Post-Audit Report: DXswap Audit No. 2

Commit
https://github.com/levelkdev/dxswap-periphery/tree/a66b559fec01f38e2460cb6713ed24870bc7fa09

Critical Defects

Defect	Status
2.1 DoSviaOrderWithdrawal	✅ Addressed
2.2 Pool Stake Calculated Incorrectly	✅ Addressed

Moderate Defects

Defect	Status
3.1 Exogenous Price-Discovery Risks	Not addressed. General feedback
3.2 Suspicious No-op Comparison	✅ Addressed

Minor Defects

Defect	Status
4.1 Bounty Race	Not addressed. General feedback
4.2 Code Simplification (1)	✅ Addressed
4.3 Comparison With Bool Literal	✅ Addressed
4.4 Code Simplification (2)	✅ Addressed
4.5 Code Simplification (3)	✅ Addressed
4.6 Code Simplification (4)	✅ Addressed
4.7 Order Index Never Zero	✅ Addressed
4.8 Code Simplification (5)	✅ Addressed
4.9 Code Simplification (6)	✅ Addressed
4.10 Code Simplification (7)	✅ Addressed

Post-Audit Report: DXswap Audit No. 3

Commit
https://github.com/levelkdev/dxswap-periphery/tree/a66b559fec01f38e2460cb6713ed24870bc7fa09

Critical Defects

Defect	Status
2.1 Overly Permissive Fee Trading	Swapr Core – Not related to Swapr Relayer

Moderate Defects

Defect	Status
3.1 Oracle Price Manipulation	Not addressed. General feedback
3.2 Relayer Denial of Service	✅ Addressed
3.3 Unspent Allowance in Router	✅ Addressed

Minor Defects

Defect	Status
4.1 Unused Return Values in _pool()	Not addressed
4.2 Permissive Oracle Window Time	✅ Addressed

DXswap Audit No. 2

Oct, 2020

Contents

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Sunfish Technology, LLC at the request of members of dxDAO, an organization governed by a smart contract on the Ethereum blockchain. The code covered by this review (see section 1.2) is a set of contracts designed to allow dxDAO to manage DXswap, a fork of of UniswapV2.

1.2 Source Files

This audit covers code from public GitHub repository located at `https://github.com/levelkdev/dxswap-periphery`.

Only code from the following git SHA was reviewed:

```
8682cb82f994586dfddca09e0c8a662b1eeba99c
```

Within that revision, only the following files received line-by-line review:

- contracts/examples/DXswapRelayer.sol
- contracts/examples/OracleCreator.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

Chapter 2

Critical Defects

Issues discussed in this sections are defects that lead to the code to misbehave in ways that are directly exploitable and have severe consequences like loss of funds.

2.1 DoS via Order Withdrawal

The expression inside `require` on line 253 of `DXswapRelayer.sol` is incorrect:

```
require(block.timestamp < order.deadline, 'DXswapRelayer: DEADLINE_NOT_REACHED');
```

For an order to be "expired", the block timestamp should be *greater than*, not less than, the order deadline.

Since the sense of this requirement is unintentionally inverted in the code, an attacker can trivially deny any attempt to provide liquidity to a "pair" simply by withdrawing orders, as the erroneous timestamp requirement will always evaluate to `true`.

2.2 Pool Stake Calculated Incorrectly

Line 347 of `DXswapRelayer.sol` performs a division operation that can lead to arithmetic truncation:

```
uint256 poolStake = (amountA.add(amountB) / reserveA.add(reserveB))
                  .mul(PARTS_PER_MILLION);
```

Since `reserveA + reserveB` is typically expected to be larger than `amountA + amountB`, the integer division expression above ought to evaluate to `0` under most circumstances. Consequently, `poolStake` will (incorrectly) always be zero.

The correct arithmetic expression is:

```
(amountA.add(amountB)).mul(PARTS_PER_MILLION) / reserveA.add(reserveB);
```

Chapter 3

Moderate Defects

Issues discussed in this section are code defects that may lead to unintended deviations in behavior. It may be possible to chain multiple moderate defects into a working exploit.

3.1 Exogenous Price-Discovery Risks

There are a variety of exogenous risks (in this case, those outside of the view of the EVM machine model) that can cause the liquidity provisioning mechanism in `DXswapRelayer` to send assets to a pair contract at a price that deviates significantly from the prevailing price (ratio) of those assets.

Given that any funds provided to the "pair" contract that are in excess of the instantaneous spot price of those assets can be immediately arbitraged away, there are a number of foreseeable scenarios in which automated or semi-automated liquidity provisioning leads to loss of funds. A short, **incomplete** list of those scenarios includes:

- The timeliness of the price information used to provide liquidity to the pair contract necessarily depends on the timely execution of the liquidity provisioning itself. The "true" (correct) price of the pair will drift while the transaction is waiting to be executed, so executing the transaction with too low a gas price will increase the risk that an exogenous price movement leads to loss of funds. However, a transaction with too high a gas price will deterministically lead to loss of funds simply due to overpaying for gas.

- Anyone who can influence the order in which transactions are mined (i.e. miners, or anyone with a relationship with miners) can cause the liquidity provisioning transaction to occur between trades that temporarily move the price of the asset, thereby creating a window in which to perform a risk-less arbitrage of the excess assets provided to the pair.

The "price oracle" mechanism used by the `DXswapRelayer` contract is a *mitigation* for the issues above, as it can deterministically limit the magnitude of the loss-of-funds, but it does not eliminate those risks. Additionally, the fact that the `DXswapRelayer` will refuse to provide liquidity under circumstances that could be attacker-influenced means that exogenous price manipulation could provide a vector for denial-of-service.

A second-order risk created by the complexity of correctly pricing pair liquidity contributions is that DAO members will not be well-equipped to evaluate the costs and benefits of a liquidity provisioning action on behalf of the DAO. Measuring the price risks outlined above requires a detailed understanding of the Ethereum block protocol *and* the social and economic dynamics of miners and decentralized exchanges more generally; very few people have the technical background and financial numeracy necessary to model those risks accurately. Concretely, if we **simplify** a liquidity provisioning operation to a futures contract that provides a fixed yield, and we simplify gas pricing such that we can reliably achieve execution within a fixed window of time, we would still need an accurate model of token price volatility to correctly model the risk of price movement during the period of time in which the liquidity provisioning transaction was waiting to be mined. (Note that the two simplifications in the previous sentence are also implausible to make in practice: the "swap fees" that generate yield are not an easily-predictable cash flow, and the relationship between gas price bids and transaction execution latency are stochastic and highly volatile from minute to minute.)

3.2 Suspicious No-op Comparison

The expression `order.amountB <= order.amountB` on line 213 of `DXswapRelayer.sol` always evaluates to `true`:

```
require(amountA <= order.amountA || order.amountB <= order.amountB,
                'DXswapRelayer: INVALID PRICES');
```

It is *possible* that the following was intended:

```
require(amountA <= order.amountA || amountB <= order.amountB);
```

However, it's not clear why this code needs to consult a price oracle to determine both `amountA` and `amountB`, as the oracle "price" is really a ratio of `amountA` to `amountB`, and thus determining one value from the other ought to yield the same information. Moreover, the pattern of specifying a minimum amount of both `amountA` and `amountB` that needs to be spent elides the central question of whether or not the *ratio* of the two has changed at all. (The range of possible ratios of `A : B` can be determined indirectly through determining which ratios satisfy the condition that neither value have changed beyond `priceTolerance`, but that is less obviously correct than examining the ratio directly, and it is possible that there are additional bugs in this code related to the unsound management of "minimum" token quantities with respect to liquidity provisioning.)

Chapter 4

Minor Defects

Issues discussed in this sections are subjective code defects that affect readability, reliability, or performance.

This section includes comments designed to reduce the number of local variables and control constructs presented to the solidity compiler, as developers noted in a pre-audit call that they ran into solidity "stack too deep" errors. (As of this writing, the solidity compiler does not implement live range analysis, value range analysis, or common sub-expression elimination, so developers generally have to work around those defects manually when contracts consume too many resources.)

4.1 Bounty Race

An external caller of `updateOracle()` is compensated `0.1 ether` plus approximately the gas cost of the transaction. Consequently, multiple people (or bots) may attempt to update the oracle simultaneously, but only one will succeed, and the rest will be penalized the gas cost of the failed transaction. Consequently, the expected value of updating the oracle is actually *lower* than `BOUNTY`; in principle it ought to be arbitraged down to zero. In other words, the expected outcome is that slightly more than `0.1 ether` (or equivalent mining resources) are spent attempting to update the oracle state.

In general, patterns that unconditionally reward a race to update some on-chain state simply create an opportunity for miners to harvest additional fees through their control of transaction ordering.

4.2 Code Simplification (1)

Lines 98 through 104 of `DXswapRelayer.sol` duplicate `TransferHelper.safeTransferFrom(tokenB, owner, address(this), amountB)`. Simply remove that expression from the consequent and alternative blocks of the `if ... else` and place it in the immediate post-dominator.

4.3 Comparison With Bool Literal

The comparison with `false` on line 191 of `DXswapRelayer.sol` is unnecessary:

```
require(order.executed == false, 'DXswapRelayer: ORDER_EXECUTED');
```

Instead, the code can simply negate the expression:

```
require(!order.executed, 'DXswapRelayer: ORDER_EXECUTED');
```

4.4 Code Simplification (2)

The code from lines 198 to 210 in `DXswapRelayer.sol` are largely identical, save for the expression that produces the `token` provided to `oracleCreator.consult()`:

```
if(tokenA == address(0)){
    amountB = oracleCreator.consult(
        order.oracleId,
        IDXswapRouter(dxSwapRouter).WETH(),
        order.amountA
    );
} else {
    amountB = oracleCreator.consult(
        order.oracleId,
        tokenA,
        order.amountA
    );
}
```

The code could simply read as follows:

```
amountB = oracleCreator.consult(
    order.oracleId,
    tokenA == address(0) ? IDXswapRouter(dxSwapRouter).WETH() : to-
kenA,
    order.amountA
);
```

This simplification ought to reduce the number of EVM instructions emitted by the solidity compiler, as it will only have to generate one call sequence.

4.5 Code Simplification (3)

The consequent and alternative blocks of the `if ... else` statement on lines 261 through 267 of `DXswapRelayer.sol` duplicate code for transferring `tokenB`:

```
if (tokenA == address(0)) {
    TransferHelper.safeTransferETH(owner, amountA);
    TransferHelper.safeTransfer(tokenB, owner, amountB);
} else {
    TransferHelper.safeTransfer(tokenA, owner, amountA);
    TransferHelper.safeTransfer(tokenB, owner, amountB);
}
```

Those lines could instead read:

```
if (tokenA == address(0)) {
    TransferHelper.safeTransferETH(owner, amountA);
} else {
    TransferHelper.safeTransfer(tokenA, owner, amountA);
}
TransferHelper.safeTransfer(tokenB, owner, amountB);
```

4.6 Code Simplification (4)

The `if ... else` tree on lines 349 through 359 of `DXswapRelayer.sol` performs redundant comparisons in each `else if` condition expression:

```
if(poolStake < 1000) {
    windowTime = 30;
} else if (poolStake >= 1000 && poolStake < 2500){
    windowTime = 60;
} else if (poolStake >= 2500 && poolStake < 5000){
    windowTime = 90;
} else if (poolStake >= 5000 && poolStake < 10000){
    windowTime = 120;
} else {
    windowTime = 150;
}
```

The following code ought to be semantically equivalent and generate fewer EVM instructions:

```
if (poolStake < 1000) {
    windowTime = 30;
} else if (poolStake < 2500) {
    windowTime = 60;
} else if (poolStake < 5000) {
    windowTime = 90;
} else if (poolStake < 10000) {
    windowTime = 120;
} else {
    windowTime = 150;
}
```

4.7 Order Index Never Zero

The implementation of `_OrderIndex()` ensures that `orderIndex` is never zero, which in turn requires that `executeOrder()` manually verify that the order index that it receives is non-zero.

Changing `_OrderIndex()` to use post- instead of pre-increment (in other words, zero-indexed) order indices would simplify the implementation of `executeOrder()`.

4.8 Code Simplification (5)

On lines 367 through 373 in `DXswapRelayer.sol`, the code in the consequent and alternative blocks of the `if` statement are functionally identical:

```
if(factory == dxSwapFactory){
    tokenA = tokenA == address(0) ?
           IDXswapRouter(dxSwapRouter).WETH() : tokenA;
    tokenB = tokenB == address(0) ?
           IDXswapRouter(dxSwapRouter).WETH() : tokenB;
} else if(factory == uniswapFactory) {
    tokenA = tokenA == address(0) ?
           IDXswapRouter(uniswapRouter).WETH() : tokenA;
    tokenB = tokenB == address(0) ?
           IDXswapRouter(uniswapRouter).WETH() : tokenB;
}
```

Instead, the code can take advantage of the fact that `_pair()` is an internal function that is only called when `tokenA < tokenB`, so `tokenA` is the only token address that could be zero:

```
if (tokenA == 0 && (factory == dxSwapRouter || factory == uniswapRouter))
{
    tokenA = IDXswapRouter(factory).WETH();
}
```

```
// rest of the function ...
```

Contextually, it may be appropriate to insert the following instead:

```
require(factory == dxSwapRouter || factory == uniswapRouter);
if (tokenA == 0) tokenA = IDXswapRouter(factory).WETH();
// rest of function ...
```

4.9 Code Simplification (6)

Lines 92:97 of `OracleCreator.sol` have extended common sub-expressions in the consequent and alternative blocks of the `if ... else` statement:

```
if (token == oracle.token0) {
    amountOut = oracle.price0Average.mul(amountIn).decode144();
} else {
    require(token == oracle.token1, 'OracleCreator: INVALID_TOKEN');
    amountOut = oracle.price1Average.mul(amountIn).decode144();
}
```

Instead, the code could read:

```
if (token == oracle.token0) {
    avg = oracle.price0Average;
} else {
    require(token == oracle.token1, 'OracleCreator: INVALID_TOKEN');
    avg = oracle.price1Average;
}
amountOut = avg.mul(amountIn).decode144();
```

Hoisting the common function calls (or inlined arithmetic) into the immediate post-dominator of the conditional blocks ought to reduce the size of the generated code. Developers cannot rely on the solidity compiler to eliminate common sub-expressions across basic blocks.

4.10 Code Simplification (7)

The ternary expression on line 101 is superfluous:

```
function isOracleFinalized(uint256 oracleIndex)
         external view returns (bool){
    return oracles[oracleIndex].observationsCount == 2 ? true : false;
}
```

The following code is semantically identical:

```
function isOracleFinalized(uint256 oracleIndex)
         external view returns (bool) {
    return oracles[oracleIndex].observationsCount == 2;
}
```

Chapter 5

Other Notes

- Both of the critical bugs discovered during this review could have been identified with unit testing. Considering the number of style comments along with the number of textually trivial (but semantically meaningful) bugs in this review, the code would likely benefit from some additional development time. Bugs like section 2.1 and section 3.2 are difficult to uncover in manual review, because they are lexicographically very close to the correct code. Consequently, it is cheapest and most effective to eliminate those sorts of bugs with testing.

DXswap Audit No. 3

Dec, 2020

Contents

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Sunfish Technology, LLC at the request of members of dxDAO, an organization governed by a smart contract on the Ethereum blockchain. The code covered by this review (see section 1.2) is a set of contracts designed to allow dxDAO to manage DXswap, a fork of of UniswapV2. Part of this review is a re-review of code already evaluated in a previous audit. Unresolved issues from the previous review are included at the end of this report.

1.2 Source Files

This audit covers code from two public GitHub repositories:

- `https://github.com/levelkdev/dxswap-periphery`
- `https://github.com/levelkdev/dxswap-core`

Only code from the following git SHAs was reviewed:

```
dxswap-periphery 8682cb82f994586dfddca09e0c8a662b1eeba99c
dxswap-core      a0def5380a43e9ee8b9b048e3379a93aac9bb05d
```

Within those revisions, only the following files received line-by-line review:

- dxswap-periphery/contracts/examples/DXswapRelayer.sol
- dxswap-periphery/contracts/examples/OracleCreator.sol
- dxswap-core/contracts/DXswapFeeReceiver.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

Chapter 2

Critical Defects

Issues discussed in this sections are defects that lead to the code to misbehave in ways that are directly exploitable and have severe consequences like loss of funds.

2.1 Overly Permissive Fee Trading

The `DXswapFeeReceiver.takeProtocolFees()` function can be called by any address, and that call in turn will call any contract provided as an argument to it. This presents a couple security risks:

- For any tokens owned by the `DXswapFeeReceiver` that are not `WETH`, the contract will immediately exchange those tokens for Ether uncritically without examining the market depth or current price of the pair contract satisfying the trade. Consequently, it would be relatively straightforward to engage in market manipulation near or around the exchange of fee tokens for Ether.
- It may be possible to get the `takeProtocolFees()` function to behave in undesirable ways in terms of making arbitrary calls to foreign contracts, since it does no validation of the addresses that it treats as pair contracts.

One practical exploit vector for `WETH` exchanges would be for an attacker to perform a triangular flash loan attack to manipulate the token exchange rate as part of a transaction that calls `takeProtocolFees()`. That attack would work as follows, given a token `FOO` that is being traded for `WETH`.

1. An attacker margins a large quantity of `FOO` from a trading pair that is not `WETH-FOO`.
2. The attacker uses the large quantity of borrowed `FOO` to sell `FOO` for `WETH`, driving down the price of `FOO` per `WETH`.
3. The attacker calls `takeProtocolFees()` on the `DXswapFeeReceiver` contract, which causes it to sell `FOO` at the current (low) price. This drives down the price of `FOO` even further.

4. The attacker buys `FOO` from the `FOO-WETH` pair, this time at a lower price due to previous step. Consequently, they receive more `FOO` for the same quantity of `WETH` received in step 2.

5. The attacker returns the loan of `FOO` from the first step and either pockets the difference between the loaned quantity and the quantity purchased in step 4.

Chapter 3

Moderate Defects

Issues discussed in this section are code defects that may lead to unintended deviations in behavior. It may be possible to chain multiple moderate defects into a working exploit.

3.1 Oracle Price Manipulation

Since any address can call `DXswapRelayer.updateOracle()`, the price that the oracle witnesses can be partly manipulated by an attack similar to that of section 2.1. Since the price that the oracle witnesses is integrated over a period of time, and since the oracle insists on a particular price tolerance, it is more challenging to exploit this price manipulation profitably. In any case, dxDAO's developers should be aware that it is still possible to manipulate the trading price witnessed by the `OracleCreator` due to the unpermissioned nature of `updateOracle()`.

3.2 Relayer Denial of Service

The `DXswapRelayer` contract requires that it posses a sufficient ERC20 balance for calls to `orderLiquidityProvision()`. However, the anyone can send the ERC20 (or Ether) balance of the `DXswapRelayer` contract to its `owner` at any time by calling `ERC20Withdraw()` (or `ETHWithdraw()`). Consequently, if the appropriate ERC20 balance is not transferred to the `DXswapRelayer` as part of *the same transaction* as the call to `orderLiquidityProvision()`, an attacker can cheaply cause calls to fail spuriously due to insufficient funds.

3.3 Unspent Allowance in Router

In `DXswapRelayer._pool()`, the allowance provided to the `DXswapRouter` contract may not be entirely consumed by the call to `addLiquidity()`. The allowance of the `DXswapRouter` contract should be reset to zero after the call completes.

Note that some tokens will refuse a call to `approve()` if the allowance isn't currently zero, so this issue could result in calls to `_pool()` failing spuriously when the existing allowance for `DXswapRouter` is already non-zero.

Chapter 4

Minor Defects

Issues discussed in this sections are subjective code defects that affect readability, reliability, or performance.

4.1 Unused Return Values in `_pool()`

The following external call in `DXswapRelayer._pool()` assigns its return values to local variables, but then the local variables are left unused:

```
(amountA, amountB, liquidity) = IDXswapRouter(dxSwapRouter).addLiquidity(
    _tokenA,
    _tokenB,
    _amountA,
    _amountB,
    _minA,
    _minB,
    address(this),
    block.timestamp
);
```

For safety, it may make sense to check that the return values for `amountA` and `amountB` are sane, and that `liquidity` is within the expected range.

4.2 Permissive Oracle Window Time

The safety of the price oracle for the `DXswapRelayer` contract depends on the size of the provided `maxWindowTime` parameter, as well as a suggested window computed by `_consultOracleParameters()`. However, the code accepts zero as an acceptable `maxWindowTime` parameter, which means that in principle the price oracle can be manipulated as part of an oracle `update()` transaction.

Although it is unlikely that the DAO managing the `DXswapRelayer` would approve a call with a zero window size, it would be safer to disallow unreasonably small window sizes.

Chapter 5

Appendix

5.1 Unresolved Issues

Issues in this section apply to code covered from a previous review. These issues have been left unfixed.

5.1.1 Exogenous Price-Discovery Risks

There are a variety of exogenous risks (in this case, those outside of the view of the EVM machine model) that can cause the liquidity provisioning mechanism in `DXswapRelayer` to send assets to a pair contract at a price that deviates significantly from the prevailing price (ratio) of those assets.

Given that any funds provided to the "pair" contract that are in excess of the instantaneous spot price of those assets can be immediately arbitraged away, there are a number of foreseeable scenarios in which automated or semi-automated liquidity provisioning leads to loss of funds. A short, **incomplete** list of those scenarios includes:

- The timeliness of the price information used to provide liquidity to the pair contract necessarily depends on the timely execution of the liquidity provisioning itself. The "true" (correct) price of the pair will drift while the transaction is waiting to be executed, so executing the transaction with too low a gas price will increase the risk that an exogenous price movement leads to loss of funds. However, a transaction with too high a gas price will deterministically lead to loss of funds simply due to overpaying for gas.
- Anyone who can influence the order in which transactions are mined (i.e. miners, or anyone with a relationship with miners) can cause the liquidity provisioning transaction to occur between trades that temporarily move the price of the asset, thereby creating a window in which to perform a risk-less arbitrage of the excess assets provided to the pair.

The "price oracle" mechanism used by the `DXswapRelayer` contract is a *mitigation* for the issues above, as it can deterministically limit the magnitude of the loss-of-funds, but it does not eliminate those risks. Additionally, the fact that the `DXswapRelayer` will refuse to provide liquidity under circumstances that could be attacker-influenced means that exogenous price manipulation could provide a vector for denial-of-service.

A second-order risk created by the complexity of correctly pricing pair liquidity contributions is that DAO members will not be well-equipped to evaluate the costs and benefits of a liquidity provisioning action on behalf of the DAO. Measuring the price risks outlined above requires a detailed understanding of the Ethereum block protocol *and* the social and economic dynamics of miners and decentralized exchanges more generally; very few people have the technical background and financial numeracy necessary to model those risks accurately. Concretely, if we **simplify** a liquidity provisioning operation to a futures contract that provides a fixed yield, and we simplify gas pricing such that we can reliably achieve execution within a fixed window of time, we would still need an accurate model of token price volatility to correctly model the risk of price movement during the period of time in which the liquidity provisioning transaction was waiting to be mined. (Note that the two simplifications in the previous sentence are also implausible to make in practice: the "swap fees" that generate yield are not an easily-predictable cash flow, and the relationship between gas price bids and transaction execution latency are stochastic and highly volatile from minute to minute.)