DXswap Audit No. 1

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 *fork* of an existing "decentralized exchange" product called UniswapV2. This audit covers **only the changes** to the upstream source code, not the entirety of the source.

1.2 Source Files

This audit covers code from public GitHub repositories located at `https://github.com/levelkdev/dxswap-core` and `https://github.com/levelkdev/dxswap-periphery`, both of which contain the complete git history of the upstream source code. Since this review specifically covers the changes to upstream code, the commit ranges that were used for producing diffs for review are as follows:

```
dxswap-core 8160750...ca021f4
dxswap-periphery a86e696...44cf11e
```

Within those commit ranges, only the changes to the following files were reviewed:

- dxswap-core/contracts/DXswapERC20.sol
- dxswap-core/contracts/DXswapPair.sol
- dxswap-core/contracts/DXswapFactory.sol
- dxswap-core/contracts/DXswapDeployer.sol
- dxswap-periphery/contracts/libraries/DXswapLibrary.sol
- dxswap-periphery/contracts/DXswapRouter.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

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 DXswapPair: Zero Fee Skips Invariant

The following code in `DXswapPair.swap()`, which needs to be executed in order to enforce the "constant-product invariant" for swaps, does not execute when `swapFee` is zero:

```
if (swapFee > 0) {
  uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(swapFee));
  uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(swapFee));
  require(balance0Adjusted.mul(balance1Adjusted) >=
      uint(_reserve0).mul(_reserve1).mul(10000**2), 'DXswapPair: K');
}
```

Consequently, a proposal that sets `swapFee` to zero will inadvertently **allow any swap for any quantity of inputs and outputs to succeed.** In other words, an attacker can drain all of the reserve tokens at once when the swap fee is zero. The purpose of the `require` statement within this block is not just to ensure that the `swapFee` is paid; it also asserts the trading invariant for swaps.

Removing the conditional block around these three lines of code remediates this issue.

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 Multiple Solidity Versions

The `Babylonian` and `FixedPoint` libraries in `dxswap-periphery` want solidity version `0.4`, but the `AddressStringUtil` and `SafeERC20Namer` libraries want solidity `0.5`. Since the rest of the `dxswap-periphery` code is compiled with solidity `0.5`, the `Babylonian` and `FixedPoint` libraries should be changed to use that solidity version, and they should be audited for behavior changes caused by the solidity version change. (The solidity compiler is not strictly backwards-compatible across minor version bumps like `0.4 => 0.5`.)

Note that the code in question here is not within the scope of this review.

3.2 Loop Induction Variable Overflow

The loops in `DXswapDeployer.sol` on lines 31 and 54 use an induction variable of type `uint8` with arbitrary loop bounds, which makes these loop bounds trivial to intentionally overflow. The particular context of the code in question makes these overflows uninteresting from an exploitation perspective, but they are still present nonetheless.

Using integer types other than `uint` or `uint256` in solidity is usually unnecessary. Counterintuitively, integer types narrower than 256 bits are **more** expensive from a gas utilization perspective, as they require that the compiler generate appropriate masking operations. (The EVM does not support native operations on scalars narrower than 256 bits.)

Chapter 4

Minor Defects

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

4.1 Y2038 Timestamp Overflow

The price accumulator mechanism in `DXswapPair` uses block timestamps modulo $2^3$2. Consequently, this code cannot be expected to work once Unix time exceeds $2^3$2 seconds (in the year 2038) *or* if the Ethereum blockchain makes changes to the semantics of `block.timestamp` in future forks.

4.2 Miner Price Oracle Manipulation

The price oracle mechanism described in the UniswapV2 whitepaper is not as robust to manipulation as intended.

The authors of the whitepaper correctly determine that the price oracle can be manipulated by anyone who is able to control all of the transactions associated with a particular swap contract for a whole block. What they do not address is that it is perfectly feasible for a miner to refuse to mine transactions associated with a particular swap contract **without absorbing serious economic costs** during periods where transaction volume is high. When there is significant demand for the execution of Ethereum transactions, miners have a tremendous amount of latitude when it comes to determining which transactions to execute, as they can easily fill a block with high-transaction-fee transactions. (They can censor transactions when demand is low, too, but then it is clear that they would have to forego some transaction fee revenue.)

As a consequence, it may be possible for one miner (with lots of hash-power) to cheaply manipulate the price oracle mechanism for short periods of time.

Chapter 5

Other Notes

- The current git state of `levelk/dxswap-core` and `levelk/dxswap-periphery` will not make it easy to re-apply changes over a different upstream base commit. Ideally, the set of patches that dxDAO maintains on top of the Uniswap upstream code should be easy to rebase on top of upstream bugfixes. Presently there are dozens of commits on top of the upstream code, plus merge commits, which would make a `git rebase` rather cumbersome. Moreover, rebasing the current changes will just become more laborious over time.

  Consider rebasing all of the current changes over the UniswapV2 v1.0.1 HEAD and squashing all of the changes into a small (half-dozen or fewer) number of patches that can be applied and/or fixed up independently. Also, consider hinting to `git` that some "new" files are actually re-named files from upstream (via appropriate invocation of `git mv`). The current history contains non-minimal diffs in part because the developers forgot to inform git about some file renaming. Additionally, the developers made a number of superficial naming changes to internal functions, interfaces, and contracts that are not user-visible but nonetheless contribute substantially to the maintenance burden of this patch set. Consider reverting any changes that aren't strictly necessary to achieve the desired functionality.

- The UniswapV2 "constant-product" price mechanism creates an incentive for swaps to be broken up into small pieces so that slippage gets arbitraged away. Consequently, we can expect that users will spend more on gas for swaps than they would otherwise. (Note that constant-product pricing makes slippage non-linear: larger trades get exponentially worse price execution.)

- The `dxswap-core` and `dxswap-periphery` repositories contain a number of generated build artifacts. It is generally best practice not to check in any build artifacts to version control, for two reasons: First, changes to these build artifacts ruins the usefulness of tools like `git log -p`, and second, it will conceal whether or not the artifacts are actually reproducible across developers using different build environments. Consider setting up a continuous integration environment that performs "porcelain" builds of the source code and archives them, and have developers routinely test whether or not they can reporduce those artifacts.