Report: flaws in the Curve pools contracts logic

experience, Killari

February 19, 2021

1 Overview

In this document, we describe some flaws that affect all of the Curve pools contracts and might be exploitable by a profit seeking attacker under some conditions, or lead to user losses. Firstly, we note that the accepted proof of path independence for the Curve invariant provided by Angeris and Chitra is incorrect, meaning that a user cannot expect to get the exact same amount back from a back and forth trade even in the case of zero trade fees. Secondly, due to the way the Stableswap invariant is calculated, it is possible to change the composition of the pools to precise values found to break the computation. An attacker may make use of this flaw to effectively steal from the pool through repeated actions in a single transaction. Attached to this report, we provide a collection of scripts used to simulate the Curve contracts during our research, as well as an implementation of two exploits on a forked Ethereum mainnet chain: the loss of funds by a user doing repeated trades, and an attack yielding a small profit by adding and removing liquidity with precise values. While the latter attack profits are outweighted by the gas fees in the current state of the Ethereum network, they could be profitable were Curve to be implemented with the same logic on a cheap layer two network like the planned zkSync implementation, or when the eth1 state is migrated to eth2.

2 Theory

2.1 Stableswap basics

For context, we recall the theory of the Stableswap invariant and the way it is implemented in the Curve contracts. Curve is a what has been referred to as a "constant function market maker", or CFMM for short. As described in the original Stableswap paper [?], Given a pool $\vec{x}$ composed of n tokens with respective balances x_i, a user requesting to swap some amount Δx_i of tokens i for tokens j should be given an amount Δx_j such that the following equality remains true with the updated balances:

$$An^n \sum x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i} \tag{1}$$

Where A is a fixed parameter called the "amplification factor". As we can see, it is implied that this equality was true to begin with. In particular, this means that the equation was set with an appropriate value of D. This is because as implied in the Stableswap paper, a D that breaks this equality would not necessarily correspond to a spot price of 1 when balances are close to each other. To do this, given an initial portfolio of tokens $\vec{x}$, we solve the equation $f(D) = 0$ where:

$$f(D) = An^n \sum x_i + D - ADn^n - \frac{D^{n+1}}{n^n \prod x_i} \tag{2}$$

This is simply Equation 1 written differently. With this equation solved, we get the invariant for the current pools. As discussed above, finding the amount to give out given an amount in is also calculated by solving the invariant equation for the unknown quantity to give out.

2.2 Path dependence

In their 2020 paper, Angeris and Chitra [**?**] formalize CFMMs like Uniswap's constant product by introducing a *trade function* φ. Under their formalism, if $\vec{x}$ is the vector representing the reserves, $\vec{\Delta}$ a vector representing the amounts in and $\vec{\Lambda}$ a vector representing the amounts out, a *trade* is a set $\{\vec{\Delta}, \vec{\Lambda}\}$ such that:

$$\varphi(\vec{x}, \vec{\Delta}, \vec{\Lambda}) = \varphi(R, 0, 0) \tag{3}$$

They then proceed to define the Curve trading function as:

$$\varphi(\vec{x}, \vec{\Delta}, \vec{\Lambda}) = \alpha \mathbf{1}^T (\vec{x} + \gamma\vec{\Delta} - \vec{\Lambda}) - \beta \prod_{i=1}^{n} (x_i + \gamma\Delta_i - \Lambda_i)^{-1} \tag{4}$$

And they later state that this function verifies the sufficient condition that the set of possible trades can be written as:

$$T(R) = \{(\Delta, \Lambda) \mid \psi(R + \Delta - \Lambda) \geq \psi(R)\} \tag{5}$$

The issue in this reasoning lies in the definition of the Curve trading function. More specifically, the Stableswap invariant does not fall under this formalism, because there are no fixed α and β such that one could always write a unique trading function as in Equation 3. Indeed, as discussed in Section 2.1, for each specific amount of tokens in the pool, a different D is calculated to make sure Equation 1 holds. This means that we cannot write the condition in Equation 5 above in that context, as a change of balance changes the value of D, and thus the function φ.

In the attached code, we prove this path dependence by showcasing a specific sequence of trades that leads to the user losing 5% of the original amount they traded, far more than what they would expect to lose from the fees. We were not able to find a sequence of trades that would let the user effectively steal from the pool, but because of the lack of path independence property, this cannot be excluded unless we prove that any sequence of trades never leads to a higher amount out than the initial amount in.

Nevertheless, this can be considered a flaw in the fundamental design of Curve, as this is not the expected behavior.

2.3 Invalid D computations

The Curve contract attempts to find an appropriate value of D for a given set of current balances by solving the $f(D) = 0$ equation iteratively using the Newton method. Starting with an initial guess D_0, the next value of D is computed using the following sequence:

$$D_{n+1} = D_n - \frac{f(D_n)}{f'(D_n)} \tag{6}$$

This is done until $\|D_{n+1} - D_n\| < \varepsilon$ with some user defined ε precision. It can be shown that the method either converges to a root of the function, that is, a value of D verifying our equation, or does not converge. The number of steps it takes to converge (if it does) is variable. Below is an example implementation of that algorithm in the Curve contracts from `StableSwapUSDT.vy` :

```
        def get_D(xp: uint256[N_COINS]) -> uint256:
            S: uint256 = 0
            for _x in xp:
                S += _x
            if S == 0:
                return 0

            Dprev: uint256 = 0
            D: uint256 = S
            Ann: uint256 = self.A * N_COINS
            for _i in range(255):
                D_P: uint256 = D
                for _x in xp:
                    D_P = D_P * D / (_x * N_COINS + 1)  # +1 is to prevent /0
                Dprev = D
                D = (Ann * S + D_P * N_COINS) * D / ((Ann - 1) * D + (N_COINS + 1) *
    D_P)
                # Equality with the precision of 1
                if D > Dprev:
                    if D - Dprev <= 1:
```

```
                break
        else:
            if Dprev - D <= 1:
                break
    return D

```

As we can see, the algorithm uses 255 iterations, which normally should be enough to converge to a root of the function. But what if we converge to something other than a root of the function before the end of the loop? In $\mathbb{R}$, as stated before, that is not possible. But we see that this algorithm uses 256 bits unsigned integers and integers division. If we encounter any value D_i such that $f(D_i) < f'(D_i)$, the integers division will give 0 for $f(D_n)/f'(D_n)$, and we will have $D_{i+1} = D_i$, thus meeting the condition to terminate the loop before we arrive at an appropriate value.

This opens the door for possible attacks that would consist in finding functions f (i.e. pool balances) that cause this early termination, making it so that `get_D` returns a value of D such that $f(D) \neq 0$. Though we didn not find any attack using this flaw specifically, it would be useful to check that the D obtained is correct to prevent any creative exploit of this flaw.

3 Set up of possible attacks or bugs

3.1 Loss of traders funds via back and forth trades

Due to the path dependency explained above, it becomes possible in principle for a particular sequence of trade to net a profit, or a loss higher than the trading fees for a trader. In particular for large trades that change the pool composition such that the new D after the trade significantly differs from the original D (for some meaning of significant not fully defined in our work), the invariant would not give the same amount out as in.

We demonstrated this bug in a local fork of the Ethereum mainnet, at block 11853997 in the USDT Curve pool. At that block the pool was composed of 1757662794027293 CDAI, 3506962588474923 CUSC and 324570592963 USDT. By swapping 2776297808957086 CUSDC to CDAI, and then swapping the exact amount of CDAI given out back to CUSD, we end up with 2624090423136564 CUSDC, or a loss of 5.48%.

The code to replicate this can be found in our project in file `exploit.sol`.

This is one particular proof of concept example, we have not explored all possibilities related to the path dependency in the case of trades. We don't know whether it would be possible for an attacker to construct a particular path of trades netting a profit, but we believe the demonstrated flaw already constitutes a deviation from the expected behavior of Curve.

A profit might come from combining invalid D computations and a back and forth trade, for example if the attacker trades an amount crafted specifically to trigger an invalid computation and then trades back the same amount that they got out from the first trade. Alternatively, one could find a small amount of liquidity to add in order to trigger an invalid D calculation, and then perform a back and forth trade. One would have to mine such sequences in a simulator. We briefly explored the parameter space with a Python simulator copying the behavior of the Curve contracts on our local machines, but there is a lot left to explore. Our simulator to mind invalid values of D can be found in the script `simulator.py`.

3.2 Small profits from adding and withdrawing liquidity in sequence

...description of the attack...

While the profits we found do not offset the transaction fees in the current state of the Ethereum network, this could be considered a critical flaw in the logic of the Curve contracts which could be exploited if the gas fees were to be much lower, for example by migration to a layer two solution or after the migration of the eth1 state to eth2.