CoW BalancerV1-AMM

This document presents the finding of a smart contract audit conducted by Côme du Crest for Gnosis.

Scope

The scope includes all contracts of defi-wonderland/balancer-v1-amm as of commit 8a60ea2.

Context

The goal of BalancerV1-AMM is to deploy BalancerV1 pools from which CoW orders can programmatically trade. The deployed pool is a fully functional BalancerV1 pool, with added function `isValidSignature()` allowing `GPv2Settlement` to verify a signature on the pool to trade one token for another of the pool. The function `isValidSignature()` will verify that enough tokens are sent for the expected amount of tokens withdrawn.

Most contracts are forked and updated version of balancer/balancer-core updated to the most recent solidity version with some features adapted.

Status

The report has been sent to developer team

Issues

▼ [Med] Incorrect fee calculation for joining/exiting pools

Summary

When joining or exiting a pool, liquidity providers should theoretically provide/withdraw a ratio of the pool's token balances equivalent to the ratio of the shares they mint/burn. The protocol allows to join a pool using a single token. This is equivalent to providing this single token and swapping a certain amount of this token to the other tokens of the pool to provide them.

Since a virtual swap takes place in that case, a fee should be applied to that swap. The fee applied by the protocol when joining/exiting a pool is incorrect and does not match the fee that should be applied when swapping the necessary tokens to provide liquidity or exit with a single token.

Vulnerability Detail

Joining a pool using a single token `tokenA` is equivalent to providing that token for a ratio of the pool's balance of that token `tokenAIn/tokenA.balanceOf(pool)` and swapping the remaining `tokenA` to provide the other tokens. The same ratio of tokens should be provided for the other tokens of the pool, e.g. for a pool with two tokens: `tokenBIn/tokenB.balanceOf(pool) = tokenAIn/tokenA.balanceOf(pool)`.

That is, the amount of `tokenB` that should be obtained by swapping `tokenA` is proportional to the ratio of the balances of `tokenA` and `tokenB` in the pool: `tokenBIn = tokenAIn * tokenB.balanceOf(pool) / tokenA.balanceOf(pool)`.

However the applied fee in the code is `(1 - weightTokenA) * poolFee * tokenAIn` as if `(1 - weightTokenA)` proportion of tokens needed to be swapped. This fee does not take into account the ratio of balance in the pool and may vary greatly from the correct fees.

```
    function calcPoolOutGivenSingleIn(
        uint256 tokenBalanceIn,
        uint256 tokenWeightIn,
        uint256 poolSupply,
        uint256 totalWeight,
        uint256 tokenAmountIn,
        uint256 swapFee
```

```
    ) public pure returns (uint256 poolAmountOut) {
        // Charge the trading fee for the proportion of tokenAi
        ///  which is implicitly traded to the other pool tokens.
        // That proportion is (1- weightTokenIn)  // @audit incorret fees applied
        // tokenAiAfterFee = tAi * (1 - (1-weightTi) * poolFee);
        uint256 normalizedWeight = bdiv(tokenWeightIn, totalWeight);
        uint256 zaz = bmul(bsub(BONE, normalizedWeight), swapFee);
        uint256 tokenAmountInAfterFee = bmul(tokenAmountIn, bsub(BONE, zaz));

        uint256 newTokenBalanceIn = badd(tokenBalanceIn, tokenAmountInAfterFee);
        uint256 tokenInRatio = bdiv(newTokenBalanceIn, tokenBalanceIn);

        // uint newPoolSupply = (ratioTi ^ weightTi) * poolSupply;
        uint256 poolRatio = bpow(tokenInRatio, normalizedWeight);
        uint256 newPoolSupply = bmul(poolRatio, poolSupply);
        poolAmountOut = bsub(newPoolSupply, poolSupply);
        return poolAmountOut;
    }
```

Picture the case where a pool has an invariant `V = tokenA.balanceOf(pool) ^ 0.5 * tokenB.balanceOf(pool) ^ 0.5` and a ratio `tokenA.balanceOf(pool) / tokenB.balanceOf(pool) = 0.01`. Joining the pool providing `tokenA` should swap most of the tokens to `tokenB` as 100 times more `tokenB` should be provided, as such the fee should be applied on `100/101 * tokenAProvided`. Instead the fee will be applied to `(1 - 0.5) * tokenAProvided` grating a way lower fee for the liquidity provider.

Impact

For users wanting to swap tokens on the pool, they can select among two fees and chose the lowest one:

- Directly swap on the pool using `swapExactAmountIn()` or `swapExactAmountOut()`
- Swap tokens by entering the pool with one token and exiting with another which incurs a different fee proportional to the weights of the tokens

Users providing liquidity from the pool can also decide on using the lowest fee by either:

- Swapping tokens to proportional ratio of the pool's tokens and calling `joinPool()`
- Swapping tokens to the single token with the lowest weight and calling `joinswapExternAmountIn()` / `joinswapPoolAmountOut()`

The same goes for exiting the pool withdrawer can decide on either:

- Exiting the pool with `exitPool()` and swapping the exited tokens into the desired token
- Exiting the pool using a single exit token `exitswapExternAmountOut()` / `exitswapPoolAmountIn()`

The result is that users always have the choice between the correct fee and a different incorrect fee that can be much lower especially in unbalanced pools. The result is a loss of fee for the pool.

The four affected functions are `calcPoolOutGivenSingleIn()`, `calcSingleInGivenPoolOut()`, `calcSingleOutGivenPoolIn()`, `calcPoolInGivenSingleOut()`.

Code Snippets

https://github.com/defi-wonderland/balancer-v1-amm/blob/8a60ea23ad6d55096f9069d5d9c2d3434192765b/src/contracts/BMath.sol#L136-L300

Recommendation

BalancerV1's protocol team did not seem to care. So you may acknowledge the issue but it feels kinda odd.

The constraints that should be respected when joining a two-tokens pool using a single `tokenA` with amount `tokenAProvided` are:

- `tokenAProvided = tokenAIn + calcInGivenOut(tokenA.balanceOfPool, tokenWeightA, tokenB.balanceOf(pool), tokenWeightB, tokenBIn, swapFee)`
- `tokenBIn/tokenB.balanceOf(pool) = tokenAIn/tokenA.balanceOf(pool)`
- `mintedShares/totalShares = tokenBIn/tokenB.balanceOf(pool)`

The unknown variables to determine are `tokenAIn`, `tokenBIn`, and `mintedShares`. We have three equations with three unknown variables which can be solved. The case is more complex with pools with more tokens.

▼ [Info] BPool may not work for some tokens

Summary

The functions `_pullUnderlying()` and `_pushUnderlying()` used to pull and push tokens from/to users expect a boolean return value from the token. Some tokens fail to correctly implement the ERC20 standard and do not return any value on `transferFrom()` and `transfer()`. The pool will revert when attempting to interact with such tokens.

Vulnerability Detail

BPool expect a boolean return value on `transferFrom()` and `transfer()`:

```
function _pullUnderlying(
    address erc20,
    address from,
    uint256 amount
) internal virtual {
    bool xfer = IERC20(erc20).transferFrom(from, address(this), amount);
    if (!xfer) {
        revert BPool_ERC20TransferFailed();
    }
}

function _pushUnderlying(
    address erc20,
    address to,
    uint256 amount
) internal virtual {
    bool xfer = IERC20(erc20).transfer(to, amount);
    if (!xfer) {
        revert BPool_ERC20TransferFailed();
    }
}
```

Tokens that do not return a boolean will make the decoding of the return value to revert.

Impact

BPool will not work with tokens that incorrectly implement ERC20 and do not return a value on `transfer()` and `transferFrom()`.

Code Snippets

https://github.com/defi-wonderland/balancer-v1-amm/blob/8a60ea23ad6d55096f9069d5d9c2d3434192765b/src/contracts/BPool.sol#L622-L640

Recommendation

Acknowledge the issue. It will be immediately clear when such tokens are used and not cause a loss of funds.

▼ [Info] Immutable domain separator in BCoWPool

Summary

The domain separator used to compute the hash of the GPv2 order in `BCoWPool` is copied at deployment time from `GPv2Settlement`. If the chain ID changes, the domain separator of `GPv2Settlement` and `BCoWPool` will still match the old chain ID separator and allow for replayability of orders across forked chains.

Vulnerability Detail

`BCoWPool` uses an immutable domain separator to match the hash of the order decode from signature with the signed hash:

```
contract BCoWPool is IERC1271, IBCoWPool, BPool, BCoWConst {

    bytes32 public immutable SOLUTION_SETTLER_DOMAIN_SEPARATOR;

    constructor(address _cowSolutionSettler, bytes32 _appData) BPool() {
        ...
        SOLUTION_SETTLER_DOMAIN_SEPARATOR = ISettlement(_cowSolutionSettler)
            .domainSeparator();  // @audit hard-coded chain ID into domain separat
        ...
    }

    function isValidSignature(
        bytes32 _hash,
        bytes memory signature
    ) external view returns (bytes4) {
        GPv2Order.Data memory order = abi.decode(signature, (GPv2Order.Data));

        ...

        bytes32 orderHash = order.hash(SOLUTION_SETTLER_DOMAIN_SEPARATOR);
        if (orderHash != _hash) {
            revert OrderDoesNotMatchMessageHash();
        }

        ...
    }
```

`GPv2Settlment` uses a chain ID computed at deployment time:

```
    bytes32 public immutable domainSeparator;

    constructor() {
        // NOTE: Currently, the only way to get the chain ID in solidity is
        // using assembly.
        uint256 chainId;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            chainId := chainid()
```

```
        }

        domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPE_HASH,
                DOMAIN_NAME,
                DOMAIN_VERSION,
                chainId,
                address(this)
            )
        );
    }
```

Impact

If the chain forks into two different chains with different chain ID, the `GPv2Settlment` orders involving a `BCoWPool` swap will be replayable across the two chains.

Code Snippets

https://github.com/defi-wonderland/balancer-v1-amm/blob/8a60ea23ad6d55096f9069d5d9c2d3434192765b/src/contracts/BCoWPool.sol#L51

https://github.com/gnosis/gp-v2-contracts/blob/16e23ec37384d63dc51f9e1878a0084c5645f1ef/src/contracts/mixins/GPv2Signing.sol#L72

Recommendation

It is unlikely that a fork occurs and GPv2 did not provision for this case anyway. This can be acknowledged.

To fix it, compute the domain separator using `chainId()` when needed instead of storing at as an immutable. Otherwise, use https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/EIP712.sol