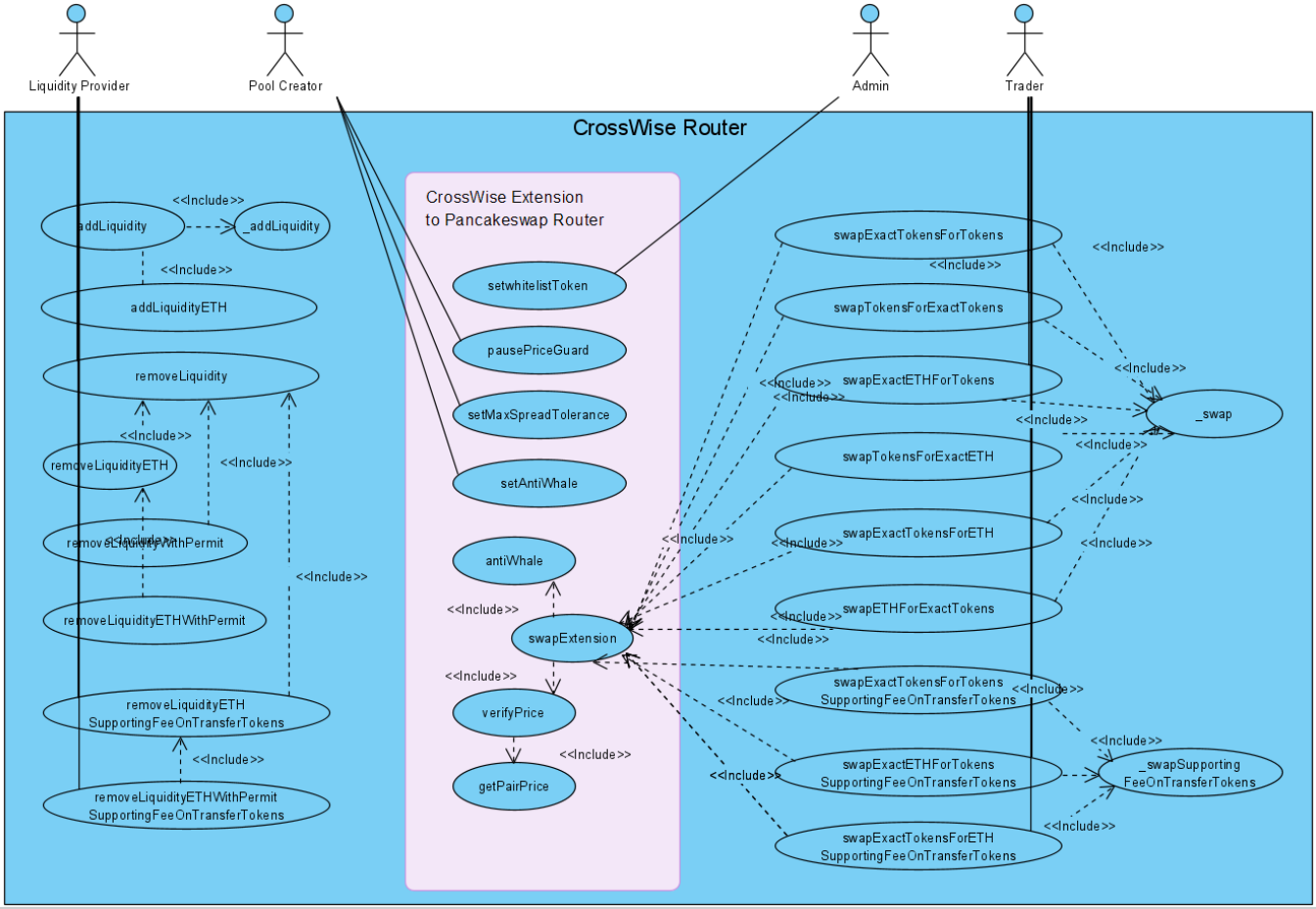
Source Core Review - CrossWise Router contract



Note: CrossWise router is essentially the Pancakeswap router contract extended with hooks for checking swap amount and price.
As the router contract extends the proven Pancakeswap router contracts, we focus on the extension.

setMaxSpreadTolerance(.) function:

The creator of a pair can change the price tolerance of the pair.

setAntiWhale(.) funciton:

The creator of a pair can change whether the anti-whale check will be active or not.

The antiWhale and verifyPrice checks:

All and only swap calls are subject to the antiWhale and veryfyPrice checks before every swap action.

admin and priceConsumer

The admin and priceConsumer is set when the contract is deployed.

- the admin has the privilege to setWhilelistToken(.)
- The priceConsumer contract is used to getLatestPrice(), maybe from the off-chain side.

0. General

0.1 The position of router, in general

- The adding/removing liquidity to/from liquidity pools and swapping one token for another, necessarily introduce the concepts of pair, pair factory, and router.
- The conceptual structure of pair, pair factory, and router was proven, and we don't need to create another structure.
- According to the pair-factory-router architecture:
    - Factory is responsible for creating, authentication of, and tracking inventory of token pairs, which is a liquidity pool.
    - Pair is responsible for:
        - managing token reserves via receiving and sending,
        - ensuring the consistency of liquidity constant K,
        - maintaining the price oracle,
    - Router is responsible for
        - Routing the requests of addLiquidity, swap, and their variations, to proper pairs
        - controlling token input amounts to pairs

0.2 The position that the CrossWise router takes

The CrossWise router extended the conventional router with the following functions:

- setWhitelistToken
- pausePriceGuard
- setMaxSpreadTolerance
- setAntiWhale

These functions belong to the category of input data to pairs, conforming to the architecture. CrossWise router keep the same position as Pancakeswap router takes. Architecture has by and large no problem.

1. Finance

1.1 Price oracle

- Logic: The code is supposed to call a price oracle for the price on pairs. The v1.1 version of CrossWise Dex ran without oracle price services.
- Comment: The pairs maintains an indicator that is virtually the accumulated price over every seconds elapsed.
  We should be able to use that indicator in a smart way to overcome price manipulations coming from attackers. See https://uniswap.org/whitepaper.pdf about the price oracle implemented in Uniswap v2, which is the same as of Pancake v2. Mathematical formulation is required.

1.2 setwhitelistToken(.) function

- Logic: The admin can whitelist a token. Only whitelisted tokens can attend to a pair.
- Comment: Do we need this regulation?

1.3 pausePriceGuard(pair) map

- Logic: The creator of a pair can pause or resume the price guard on the pair.
- Comment: This could, alternatively, be implemented by the price tolerance. Because of this redundancy, the verifyPrice(.) function has un unpleasant revert:

```
    if (!priceGuardPaused[address(pair)]) {
        require(
            maxSpreadTolerance[address(pair)] > 0,
            "max spread tolerance not initialized"
        );
```

- Solution: Remove the pirceGuardPaused map. maxSpreadTolerance == 0/infinity can replace it.

1.4 The antiWhale check

- Logic: ...
- **Issue**: There is an **error** in this code. The coder does **NOT** understand how to use a pair contract. The code fragment comes below:

```
    function antiWhale(address[] memory path, uint amountIn) internal view {
        for (uint256 i = 0; i < path.length - 1; i++) {
            ICrosswisePair pair = ICrosswisePair(
                CrosswiseLibrary.pairFor(factory, path[i], path[i + 1])
            );

            if (antiWhalePerLp[address(pair)]) {
                (uint reserve0, uint reserve1,) = pair.getReserves();
                (reserve0, reserve1) = path[i] == pair.token0() ? (reserve0,
reserve1) : (reserve1, reserve0);
                uint maxTransferAmount = (reserve0 * maxTransferAmountRate) /
10000;
                require(amountIn <= maxTransferAmount, "CrssRouter.antiWhale:
Transfer amount exceeds the maxTransferAmount");
                // The coder thinks that 0 refers to the input side and 1 the
output side in the pair terminology.
                // The coder thought the same 'amountIn' entered every pair on
the path
            }
        }
    }
```

- **Comment**: The variable 'amountIn' should be updated as the swap traverses over the path of pairs.
- **Solution**: (Sketch) Place the antiWhale check in the function _swap(.), where the 'amountId' is updated along the path.

1.5 verifyPrice(.) function, pairPrice

- Logic: ...

```
  function verifyPrice(address[] memory _path, uint256 _amountIn, uint256
_amountOut) {
    ...
    for (uint256 i = 0; i < _path.length - 1; i++) {
      ...
      uint256 pairPrice = getPairPrice( _path[i], _path[i + 1], _amountIn,
_amountOut );
      ...
    }
  }
```

- **Error**: _amountIn and _amountOut were used as if they were for all the intermediary pairs on the path.
- Solution: I would better to modify the _swap(.) function, which enumerates all the intermediary pairs and the amountIn and amountOut values of those pairs.

1.6 verifyPrice(.) function, price comparison with the oracle price.

- Logic: ...
- **Problem**: There is an **issue** in this code. The error is found in the code fragment below:

```
              minPrice = minimum( oraclePrice, pairPrice );
              maxPrice = maximum( oraclePrice, pairPrice );
              upperLimit =
minPrice.mul(maxSpreadTolerance[address(pair)].add(10000)) / 10000;
              require( maxPrice <= upperLimit, 'CrosswiseRouter.verifyPrice:
verify price is failed' );
```

  This code effectively equalize the oraclePrice and pairPrice and only take into account the scaled distance between the two prices. Remember this is the place to check the pairPrice's compliance with the oraclePrice and not the reverse. If the pairPrice is less than the oraclePrice then the code effectively check the oraclePrice against the pairPrice, as if the pairPrice is the standard.
- **Solution**: Check if the exp(pairPrice) belongs to the interval (exp(oraclePrice - tolerance), exp(oraclePrice + tolerance)) This is equivalent to, oraclePrice / (1 + logTolerance) < pairPrice < oraclePrice * (1 + logTolerance). Let logTolerance be 0.1, for example. This is actually a logarithmic neighborhood of the oraclePrice. It will be developed further later.

2. Security

2.1 **transfer-wise security vs. transaction-wise security**

- Logic: The CrossWise router takes care of security on individual transfers, placing checks for antiWhale and varifyPrice on each transfers/swaps independently.
- **Issue**: The **Jan.18 attack** throw a large transaction which has 12 individual swaps from crss token to busd tokens. The attack suggests that we need to transit to transaction-wise security where the accumulated transfer amount is checked for antiWhale and the whole price change is checked in a smart way. The later versions will develop to history-wise security.
- Solution (Sketch):

- Start measuring accumulated transfer amount and price change when a new transaction begins.
- Check the accumulated quantities on every transfer or swap or removeLiquidity.

2.2 transfer-wise security vs. transaction-wise security

3. Programming

3.1 _addLiquidity(.) function

- Logic: All the same as the Pancakeswap router, but a line added:

```
  if (ICrosswiseFactory(factory).getPair(tokenA, tokenB) == address(0)) {
      ICrosswiseFactory(factory).createPair(tokenA, tokenB);
  }
  (uint reserveA, uint reserveB) = CrosswiseLibrary.getReserves(factory,
tokenA, tokenB);
  if (reserveA == 0 && reserveB == 0) {
      lpCreators[ICrosswiseFactory(factory).getPair(tokenA, tokenB)] =
msg.sender; // ------------- added line
      (amountA, amountB) = (amountADesired, amountBDesired);
  } else ...
```

- **Issue**: It would be better to place the line, which defines the pair creator, at the spot of creating the pair. Logically, if a pair was just created, then (reserveA == 0 && reserveB == 0) holds true. But the reverse is generally not proved. So (reserveA == 0 && reserveB == 0) does not necessarily means the pair was just created. And, the readability/productivity will be better if a pair creator is defined on the spot of creating the pair.
- **Solution**:
  - First, make sure the following code is the only place a pair is created.
  - Second, change the code as follows:

```
  if (ICrosswiseFactory(factory).getPair(tokenA, tokenB) == address(0)) {
      ICrosswiseFactory(factory).createPair(tokenA, tokenB);
      lpCreators[ICrosswiseFactory(factory).getPair(tokenA, tokenB)] =
msg.sender; // ------------- added line
  }
  (uint reserveA, uint reserveB) = CrosswiseLibrary.getReserves(factory,
tokenA, tokenB);
  if (reserveA == 0 && reserveB == 0) {
      (amountA, amountB) = (amountADesired, amountBDesired);
  }
```

3.2 getPrice(.) function

- Logic:

```
  function getPairPrice(address _token0, address _token1, uint256 _amountIn,
uint256 _amountOut) internal view returns (uint256) {
      (address token0,) = CrosswiseLibrary.sortTokens(_token0, _token1);
      (uint256 reserve0, uint256 reserve1) =
CrosswiseLibrary.getReserves(factory, _token0, _token1);
      (reserve0, reserve1) = token0 == _token0 ? (reserve0, reserve1) :
(reserve1, reserve0);
      (uint256 amountIn, uint256 amountOut) = _amountIn == 0 ?
        (CrosswiseLibrary.getAmountIn(_amountOut, reserve0, reserve1 ),
_amountOut) :
        (_amountIn, CrosswiseLibrary.getAmountOut(_amountIn, reserve0,
reserve1 ));

      (uint256 decimals0, uint256 decimals1) = (IBEP20(_token0).decimals(),
IBEP20(_token1).decimals());
      return (amountIn.mul(10 ** decimals1)).mul(10 ** 8) / (amountOut * (10
** decimals0));
  }
```

- **Issue**: The names of arguments and variable confuse readers, devs, and maintainers. The names come from the coders misbelief that token0 and token1, respectively, mean input and output. The coder implicitly assumes that:
    - _token0 is the input token and _amountIn is the amount of _token0.
    - _token1 is the output token and _amountOut is the amount of _token1.
    - Accordingly, the variables reserve0, reserve1, decimals0, and decimals1 should be named reserveIn, reserveOut, decimalsIn, and decimalsOut respectively.
    - Please check if the oracle prices are based on the decimals value of 8. If 10 ** (decimals1 - decimals0 + 8) is less than amountOut/amountIn, then the return value will be absolute zero.
- Solution:

```
  function getPairPrice(address _tokenIn, address _tokenOut, uint256
_amountIn, uint256 _amountOut) internal view returns (uint256) {
      (address token0,) = CrosswiseLibrary.sortTokens(_tokenIn, _tokenOut);
      (uint256 reserve0, uint256 reserve1) =
CrosswiseLibrary.getReserves(factory, _tokenIn, _tokenOut);
      (reserveIn, reserveOut) = token0 == _tokenIn ? (reserve0, reserve1) :
(reserve1, reserve0);
      (uint256 amountIn, uint256 amountOut) = _amountIn == 0 ?
        (CrosswiseLibrary.getAmountIn(_amountOut, reserveIn, reserveOut ),
_amountOut) :
        (_amountIn, CrosswiseLibrary.getAmountOut(_amountIn, reserveIn,
reserveOut ));

      (uint256 decimalsIn, uint256 decimalsOut) =
(IBEP20(_tokenIn).decimals(), IBEP20(_tokenOut).decimals());
```

```
      return (amountIn.mul(10 ** decimalsOut)).mul(10 ** 8) / (amountOut *
(10 ** decimalsIn));
  }
```