

# Report: possible vulnerabilities in the Curve pools logic

experience, Killari

February 22, 2021

## 1 Overview

In this document, we describe some vulnerabilities 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 [1] 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. Though we have not yet found a profitable attack yet, an attacker may be able to make use of this vulnerability to effectively steal from the pool.

## 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 what has been referred to as a "constant function market maker", or CFMM for short. As described in the original Stableswap paper [2], Given a pool  $\vec{x}$  composed of  $n$  tokens with respective balances  $x_i$ , a user requesting to swap some amount  $\Delta x_i$  of tokens  $i$  for tokens  $j$  should be given an amount  $\Delta x_j$  such that the following equality remains true with the updated balances:

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

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. In order to achieve this, given an initial portfolio of tokens  $\vec{x}$ , we solve the equation  $f(D) = 0$  where:

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

where  $f$  is obtained from Equation 1. With this equation solved, we get the invariant for the current pool composition. 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*  $\varphi$ . 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:

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

They then proceed to define the Curve trading function as:

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

They later state that this function verifies the sufficient condition for path independence that the set of possible trades can be written as:

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

where  $\psi$  is a single non decreasing quasiconcave function. 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  $\alpha$  and  $\beta$  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  $\varphi$ .

We tested this path dependence by executing a specific sequence of trades on a local Ethereum mainnet fork 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 vulnerability 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:

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

This is done until  $\|D_{n+1} - D_n\| < \varepsilon$  with some user defined  $\varepsilon$  precision. It can be shown that in  $\mathbb{R}$  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 :

```

1  def get_D(xp: uint256[N_COINS]) -> uint256:
2      S: uint256 = 0
3      for _x in xp:
4          S += _x
5      if S == 0:
6          return 0
7
8      Dprev: uint256 = 0
9      D: uint256 = S
10     Ann: uint256 = self.A * N_COINS
11     for _i in range(255):
12         D_P: uint256 = D
13         for _x in xp:
14             D_P = D_P * D / (_x * N_COINS + 1) # +1 is to prevent /0
15         Dprev = D
16         D = (Ann * S + D_P * N_COINS) * D / ((Ann - 1) * D + (N_COINS + 1) * D_P)
17         # Equality with the precision of 1
18         if D > Dprev:
19             if D - Dprev <= 1:
20                 break
21         else:
22             if Dprev - D <= 1:
23                 break
24     return D
25

```

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 did not find any attack using this vulnerability specifically, it would be useful to check that the  $D$  obtained is correct to prevent any creative exploit that we might not have thought of.

### 3 Setup of possible attacks or bugs

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 to be defined through further exploration), the invariant would not give the same amount out as in.

We demonstrated this issue in a local fork of the Ethereum mainnet, at block 11853997 in the USDT Curve pool. At that block the pool was composed of  $1757662794027293/10^8$  CDAI,  $3506962588474923/10^8$  CUSC and  $324570592963/10^8$  USDT. By swapping  $2776297808957086/10^8$  CUSDC to CDAI, and then swapping the exact amount of CDAI given out back to CUSD, we end up with  $2624090423136564/10^8$  CUSDC, or a loss of 5.48%.

This is a specific 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 vulnerability 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  $D$  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. The attacker 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.

## 4 Mitigation strategies

### 4.1 Path dependency

If our reasoning regarding path dependency is correct, then it is a fundamental limitation of the Stableswap implementation. However, since we couldn't find any profitable sequence of trades in our initial search, one could suspect that this can only lead to losses, which would preclude this vulnerability from ever becoming an attack vector. In order to be confident that it is the case, one would need to formally analyze the properties of the Stableswap system and prove that the path dependency can only lead to losses. As long as this is not proved, the door is opened to potential profit motivated exploitation of this aspect of the automated market maker.

### 4.2 Invalid $D$ s

To prevent any potential exploit of the invalid  $D$  calculations, an easy fix would be to change the contracts to include a check that *after* a trade or the addition / removal of liquidity, the  $D$  obtained indeed verifies Equation 1, and refuse the action if it doesn't.

Another potential mitigation strategy related to  $D$  would be to check that it moves in the "expected" direction after each trade. This is already partly done when adding liquidity (the fee calculation is missing an assert), but not for trading or when an user is withdrawing liquidity. One would have to

explore what the expected direction is for different types of trades analytically. This might prevent some exploits that we have not found in our initial research.

## References

- [1] Angeris and Chitra. Improved price oracles: Constant function market makers. 2020.
- [2] Michael Egorov. Stableswap - efficient mechanism for stablecoin liquidity. 2019.