Potential critical vulnerability in Curve based contracts

experience, Killari

February 11, 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. This results in a deviation of the spot price from the expected price of ~ 1, *even in the case of balanced pools.* The attacker can then arbitrage against that virtual pool to turn a profit and effectively steal from it 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 wouldn 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 $\mathbb{R}$ 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.

Our attack 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 $f(D) \neq 0$, *and* such that the spot price at these balances deviates significantly enough from 1. We can thus do risk-free "virtual arbitrage" between that new pool and the original pool in a single transaction, which could gradually empty the pool if repeated enough times.

3 Step-by-step description of the attack