readability, document the purposes of individual functions and pieces of code, and provide external documentation such as a whitepaper or a wiki. Simplifying the code would facilitate code reviews, verification of any expected invariants, and the safe deployment of DFX Finance smart contracts.

- Simplify code paths that contain redundant artifacts (specifically structures/variables) of DFX Finance's fork from the Shell Protocol. For example, data structures populated by `CurveFactory` when initializing a new `Curve` contain many redundant/duplicated entries. Similarly, the reasoning behind the adjustments made to `Curve` construction arguments may be unclear to external reviewers/users, which could cause confusion (TOB-DFX-015).

- Improve data validation within the `Curve` and `Router` constructions, as well as in code paths that interface with external contracts/tokens such as assimilator code and the `safeApprove` function (TOB-DFX-004, TOB-DFX-005, TOB-DFX-006, TOB-DFX-007, TOB-DFX-008).

- Expand the unit testing suite to cover (for example) additional `Curve` parameters, weights, and withdrawal, deposit, and swap scenarios. Ensure that the logic of each assimilator is tested to prevent arithmetic issues such as the return of raw values by assimilator balance functions (TOB-DFX-012) and invalid `Curve` parameters such as weight (TOB-DFX-015).

- If any of the abovementioned code paths are refactored, perform an additional internal review of the invariants.

Update: On May 19, 2021, Trail of Bits reviewed fixes implemented for the issues in this report. A detailed review of the current status of each issue is provided in Appendix C.

Project Dashboard

Application Summary

Name	DFX Finance
Version	`906bd5274dcd07c458e6bbd6f13adced873ac952`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	April 12 – April 30, 2021
Method	Whitebox
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	6	■■■■■■
Total Informational-Severity Issues	3	■■■
Total Undetermined-Severity Issues	4	■■■■
Total	16	

Category Breakdown

Configuration	1	■
Data Validation	11	■■■■■■■■■■■
Patching	1	■
Timing	1	■
Undefined Behavior	2	■■
Total	16	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory**. We did not find any serious access control issues, though we identified one issue involving the transfer of ownership (TOB-DFX-004).
Arithmetic	**Weak.** We found many minor issues related to arithmetic (TOB-DFX-002, TOB-DFX-010, TOB-DFX-011, TOB-DFX-014) and one critical issue stemming from the use of an incorrect value type (TOB-DFX-012).
Assembly Use	**Strong**. The system uses a minimal amount of assembly. Although one token compliance issue was identified (TOB-DFX-005), it does not currently have any practical impact on the codebase.
Centralization	**Moderate.** The owner of a `Curve` has significant influence over the system. The owner can set the parameters of the `Curve`, put it into an emergency mode that does not check for liquidity invariants during withdrawals, and even freeze the contract, disallowing swaps and new deposits.
Function Composition	**Weak.** There is a large amount of code duplication. For example, the only difference between `getOriginSwapData` and `getTargetSwapData` (and `viewOriginSwapData` and `viewTargetSwapData`) is one line of code out of more than 20 lines. Having all four functions pass in a boolean and switch on that particular line would improve their implementation. Furthermore, there are numerous code remnants from the original Shell Protocol (TOB-DFX-007). Lastly, there is unused code in the codebase, such as the large library `ABDKMathQuad.sol`.
Front-Running	**Satisfactory**. We found only one front-running issue, which would allow a Curve contract owner to freeze contracts to prevent specific deposits and swaps (TOB-DFX-016). A front-runner may be able to generate profits from the swap functionality, but the loss incurred by the user would be mitigated by the limit price. Due to its nature, the system allows for arbitrage opportunities; documentation regarding those opportunities would be beneficial to users.
Monitoring	**Satisfactory.** The contracts use a deprecated Chainlink API. However, the contracts emit events where appropriate.
Specification	**Moderate.** There is no whitepaper for the project, and the whitepaper for the original Shell Protocol diverges from the DFX

	Finance implementation in many ways. For example, it states that the gain function is the sum of all token balances, when in DFX Finance, it is the sum of their numeraire values. However, various functions have code comments and natural specifications.
Testing & Verification	**Weak.** The project has relatively extensive integration testing but lacks unit and fuzz tests. More thorough end-to-end coverage may catch critical issues such as TOB-DFX-012.
Upgradeability	**Not applicable.** The system cannot be upgraded.

Engagement Goals

The engagement was scoped to provide a security assessment of the DFX Finance smart contracts.

Specifically, we sought to answer the following non-exhaustive list of questions:

- Can `Curves` be constructed with parameters that cause trapped state transitions or arithmetic flaws?
- Are the assimilators generally sound? Do they update balances and handle transfers appropriately?
- Is the swap arithmetic correct? Is it susceptible to rounding errors or underflows/overflows?
- Are interactions with external systems such as the Chainlink API sound?
- Does the system properly calculate fees?
- Are there appropriate access controls on critical functions?
- Is the system vulnerable to re-entrancy attacks?
- Is it possible to front-run transactions to negatively affect the system?

Coverage

This section highlights some of the analysis coverage we achieved based on our high-level engagement goals. Our approaches and their results include the following:

- A review of `Curve` deployment parameters revealed unintuitive/undocumented behavior in `CurveFactory.newCurve`, which may return an existing `Curve` rather than a new one (TOB-DFX-003); unorthodox array parameters (TOB-DFX-004); and CurveFactory's incorrect initialization of `_derivativeAssimilators` (TOB-DFX-005).
- Analysis of the assimilators identified the use of deprecated Chainlink APIs (TOB-DFX-001), incorrect assimilator implementations that could lead to accounting errors (TOB-DFX-012), and an improper assumption (i.e., the USDC assimilator assumes that the value of USDC is always 1 USD) (TOB-DFX-013).
- A review of the use of access modifiers did not reveal any concerns.
- Analysis of swap arithmetic and user/pool balances found that the weight values used in calculations could cause unexpected deviations in the amount of funds transferred during deposits (TOB-DFX-015).
- Validation of external interactions did not reveal concerns regarding re-entrancy attacks; however, improving the data validation mechanisms and compliance with external contracts would help prevent future errors (TOB-DFX-008, TOB-DFX-009).
- A review of events that should be emitted during critical operations did not yield any concerns.

- An audit of functions focused on front-running opportunities did not reveal any critical concerns, although a contract owner could front-run with a call to the `setFrozen` function to deny certain deposits/swaps (TOB-DFX-016).

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Unless there is a reason not to, update `tests/Constants.ts` to the latest versions of the Chainlink oracle contracts and the assimilators and `IOracle` to the latest version of the Chainlink price feed API.** TOB-DFX-001

❑ **Unless they are intended to be strict, make the inequalities in the `require` statements non-strict.** Alternatively, consider refactoring the variables or providing additional documentation to convey that they are meant to be exclusive bounds. TOB-DFX-002

❑ **Consider rewriting `newCurve` such that it reverts in the event that a base-and-quote-currency pair already exists.** A `view` function can be used to check for and retrieve existing `Curves` without any gas payment prior to an attempt at `Curve` creation. TOB-DFX-003

❑ **Consider adding zero-address checks to the `Router`'s constructor and `Curve`'s `transferOwnership` function to prevent operator errors.** TOB-DFX-004

❑ **Consider adding contract existence checks to the `delegate` function to prevent future errors in the event that these code paths change.** TOB-DFX-005

❑ **Add data validation to ensure that the length of `_assets`, `_assetWeights`, and `_derivativeAssimilators` are proportionate.** Additionally, ensure that a `Curve` can be initialized only with the expected `curve.assets` and that `curve.assimilators` entries created by `includeAsset` are consistent with those set by `includeAssimilator`. TOB-DFX-006

❑ **Refactor `Curve.initialize` so that it properly validates parameters and succeeds only if the provided data is valid.** This will prevent operator errors. TOB-DFX-007

❑ **Leverage OpenZeppelin's `safeApprove` function wherever possible.** TOB-DFX-008

❑ **Implement `symbol`, `name`, and `decimals` on `Curve` contracts.** TOB-DFX-009

❑ **Rewrite the `if` statement such that it does not use and assign the same variable in an equality check.** TOB-DFX-010

❑ **Change all instances of `us_mul` and `us_div` to `ABDKMath64x64.mul` and `.div`, respectively, which also operate on two fixed-point numbers but have overflow/underflow protections.** TOB-DFX-011

❑ **Change the semantics of the three functions listed above in the CADC, XSGD, and EURS assimilators to return the numeraire balance.** TOB-DFX-012

❑ **Replace the hard-coded integer literal in the `UsdcToUsdAssimilator`'s `getRate` method with a call to the relevant Chainlink oracle, as is done in other assimilator contracts.** TOB-DFX-013

❑ **Consider rounding up when determining the amount of tokens required in a deposit.** TOB-DFX-014

❑ **Refactor the weight-related code to ensure that the correct amount of assets is transferred when `deposit` is called.** TOB-DFX-015

❑ **Consider rewriting `setFrozen` such that any contract freeze will not last long enough for a malicious user to easily execute an attack.** Alternatively, depending on the intended use of this function, consider implementing permanent freezes. TOB-DFX-016

Long Term

❑ **Use the latest stable versions of any external libraries or contracts leveraged by the codebase.** TOB-DFX-001

❑ **Ensure that mathematical terms such as "minimum," "at least," and "at most" are used in the typical way—that is, to describe values inclusive of minimums or maximums (as relevant).** TOB-DFX-002

❑ **Review state variables which referencing contracts to ensure that the code that sets the state variables performs zero-address checks where necessary.** TOB-DFX-004

❑ **Be mindful of the need for contract existence checks in low-level calls to external contracts.** TOB-DFX-005

❑ **Review all code paths that process contract construction arguments to prevent contracts from being deployed in invalid states.** If these issues are overlooked during deployment, they can cause undefined behavior in the future. TOB-DFX-006, TOB-DFX-007

❑ **Ensure that all low-level calls have accompanying contract existence checks and return value checks where appropriate.** TOB-DFX-008

❑ **Ensure that contracts conform to all required and recommended industry standards.** TOB-DFX-009

❑ **Ensure that the codebase does not contain undefined Solidity or EVM behavior.** TOB-DFX-010

❑ **Review all critical arithmetic to ensure that it accounts for underflows, overflows, and the loss of precision.** Consider using `SafeMath` and the safe functions of `ABDKMath64x64` where possible to prevent underflows and overflows. TOB-DFX-011

❑ **Use unit tests and fuzzing to ensure that all calculations return the expected values.** Additionally, ensure that changes to the Shell Protocol do not introduce bugs such as this one. TOB-DFX-012

❑ **Ensure that the system is robust against a decrease in the price of any stablecoin.** TOB-DFX-013

❑ **Document and evaluate the effects of rounding errors in deposit and withdrawal operations, and account for scaling.** TOB-DFX-014

Findings Summary

#	Title	Type	Severity
1	Assimilators use a deprecated Chainlink API	Patching	Undetermined
2	`_min*` and `_max*` have unorthodox semantics	Data Validation	Low
3	`CurveFactory.newCurve` returns existing curves without provided arguments	Undefined Behavior	Low
4	Missing zero-address checks in `Curve.transferOwnership` and `Router.constructor`	Data Validation	Low
5	Missing contract existence checks prior to `delegatecall`	Data Validation	Informational
6	Unorthodox `Curve` constructions may cause undefined behavior	Data Validation	Undetermined
7	`CurveFactory.newCurve` fails to initialize `_derivativeAssimilators`	Data Validation	Informational
8	`safeApprove` does not check return values for `approve` call	Data Validation	Low
9	ERC20 token `Curve` does not implement `symbol`, `name`, or `decimals` method	Configuration	Informational
10	Use of undefined behavior in equality check	Undefined Behavior	High
11	Insufficient use of `SafeMath`	Data Validation	Undetermined
12	Assimilators' balance functions return raw values	Data Validation	High
13	System always assumes USDC is equivalent to USD	Data Validation	Medium

14	Division operations do not consider rounding direction	Data Validation	Undetermined
15	Curve parameters are set incorrectly	Data Validation	Low
16	`setFrozen` can be front-run to deny deposits/swaps	Timing	Low

1. Assimilators use a deprecated Chainlink API

Severity: Undetermined | Difficulty: High
Type: Patching | Finding ID: TOB-DFX-001
Target: `tests/Constants.ts`, `assimilators/*`, `IOracle.sol`

Description

The old version of the Chainlink price feed API (`AggregatorInterface`) is used throughout the contracts and tests. For example, the deprecated function `latestAnswer` is used:

```
function getRate() public view override returns (uint256) {
    return uint256(oracle.latestAnswer());
}
```

Figure 1.1: `CadcToUsdAssimilator.sol#L39-L41`

This function is not present in the latest API reference (`AggregatorInterfaceV3`). However, it is present in the deprecated API reference.

Exploit Scenario

We are still investigating potential exploit scenarios. In the worst-case scenario, the deprecated contract could cease to report the latest values, which would very likely cause liquidity providers to incur losses.

Recommendations

Short term, unless there is a reason not to, update `tests/Constants.ts` to the latest versions of the Chainlink oracle contracts and the assimilators and `IOracle` to the latest version of the Chainlink price feed API.

Long term, use the latest stable versions of any external libraries or contracts leveraged by the codebase.

2. `_min*` and `_max*` have unorthodox semantics

Severity: Low
Type: Data Validation
Target: `Curve.sol`

Difficulty: Low
Finding ID: TOB-DFX-002

Description

Throughout the `Curve` contract, `_minTargetAmount` and `_maxOriginAmount` are used as open ranges (i.e., ranges that exclude the value itself), such as in the last line in `originSwap`:

```
    function originSwap(
        address _origin,
        address _target,
        uint256 _originAmount,
        uint256 _minTargetAmount,
        uint256 _deadline
    ) external deadline(_deadline) transactable nonReentrant returns (uint256 targetAmount_)
{
        targetAmount_ = Swaps.originSwap(curve, _origin, _target, _originAmount,
msg.sender);

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

Figure 2.1: `Curve.sol#L401-L411`

This contravenes the standard meanings of the terms “minimum” and “maximum,” which are generally used to describe closed ranges.

Exploit Scenario

Alice calls `viewOriginSwap`, gets the `targetAmount_`, and atomically calls `originSwap`, passing in the `targetAmount_`. She expects the call to succeed, but it reverts, which could cause unexpected behavior.

Recommendations

Short term, unless they are intended to be strict, make the inequalities in the `require` statements non-strict. Alternatively, consider refactoring the variables or providing additional documentation to convey that they are meant to be exclusive bounds.

Long term, ensure that mathematical terms such as “minimum,” “at least,” and “at most” are used in the typical way—that is, to describe values inclusive of minimums or maximums (as relevant).

3. `CurveFactory.newCurve` returns existing `curves` without provided arguments

Severity: Low
Type: Undefined Behavior
Target: `contracts/CurveFactory.sol`

Difficulty: Low
Finding ID: TOB-DFX-003

Description

`CurveFactory.newCurve` takes values and creates a `Curve` contract instance for each `_baseCurrency` and `_quoteCurrency` pair, populating the `Curve` with provided weights and assimilator contract references. However, if the pair already exists, the existing `Curve` will be returned without any indication that it is not a newly created `Curve` with the provided weights.

```
function newCurve(
    address _baseCurrency,
    address _quoteCurrency,
    uint256 _baseWeight,
    uint256 _quoteWeight,
    address _baseAssimilator,
    address _quoteAssimilator
) public onlyOwner returns (Curve) {
    bytes32 curveId = keccak256(abi.encode(_baseCurrency, _quoteCurrency));
    if (curves[curveId] != address(0)) {
        return Curve(curves[curveId]);
    }

[...]
```

Figure 3.1: CurveFactory will return existing `Curves` for a base-and-quote-currency pair even if other desired properties differ. (`contracts/CurveFactory.sol#L31-L42`)

If an operator attempts to create a new `Curve` for a base-and-quote-currency pair that already exists, `CurveFactory` will return the existing `Curve` instance regardless of whether other creation parameters differ. A naive operator may overlook this issue.

Note that this function is triggered by the contract owner, and the testnet seems to be deployed with hard-coded `Curve` values in `scripts/testnet/scaffold.ts`. As such, `CurveFactory`'s behavior is unlikely to cause problems in practice, though future changes may introduce an issue.

Exploit Scenario

Alice is an operator of DFX Finance contracts. When using `CurveFactory` to deploy a new `Curve`, Alice creates a `Curve` for a base-and-quote-currency pair that already exists but has different weights and assimilator references. Unbeknownst to her, `newCurve` returns the existing `Curve`. Alice then begins operating on a `Curve` that does not represent the parameters she supplied to `newCurve`.

Recommendations

Short term, consider rewriting `newCurve` such that it reverts in the event that a base-and-quote-currency pair already exists. A `view` function can be used to check for and retrieve existing `Curve`s without any gas payment prior to an attempt at `Curve` creation.

4. Missing zero-address checks in `Curve.transferOwnership` and `Router.constructor`

Severity: Low
Type: Data Validation
Target: `contracts/{Curve.sol, Router.sol}`

Difficulty: Low
Finding ID: TOB-DFX-004

Description
Like other similar functions, `Curve._transfer` and `Orchestrator.includeAsset` perform zero-address checks. However, `Curve.transferOwnership` and the `Router` constructor do not.

This may make sense for `Curve.transferOwnership`, because without zero-address checks, the function may serve as a means of burning ownership. However, popular contracts that define similar functions often consider this case, such as [OpenZeppelin's Ownable contracts](). Conversely, a zero-address check should be added to the `Router` constructor to prevent the deployment of an invalid `Router`, which would revert upon a call to the zero address.

```
constructor(address _factory) {
    factory = _factory;
}
```

Figure 4.1: `Router`'s constructor does not perform zero-address checks.
(`contracts/Router.sol#L33-L35`)

Exploit Scenario
Alice, an operator of DFX Finance contracts, authors an external tool to manage the contracts. However, because of a bug, when she calls `Curve`'s `transferOwnership` function, her tool does not set the address; instead, it passes in a zero address for the ownership transfer. Because of the lack of zero-address checks, this results in ownership burning.

Recommendations
Short term, consider adding zero-address checks to the `Router`'s constructor and `Curve`'s `transferOwnership` function to prevent operator errors.

Long term, review state variables which referencing contracts to ensure that the code that sets the state variables performs zero-address checks where necessary.

5. Missing contract existence checks prior to `delegatecall`

Severity: Informational
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFX-005
Target: `contracts/Assimilators.sol`

Description

The `Assimilators` library contains a function, `delegate`, that performs a `delegatecall` to a designated address with the provided data or reverts in the event that the delegate call fails. However, this function does not confirm the existence of a contract before calling it. `delegatecall` will always return success, even if it calls a non-existent contract (one with empty bytecode).

The relevant code is shown below:

```
function delegate(address _callee, bytes memory _data) internal returns (bytes memory) {
    // solhint-disable-next-line
    (bool _success, bytes memory returnData_) = _callee.delegatecall(_data);

    // solhint-disable-next-line
    assembly {
        if eq(_success, 0) {
            revert(add(returnData_, 0x20), returndatasize())
        }
    }

    return returnData_;
}
```

Figure 5.1: The Assimilators library does not perform a contract existence check before a delegate call. ([contracts/Assimilators.sol#L26-L38]())

All subsequent uses of this function are passed to `abi.decode`, resulting in a revert. However, if code paths are changed and an assimilator contract is destroyed, or if an invalid assimilator contract reference is used, this may result in undefined behavior.

Recommendations

Short term, consider adding contract existence checks to the `delegate` function to prevent future errors in the event that these code paths change.

Long term, review all contract code that performs low-level calls to ensure that contract existence checks are conducted before those calls where necessary.

6. Unorthodox `Curve` constructions may cause undefined behavior

Severity: Undetermined
Type: Data Validation
Target: `contracts/{Curve.sol, Orchestrator.sol}`

Difficulty: Low
Finding ID: TOB-DFX-006

Description

When initializing a `Curve`, an operator should typically account for base and quote currencies, which will result in a `curve.assets` length of two. However, when constructing a `Curve` directly, the `_assets` and `_assetWeights` parameters can contain more elements than a `CurveFactory` would normally provide in construction.

These additional elements would likely result in undefined behavior, as functions such as `proportionalWithdraw` and `proportionalDeposit` loop through and consider all `curve.assets`; other functions, such as `getFee`, consider only the first two elements of `curve.assets` during fee calculations.

```
function getFee(Storage.Curve storage curve) private view returns (int128 fee_) {
     int128 _gLiq;

     // Always pairs
     int128[] memory _bals = new int128[](2);

     for (uint256 i = 0; i < _bals.length; i++) {
          int128 _bal = Assimilators.viewNumeraireBalance(curve.assets[i].addr);

[...]
```

Figure 6.1: The `getFee` function considers only the first two elements of `curve.assets`, while other functions consider all elements when performing calculations. *(`contracts/Orchestrator.sol#L80-L95`)*

Similarly, the proportionality of `_assets`, `_assetWeights`, and `_derivativeAssimilators` is not validated. If the amount of `_assets` is more than five times the amount of `_assetWeights`, some elements of`_assets` may be silently ignored. Additionally, if the number of `_derivativeAssimilators` is not divisible by five, some entries may not be considered.

```
    function initialize(
        [...]
    ) external {
        for (uint256 i = 0; i < _assetWeights.length; i++) {
            uint256 ix = i * 5;

            [...]

            includeAsset(
```

```
                curve,
                _assets[ix], // numeraire
                _assets[1 + ix], // numeraire assimilator
                _assets[2 + ix], // reserve
                _assets[3 + ix], // reserve assimilator
                _assets[4 + ix], // reserve approve to
                _assetWeights[i]
            );
        }

        for (uint256 i = 0; i < _derivativeAssimilators.length / 5; i++) {
            uint256 ix = i * 5;

            [...]

            includeAssimilator(
                curve,
                _derivativeAssimilators[ix], // derivative
                _derivativeAssimilators[1 + ix], // numeraire
                _derivativeAssimilators[2 + ix], // reserve
                _derivativeAssimilators[3 + ix], // assimilator
                _derivativeAssimilators[4 + ix] // derivative approve to
            );
        }
    }
```

Figure 6.2: The `initialize` function performs appropriate data validation of all elements but argument array lengths. (contracts/Orchestrator.sol#L109-L152)

Also note that arguments can be passed such that `includeAssimilator` will overwrite data within the `curve.assimilators` entry created by earlier calls to `includeAsset`.

Recommendations

Short term, add data validation to ensure that the length of `_assets`, `_assetWeights`, and `_derivativeAssimilators` are proportionate. Additionally, ensure that a `Curve` can be initialized only with the expected `curve.assets` and that `curve.assimilators` entries created by `includeAsset` are consistent with those set by `includeAssimilator`.

Long term, review all code paths that process contract construction arguments to prevent contracts from being deployed in invalid states. If these issues are overlooked during deployment, they may cause undefined behavior in the future.

7. `CurveFactory.newCurve` fails to initialize `_derivativeAssimilators`

Severity: Informational
Type: Data Validation
Target: `contracts/{CurveFactory.sol, Curve.sol}`

Difficulty: Low
Finding ID: TOB-DFX-007

Description

`CurveFactory.newCurve(...)` passes a `_derivativeAssimilators` array with two elements to the `Curve` constructor when creating an instance of a `Curve`. However, as mentioned in [TOB-DFX-006](#), when the `Curve`'s constructor calls `Curve.initialize` to set all `Curve` parameters, it will assume that derivative assimilators will be represented by five array elements and will not account for these items.

Figure 7.1 shows two derivative assimilators passed as construction arguments.

```
address[] memory _derivativeAssimilators = new address[](2);

[...]

 // Assimilators
_derivativeAssimilators[0] = _baseAssimilator;
_derivativeAssimilators[1] = _quoteAssimilator;

// New curve
Curve curve = new Curve(_assets, _assetWeights, _derivativeAssimilators);
```

Figure 7.1: `CurveFactory.newCurve` constructs an array for derivative assimilators of size two. ([`contracts/CurveFactory.sol#L46-L71`](#))

When these arguments reach `Curve.initialize`, they will be ignored because of the bounds of the loop, which will never be satisfied.

```
for (uint256 i = 0; i < _derivativeAssimilators.length / 5; i++) {
    [...]
    includeAssimilator(
    [...]
```

Figure 7.2: `Curve.initialize` will process elements of `_derivativeAssimilators` only in blocks of five elements. ([`contracts/Orchestrator.sol#L138-L151`](#))

This issue appears unlikely to have a significant impact on the system, because these arguments should be included in the `includeAsset` function as part of `CurveFactory`'s argument construction.

Recommendations

Short term, refactor `Curve.initialize` so that it properly validates parameters and succeeds only if the provided data is valid. This will prevent operator errors.

Long term, review all code paths that process contract construction arguments to prevent contracts from being deployed in invalid states. If these issues are overlooked during deployment, they can cause undefined behavior in the future.

8. `safeApprove` does not check return values for `approve` call

Severity: Low
Type: Data Validation
Target: `contracts/Orchestrator.sol`

Difficulty: Low
Finding ID: TOB-DFX-008

Description

Although the `Router` contract uses OpenZeppelin's `SafeERC20` library to perform safe calls to ERC20's `approve` function, the `Orchestrator` library defines its own `safeApprove` function. This function checks that a call to `approve` was successful but does not check `returndata` to verify whether the call returned `true`.

```
    function safeApprove(
        address _token,
        address _spender,
        uint256 _value
    ) private {
        (bool success, ) =
            // solhint-disable-next-line
            _token.call(abi.encodeWithSignature("approve(address,uint256)", _spender,
_value));

        require(success, "SafeERC20: low-level call failed");
    }
```

Figure 8.1: `safeApprove` performs a low-level call to an ERC20 spec approve function but checks only the result of the low-level call attempt, not the result returned from a successful call. *(`contracts/Orchestrator.sol#L97-L107`)*

In contrast, OpenZeppelin's `safeApprove` function checks return values appropriately.

This issue may result in uncaught `approve` errors in successful `Curve` deployments, causing undefined behavior.

Recommendations

Short term, leverage OpenZeppelin's `safeApprove` function wherever possible.

Long term, ensure that all low-level calls have accompanying contract existence checks and return value checks where appropriate.

9. ERC20 token `Curve` does not implement `symbol`, `name`, or `decimals` method

Severity: Informational
Type: Configuration
Target: `contracts/Curve.sol`

Difficulty: Low
Finding ID: TOB-DFX-009

Description

`Curve.sol` is an ERC20 token and implements all six required ERC20 methods: `balanceOf`, `totalSupply`, `allowance`, `transfer`, `approve`, and `transferFrom`. However, it does not implement the optional but extremely common view methods `symbol`, `name`, and `decimals`.

The `symbol` and `name` methods do not exist in the contract or its dependencies. Although `decimals` is included in the library `Curves`, it is a `public pure` function and therefore will not be copied into `Curve`. In other words, the method will exist in the separately deployed library but not in the `Curve` contract.

```
    function decimals() public pure returns (uint8) {
        return 18;
    }
```

Figure 9.1: The abovementioned getter function for decimals. ([`Curve.sol#L67-L69`]())

Exploit Scenario

Alice knows that ERC20 contracts commonly implement the `symbol`, `name`, and `decimals` methods (even though they are not required) and assumes that `Curve` contracts do too. When she calls one of the methods in `Curve`, it reverts, which can lead to unexpected outcomes.

Recommendations

Short term, implement `symbol`, `name`, and `decimals` on `Curve` contracts.

Long term, ensure that contracts conform to all required and recommended industry standards.

10. Use of undefined behavior in equality check

Severity: High
Type: Undefined Behavior
Target: `contracts/Curve.sol`

Difficulty: High
Finding ID: TOB-DFX-010

Description

`CurveMath.calculateTrade` is used to compute the output amount for a trade. It computes the `S_{i,j}` function from the whitepaper using an iterative approach. The loop contains the following `if` statement:

```
            if (
                (outputAmt_ = _omega < _psi
                    ? -(_inputAmt + _omega - _psi)
                    : -(_inputAmt + _lambda.us_mul(_omega - _psi))) /
                    1e13 ==
                outputAmt_ / 1e13
            ) {
```

Figure 10.1: The output amount computation includes an equality check that may result in undefined behavior. ([CurveMath.sol#L105-L111])

On the left-hand side of the equality check, there is an assignment of the variable `outputAmt_`. The right-hand side uses the same variable. The Solidity 0.7.3. documentation states the following:

```
The evaluation order of expressions is not specified (more formally, the order in which the
children of one node in the expression tree are evaluated is not specified, but they are of
course evaluated before the node itself). It is only guaranteed that statements are executed
in order and short-circuiting for boolean expressions is done.
```

Figure 10.2: Solidity Documentation: "Order of Evaluation of Expressions"

It follows that this check constitutes an instance of undefined behavior. As such, the behavior of this code is not specified and could change in a future release of Solidity.

Exploit Scenario

A future Solidity release uses different semantics for this case. As a result, the `if` statement is executed differently, leading to unexpected outcomes.

Recommendations

Short term, rewrite the `if` statement such that it does not use and assign the same variable in an equality check.

Long term, ensure that the codebase does not contain undefined Solidity or EVM behavior.

11. Insufficient use of SafeMath

Severity: Undetermined
Type: Data Validation
Target: `contracts/CurveMath.sol`

Difficulty: High
Finding ID: TOB-DFX-011

Description

`CurveMath.calculateTrade` is used to compute the output amount for a trade. However, although `SafeMath` is used throughout the codebase to prevent underflows/overflows, it is not used in this calculation.

```
            if (
                (outputAmt_ = _omega < _psi
                    ? -(_inputAmt + _omega - _psi)
                    : -(_inputAmt + _lambda.us_mul(_omega - _psi))) /
                    1e13 ==
                outputAmt_ / 1e13
            ) {
```

Figure 11.1: The calculation of the difference between `_omega` and `_psi` does not account for an underflow. (CurveMath.sol#L105-L111)

Although we could not prove that the lack of `SafeMath` would cause an arithmetic issue in practice, all such calculations would benefit from the use of `SafeMath`.

Recommendations

Short term, change all instances of `us_mul` and `us_div` to `ABDKMath64x64.mul` and `.div`, respectively, which also operate on two fixed-point numbers but have overflow/underflow protections.

Long term, review all critical arithmetic to ensure that it accounts for underflows, overflows, and the loss of precision. Consider using `SafeMath` and the safe functions of `ABDKMath64x64` where possible to prevent underflows and overflows.

12. Assimilators' balance functions return raw values

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DFX-012
Target: `contracts/assimilators/(Cadc|Xsgd|Eurs)ToUsdAssimilator.sol`

Description

The system converts raw values to numeraire values for its internal arithmetic. However, in one instance it uses raw values alongside numeraire values.

```
    function getOriginSwapData(
        Storage.Curve storage curve,
        uint256 _inputIx,
        uint256 _outputIx,
        address _assim,
        uint256 _amt
    )
        private
        returns (
            int128 amt_,
            int128 oGLiq_,
            int128 nGLiq_,
            int128[] memory,
            int128[] memory
        )
    {
        uint256 _length = curve.assets.length;

        int128[] memory oBals_ = new int128[](_length);
        int128[] memory nBals_ = new int128[](_length);
        Storage.Assimilator[] memory _reserves = curve.assets;

        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.intakeRawAndGetBalance(_assim, _amt);

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

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

Figure 12.1: `getOriginSwapData` handles numeraire values. ([`Swaps.sol#L175-L209`]())

`originSwap` calls `getOriginSwapData` to retrieve an array of numeraire balances. The first `if` case is handled correctly, as `viewNumeraireBalance` correctly returns a numeraire value.

However, the second case assigns `_nBals[i]` to the result of `intakeRawAndGetBalance`. Here is an example of that function in the `CadcToUsdAssimilator`:

```
    function intakeRawAndGetBalance(uint256 _amount) external override returns (int128
amount_, int128 balance_) {
        bool _transferSuccess = cadc.transferFrom(msg.sender, address(this), _amount);

        require(_transferSuccess, "Curve/CADC-transfer-from-failed");

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

        uint256 _rate = getRate();

        balance_ = _balance.divu(1e18);

        amount_ = ((_amount * _rate) / 1e8).divu(1e18);
    }
```

Figure 12.2: The error-prone function within the `CadcToUsdAssimilator` ([`contracts/assimilators/CadcToUsdAssimilator#L44-L56`](contracts/assimilators/CadcToUsdAssimilator#L44-L56))

It is clear that the balance returned is a *raw* value, rather than a numeraire value.

The same bug is also present in the XSGD and EURS assimilators. It is absent from the USDC assimilator, since the raw value of USDC is also the numeraire value. (See TOB-DFX-013.)

Note that the original Shell Protocol does not contain this bug. That protocol computes the raw value of the *numeraire* token, which necessarily corresponds to the numeraire value.

```
    function intakeRawAndGetBalance (uint256 _amount) public returns (int128 amount_, int128
balance_) {

        bool _transferSuccess = cdai.transferFrom(msg.sender, address(this), _amount);

        require(_transferSuccess, "Shell/cDAI-transfer-from-failed");

        uint _redeemSuccess = cdai.redeem(_amount);

        require(_redeemSuccess == 0, "Shell/cDAI-redeem-failed");

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

```

```
        uint256 _rate = cdai.exchangeRateStored();

        balance_ = _balance.divu(1e18);

        amount_ = ( ( _amount * _rate ) / 1e18 ).divu(1e18);

    }
```

Figure 12.2: Shell Protocol contains the above assimilator code.
(`MainnetCDaiToDaiAssimilator#L35-L53`)

Our early test results show that for a deployed system containing USDC and CADC worth $1,000 each (with USDC valued at $1 and the CADC price oracle reporting a value of $0.10), an `originSwap` with CADC as the origin will always revert. If the `originAmount > 5e16`, it will revert in `CurveMath.calculateTrade` (i.e., the convergence algorithm fails). If `originAmount <= 5e16`, it will revert in `CurveMath.enforceHalts` (having reached the upper threshold).

It is clear that even for different values, when the swap does not revert, interchanging raw and numeraire values will produce unwanted results.

The same issue is present in the following functions corresponding to `targetSwap`, `viewOriginSwap`, and `viewTargetSwap`:

- `targetSwap: outputRawAndGetBalance`
- `viewOriginSwap: viewNumeraireAmountAndBalance`
- `viewTargetSwap: viewNumeraireAmountAndBalance`

These four affected functions constitute the only times these three functions are misused; therefore, correcting them will not cause any further issues.

Exploit Scenario
Alice uses a `Curve` to swap tokens. Since the calculation uses the raw value instead of the numeraire value, she receives more tokens than she should. As a result, Bob, a liquidity provider, loses funds.

Recommendations
Short term, change the semantics of the three functions listed above in the CADC, XSGD, and EURS assimilators to return the numeraire balance.

Long term, use unit tests and fuzzing to ensure that all calculations return the expected values. Additionally, ensure that changes to the Shell Protocol do not introduce bugs such as this one.

13. System always assumes USDC is equivalent to USD

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-DFX-013
Target: `contracts/assimilators/UsdcToUsdAssimilator.sol`

Description

Throughout the system, assimilators are used to facilitate the processing of various stablecoins. However, the `UsdcToUsdAssimilator`'s implementation of the `getRate` method does not use the USDC-USD oracle provided by Chainlink; instead, it assumes 1 USDC is always worth 1 USD.

```
// Multiple by the rate for usd its always 1
// solhint-disable-next-line
function getRate() public view override returns (uint256) {
    return uint256(1e8); // Oracle 8 decimals
}
```

Figure 13.1: The `UsdcToUsdAssimilator`'s `getRate` method uses a fixed rate.
([`UsdcToUsdAssimilator.sol#L32-L36`](UsdcToUsdAssimilator.sol#L32-L36))

A deviation in the exchange rate of 1 USDC = 1 USD could result in exchange errors.

Exploit Scenario

Alice is a user of DFX Finance contracts. The system assumes that the value of USDC is pegged to USD until it is discovered that each USDC is not backed by a US dollar. This causes the USDC-USD exchange rate to drop. Unfortunately, because the USDC-USD exchange rate is derived from a hard-coded integer literal, the system will not account for this change. As a result, Alice may unexpectedly lose funds when performing swaps.

Recommendations

Short term, replace the hard-coded integer literal in the `UsdcToUsdAssimilator`'s `getRate` method with a call to the relevant Chainlink oracle, as is done in other assimilator contracts.

Long term, ensure that the system is robust against a decrease in the price of any stablecoin.

14. Division operations do not consider rounding direction

Severity: Undetermined
Difficulty: High
Type: Data Validation
Finding ID: TOB-DFX-014
Target: `contracts/assimilators/*`

Description
Division operations throughout the codebase appear to be consistently rounded down, rather than up. Although no proof of concept has been created to verify this issue, conceptually, a system should round up when requesting funds from a user and round down when disbursing them. That would ensure that the pool remained appropriately funded and would prevent withdrawals from failing.

If the arithmetic is rounded down in a calculation of the amount a user owes to the system, the amount of the user's payment may be lower than expected. That could result in accounting issues during proportional deposits. Rounding up would ensure that the pool remained appropriately funded and prevent issues such as reverts during the last user's attempt to execute a withdrawal.

Recommendations
Short term, consider rounding up when determining the amount of tokens required in a deposit.

Long term, document and evaluate the effects of rounding errors in deposit and withdrawal operations, and account for scaling.

15. Curve parameters are set incorrectly

Severity: Low
Type: Data Validation
Target: `contracts/Orchestrator.sol`

Difficulty: Medium
Finding ID: TOB-DFX-015

Description
When initializing a `Curve`, the `includeAsset` method adds a new weight value to `curve.weights`. The provided numerator must be less than the denominator (`1e18`). However, the smallest denomination of weight possible (`1/1e18`) is added to the supplied weight. Intuitively, it would follow that the sum of all weights should represent 100% (i.e., the sum of all numerators should be no more than the denominator, `1e18`). This is not the case, though, as initializing with `0` and `(1e18 - 1/1e18)` would yield `1/1e18` and `1e18`, the sum of which is larger than the denominator.

For example, consider the following scenario: the system creates a test pool with two assets (a mock token and a mock USDC coin with a weight of 50% each) and writes a test that mints and approves one thousand dollars' worth of each asset for the `Curve` to use. In that case, a deposit of two thousand dollars would result in a revert. This is because the weight on each asset is slightly higher than usual and requires a deposit of additional tokens, which represent a small margin of error.

Additionally, the `Curve` construction increments many supplied integer parameters by one (such as within the `setParams` function) before storing them. Defining the parameters differently would generally increase readability while preventing confusion about the practical underlying values to a reader. In the least, the intent of these code paths is not immediately clear.

Exploit Scenario
Alice deposits funds into DFX Finance contracts using the `deposit` function. Because of the above error in arithmetic, the number of tokens transferred from Alice to the `Curve` contract is higher than the number she planned to deposit.

Recommendations
Short term, refactor the weight-related code to ensure that the correct amount of assets is transferred when `deposit` is called.

16. `setFrozen` can be front-run to deny deposits/swaps

Severity: Low
Difficulty: Medium
Type: Timing
Finding ID: TOB-DFX-016
Target: `contracts/Orchestrator.sol`

Description
Currently, a `Curve` contract owner can use the `setFrozen` function to set the contract into a state that will block swaps and deposits. A contract owner could leverage this process to front-run transactions and freeze contracts before certain deposits or swaps are made; the contract owner could then unfreeze them at a later time.

Exploit Scenario
Alice, a DFX Finance smart contract user, attempts to execute a swap. Around the same time, the contract owner front-runs a transaction, which freezes the contracts and causes the swap to fail. The contract owner subsequently sends another transaction to unfreeze the contracts. In this way, the contract owner can deliberately impede Alice's deposits and swaps.

Recommendations
Short term, consider rewriting `setFrozen` such that any contract freeze will not last long enough for a malicious user to easily execute an attack. Alternatively, depending on the intended use of this function, consider implementing permanent freezes.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Non-Security-Related Findings

The following findings do not have immediate or obvious security implications.

- **Remove the unused variable `iAsmltr`.** In the `Assimilators` library, there is a constant variable, `iAsmltr`, used throughout the library code to obtain the selector of a function from the IAssimilator interface. This is unnecessary, as Solidity supports the direct use of the interface to obtain selectors. For example, `abi.encodeWithSelector(iAsmltr.intakeRaw.selector, _amt)` can be replaced by `abi.encodeWithSelector(IAssimilator.intakeRaw.selector, _amt)`.

- **Consider refactoring the code such that equivalent integer literals are seldom repeated.** Contracts including the assimilators reuse large numbers such as `1e18` and `1e6` in rate calculations. These integer literals are repeated throughout the contracts many times. Consider refactoring them into constants to increase the code's readability, facilitate maintenance of the code, and prevent developer errors.

- **`ProportionalLiquidity`'s `burn` and `mint` functions should use the `account` parameter instead of `msg.sender` in the emission of a `Transfer` event.** These functions mint/burn tokens for the provided `account` address. However, they emit an event using `msg.sender` instead of `account`. Because the `account` argument is always set to `msg.sender` by the caller, this does not have a significant impact on the system; however, it should be fixed to enhance the code's correctness.

- **`CurveMath` and `ProportionalLiquidity` both define a constant `ONE_WEI` that is unnecessary and can be removed.** This may have previously served a purpose. However, if it is no longer necessary, it should be removed to improve the code's readability.

- **`Curve.originSwap` has an incorrect natural specification (natspec).** The first line of the natspec for `Curve.originSwap` states the following: "`/// @notice swap a dynamic origin amount for a fixed target amount.`" In actuality, `originSwap` swaps a fixed origin amount for a dynamic target amount. We recommend that the line be changed to reflect that. A correct natspec increases the readability of code and helps developers and users interact with it.

- **`CurveStorage.Curve.oracles` is never used**. `CurveStorage` defines a struct `Curve` with a member element, `mapping(address => IOracle) oracles`, which is neither set nor read in the codebase. We recommend removing this field to enhance the readability of the code.

C. Fix Log

After Trail of Bits completed the initial assessment, DFX Finance addressed certain issues identified in the report. The audit team then verified each of the fixes to ensure that it was applied appropriately. The results of these checks are provided below.

ID	Title	Severity	Status
01	Assimilators use a deprecated Chainlink API	High	Fixed
02	`_min*` and `_max*` have unorthodox semantics	Undetermined	Fixed
03	`CurveFactory.newCurve` returns existing curves without provided arguments	Informational	Fixed
04	Missing zero-address checks in `Curve.transferOwnership` and `Router.constructor`	Low	Fixed
05	Missing contract existence checks prior to `delegatecall`	Undetermined	Fixed
06	Unorthodox `Curve` constructions may cause undefined behavior	Low	Partially fixed
07	`CurveFactory.newCurve` fails to initialize `_derivativeAssimilators`	Low	Fixed
08	`safeApprove` does not check return values for `approve` call	Low	Fixed
09	ERC20 token `Curve` does not implement `symbol`, `name`, or `decimals` method	Informational	Fixed
10	Use of undefined behavior in equality check	Informational	Fixed
11	Insufficient use of `SafeMath`	Informational	Fixed
12	Assimilators' balance functions return raw values	Informational	Fixed
13	System always assumes USDC is equivalent to USD	Informational	Fixed
14	Division operations do not consider rounding direction	Informational	Fixed
15	Curve parameters are set incorrectly	Informational	Not fixed

16	`setFrozen` can be front-run to deny deposits/swaps	Informational	Not fixed

For additional information on each fix, please refer to the detailed fix log on the next page.

Detailed Fix Log

Finding 1: Assimilators use a deprecated Chainlink API
Fixed. The deprecated Chainlink API calls were replaced with the superseding API calls.

Finding 2: `_min*` and `_max*` have unorthodox semantics
Fixed. These variables are now used in the expected way—that is, as inclusive minimums (>=, <=) rather than exclusive minimums (>, <).

Finding 3: `CurveFactory.newCurve` returns existing curves without provided arguments
Fixed. This function now returns only new curves or reverts if one already exists. Another function, `getCurve`, was added to obtain the address of any existing curve. However, it will return a zero address when no curve with the provided parameters exists. This behavior should be documented for external users.

Finding 4: Missing zero-address checks in `Curve.transferOwnership` and `Router.constructor`
Fixed. The missing zero-address checks have been added to the code.

Finding 5: Missing contract existence checks prior to `delegatecall`
Fixed. The contract code now conducts contract existence checks with `require` statements prior to performing a delegate call.

Finding 6: Unorthodox `Curve` constructions may cause undefined behavior
Partially fixed. Although checks were added to make sure that each array would be divisible by the expected value (and therefore able to enter its respective initialization loop), other checks were not added. For example, there are no checks verifying whether the amount of `_assets` is more than five times the amount of `_assetWeights`, in which case some elements of `_assets` may be silently ignored.

Finding 7: `CurveFactory.newCurve` fails to initialize `_derivativeAssimilators`
Fixed. The `_derivativeAssimilators` code was removed entirely, as it was deemed unnecessary. Elements of this array will already be included in `_assets`.

Finding 8: `safeApprove` does not check return values for `approve` call
Fixed. The affected contract now uses OpenZeppelin's `safeApprove` function, which appropriately checks any values returned by an `approve` call.

Finding 9: ERC20 token `Curve` does not implement `symbol`, `name`, or `decimals` method
Fixed. The abovementioned fields have been added to the `Curve` contract.

Finding 10: Use of undefined behavior in equality check
Fixed. The assignment and comparison of `outputAmt_` were split into separate statements.

Finding 11: Insufficient use of `SafeMath`
Fixed. The affected contracts now leverage safe functions provided by `ABDKMath64x64`.

Finding 12: Assimilators' balance functions return raw values
Fixed. The affected contracts now use numeraire values instead of raw values.

Finding 13: System always assumes USDC is equivalent to USD
Fixed. This assimilator now uses the relevant USDC-USD Chainlink oracle contract.

Finding 14: Division operations do not consider rounding direction
Fixed. Deposit calculations now include `ONE_WEI`, which should provide a mechanism for rounding up. Note that this fix was reviewed manually, and our fix review did not include in-depth testing of the new arithmetic.

Finding 15: Curve parameters are set incorrectly
Not fixed. The DFX Finance team indicated that precision loss is minimal. For example, the team stated that converting a `uint256` with a value of `300000000000000000` to a floating-point representation (`int128`) and back would yield only a loss of `1`. As such, adding `1/1e18` to the weight will compensate for the precision loss.

Finding 16: `setFrozen` can be front-run to deny deposits/swaps
Not fixed. The DFX Finance team has indicated that it intends to leave this mechanism "as is," as only the DFX Finance team could access this functionality.