Summary

We recently executed AIP-267: Treasury Management - Acquire B-80BAL-20WETH which converted Aave's aBAL, aEthBal, and BAL holdings, along with wETH, to B-80BAL-20WETH.

In order to accomplish this, we built our proposal payload with CoW Swap to protect the DAO from MEV exploits, bad slippage, and reentrancy risk.

Another method of acquiring 80BAL-20WETH would have been to deposit into the Balancer pool directly, though as we show below, this option would have incurred more technical complexity and would have resulted in the DAO receiving less 80BAL-20WETH as a result of slippage and MEV.

As a result, the swap resulted in Aave receiving an additional 1,918.72 (USD: $24,732.30) units of B-80BAL-20WETH

compared to a direct deposit.

Analysis

On July 10, 2023, at 1:21:59 PM UTC on block 17663526

we executed the following proposal.

Transaction: Ethereum Transaction Hash (Txhash) Details | Etherscan

The proposal swapped BAL and wETH tokens for B-80BAL-20WETH using COW Swap instead of doing a direct deposit into Balancer.

Here is a rundown of what the DAO received via swap and what could have received doing a direct deposit.

Leg 1 of Swap: BAL → B-80BAL-20WETHEthereum Transaction Hash (Txhash) Details | Etherscan

Block: 17663984

Quantity: 310,651.13 BAL

Received: 110,302.65 B-80BAL-20WETH

Price of BAL: $4.56

USD value of BAL sent: $1,417,122.28

Leg 2 of Swap: ETH → B-80BAL-20WETH:Ethereum Transaction Hash (Txhash) Details | Etherscan

Block: 17665329

Quantity: 326.88 Ether

Received: 46,867.21 B-80BAL-20WETH

Price of ETH: $1864.45

USD value of ETH sent: $609,451.416

Total USD sent: $2,026,573.69

Total BPT Received: 157,169.86 B-80BAL-20WETH

Average BPT price: $12.89

USD value of BPTs received: $2,025,919.53

Using Balancer's simulation tools (detailed below) we obtain at the Payload execution's block the following number of BPT expected to be received: 157,615.37

This is a negative difference "lost" by the DAO of 445.51 B-80BAL-20WETH (USD: -$11,176.72)

. However, this number does not take into account the slippage tolerance of 1.5% we allowed for the trade. If we had executed the two-sided deposit expecting 157,615.37

units out, the execution of the proposal would have most likely (very high likelihood) failed.

Accounting for 1.5% slippage, we would have expected to get out 155,251.14

units of the BPT. Had we allowed for 1.5% slippage, it would have certainly been taken advantage of by an MEV bot and thus, resulted in a positive difference for the swap vs. the direct deposit of 1918.72 (USD: $24,732.30)

units.

In order to receive the same amount of tokens via direct-deposit as the DAO received by utilizing COW Swap, the direct-deposit would have only been able to allow for 0.28% slippage

, a very tight range that could or could not have been executed.

Even though the swap allowed for 1.5% slippage, the actual slippage was lower at 1%, which resulted in the DAO receiving an extra 556 B-80BAL-20WETH (USD: $7,300)

units than it could have expected, had the maximum slippage been allowed. COW Swap's routing ensured the DAO got the best possible price.

Conclusion

Even though at first glance it seems that a direct-deposit would have yielded similar results to using COW Swap without having to code a custom solution, this does not take into account the fact that governance means execution is delayed and we're comparing two tools that work quite differently.

For the DAO, it is impossible to know in advance how many BPTs to expect 5 days from the proposal being submitted. Thus, configuring the minimum amount out allowed in the proposal would have been a guessing game that may or may not have resulted in a successful execution. In order to guarantee execution, the allowed slippage would have had to be way greater than the 1.5% specified. With how rampant MEV bots are running, it is almost a guarantee they would have been able to profit off the DAO while front-running this operation.

The delay in the second execution was not ideal and added uncertainty to the job. The oversight here was understanding that Chainlink updates oracle prices once every 24hours, or after a 2% move. While not what was expected, that was the slippage protection doing its job as prices had moved away too much from where the contract was expecting to trade at and meant it would just wait until the units expected coming back from Chainlink (accounting for slippage) and the units being offered by the trade were aligned. Once the market moved back closer to our price, the swap was executed.

With how volatile exchange rates can be, it would not have been possible to use a different tool while guaranteeing good execution. Leveraging COW Swap + Milkman, the DAO traded within 1.5% of the prices at time of execution, whatever that price would be.

This resulted in the DAO maximizing the BPTs it received while being protected against MEV and bad slippage.

Tests

//SPDX-License-Identifier: MIT

pragma solidity ^0.8.14;

import "forge-std/Test.sol";

import "@balancer-labs/v2-interfaces/contracts/standalone-utils/IBalancerQueries.sol"; import "@balancer-labs/v2-interfaces/contracts/vault/IVault.sol";

// forge test -vv --fork-url https://mainnet.infura.io/v3/{API_KEY} --fork-block-number 17665329 --match-contract TwoTokenJoinTest

contract TwoTokenJoinTest is Test { // state vars bytes32 public poolId = 0x5c6ee304399dbdb9c8ef030ab642b10820db8f56000200000000000000000014; address public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2; address public constant BAL =0xba100000625a3754423978a60c9317c58a424e3D; address public constant BPT = 0x5c6Ee304399DBdB9C8Ef030aB642B10820DB8F56;

uint256 wethIn = 326880000000000000000;
uint256 balIn = 310651125031164000000000;

// using 0 here is fine as there is no risk of being sandwhiched in a forked environment.
uint256 minBptOut = 0;

IBalancerQueries public queryHelper;
IVault public vault;

// Build the Join request
IAsset[] tokens = [IAsset(BAL), IAsset(WETH)];
uint256[] maxAmountsIn = [balIn, wethIn];

```
uint256 constant JOIN_KIND_EXACT = 1; // join with exact tokens
bytes joinUserData = abi.encode(JOIN_KIND_EXACT, maxAmountsIn, minBptOut);

IVault.JoinPoolRequest joinRequest = IVault.JoinPoolRequest({
    assets: tokens,
    maxAmountsIn: maxAmountsIn,
    userData: joinUserData,
    fromInternalBalance: false
});

// tests
function setUp() public {
    queryHelper = IBalancerQueries(0xE39B5e3B6D74016b2F6A9673D7d7493B6DF549d5);
    vault = IVault(0xBA12222222228d8Ba445958a75a0704d566BF2C8);

    deal(BAL, address(this), balIn);
    deal(WETH, address(this), wethIn);

    IERC20(BAL).approve(address(vault), type(uint256).max);
    IERC20(WETH).approve(address(vault), type(uint256).max);
}

function test_JoinPool() public {
    //part 1: https://etherscan.io/tx/0x5269878143138e6a12ed28c7df5f5ab89d3e629da0410d59b6a211141a36207c
    //block: 17663984
    //quantity: 310,651.125031164965964229 BAL
    // amnt: 310651125031164000000000

    //part 2: https://etherscan.io/tx/0x94f5e056800d2256b018e526fbf2861120c7c05fa491f56ae4f55601cf68805a
    //block: 17665329
    //326.88 Ether
    // amnt 326880000000000000000

    (uint256 bptOut,) = queryJoin();

    console.log("bptOut at block execution", bptOut);

    console.log("joining after block %d has been mined", block.number);
    console.log("WETH balance before join: %d", IERC20(WETH).balanceOf(address(this)));
    console.log("BAL balance before join: %d", IERC20(BAL).balanceOf(address(this)));
    console.log("BPT balance before join: %d", IERC20(BPT).balanceOf(address(this)));

    vault.joinPool(
        poolId,
        address(this),
        address(this),
        joinRequest
    );

    console.log("WETH balance after join: %d", IERC20(WETH).balanceOf(address(this)));
    console.log("BAL balance after join: %d", IERC20(BAL).balanceOf(address(this)));
    console.log("BPT balance after join: %d", IERC20(BPT).balanceOf(address(this)));
}

function queryJoin() public returns(uint256 bptOut, uint256[] memory amountsIn) {
    // this function should never be called onchain to calculate limits as it
    // can easily be frontrun.
    (bptOut, amountsIn) = queryHelper.queryJoin(
        poolId,
        address(this),
        address(this),
        joinRequest
    );
}

}
```

Execute test with the following command:

forge test -vv --fork-url mainnet --fork-block-number 17663977 --match-contract TwoTokenJoinTest