VinuSwap Security Companion Document

Samuele Marro

September 4, 2023

1 Introduction

This document provides relevant security-related information for the `VinuSwap` contract ecosystem, and explains the security rationale behind several design decisions. It should be read in conjunction with the security audits of the Uniswap v3 core[1] and periphery[2] contracts. This document is not meant to replace the documentation of the code, but rather provide insight into potential challenges and considerations when using, maintaining and upgrading the contracts.

2 Background on Uniswap v3

While the Uniswap v3 contract is very flexible, gas-efficient and secure, these features come at the cost of increased complexity. This section aims to provide some background information on how Uniswap v3 operates.

2.1 Contracts

The main contracts used in Uniswap v3 can be divided in two categories:

- Core contracts:
  - `UniswapV3Pool`: A pool for a given `TOKEN0-TOKEN1` pair, where `TOKEN0` is the ERC20 having a lower address;
  - `UniswapV3PoolDeployer` (which inherits from `UniswapV3Factory`): Deploys a pool;
- Periphery contracts:
  - `SwapRouter`: Provides a user-friendly interface to `UniswapV3Pool`;
  - `NonfungiblePositionManager`: Packages a `UniswapV3Pool` position into an NFT;
  - `NonfungibleTokenPositionDescriptor`: Computes an on-chain JSON description of a `NonfungiblePositionManager` NFT.

2.2 Uniswap Math

Uniswap v3 uses fixed-point precision in Q64.96 format (i.e. 64 bits for the integer part and 96 bits for the decimal part). Additionally, the prices are expressed in "ticks", which correspond to the logarithm in base 1.0001 of the price. For more information on the topic, refer to the Uniswap blog post on the topic[3], as well as the primer by Ryan James Kim[4].

[1] https://github.com/Uniswap/v3-core/tree/main/audits
[2] https://github.com/Uniswap/v3-periphery/tree/main/audits
[3] https://blog.uniswap.org/uniswap-v3-math-primer
[4] https://ryanjameskim.medium.com/uniswap-v3-part-2-ticks-and-fee-acounting-explainer-with-toy-example-e9bf4d706884

2.3 Pool Indexing

The address of a Uniswap pool is computed deterministically. This allows contracts such as `SwapRouter` to perform calls to a pool without storing its address beforehand. The address depends on three variables:

- The address of `TOKEN0`;
- The address of `TOKEN1`;
- The fee of the pool.

Refer to `contracts/periphery/libraries/PoolAddress.sol` for an implementation of the pool address algorithm.

3 Contract Structure

The VinuSwap ecosystem includes all the contracts mentioned in Section 2.1, with some renamings:

- `UniswapV3Pool` is renamed to `VinuSwapPool`;
- `UniswapV3Factory` is renamed to `VinuswapFactory`;
- `UniswapV3PoolDeployer` is renamed to `VinuSwapPoolDeployer`.

In addition, two new contracts are introduced:

- `Controller`: Collects the revenue of the pools, splits it between the various parties, and manages the fee structure of the pools;
- A contract implementing the `IFeeManger` interface, of which two are provided as example implementations:
  - `NoDiscount`: Applies no discounts on the fees. Used to achieve the standard behavior of Uniswap v3;
  - `TieredDiscount`: Applies a discount based on the token balance of `tx.origin` for a given ERC20.

In typical setups, the contracts have a chain of ownership as follows:

1. `Controller` has an arbitrary owner (which can be either an EOA or another contract);
2. `Controller` owns `VinuSwapFactory`;
3. `VinuSwapPool` and `VinuSwapFactory` have the same owner (which is enforced by `VinuSwapPool` querying `VinuSwapFactory` to know its owner).

4 Contract Differences

VinuSwap was developed by modifying the original contracts as little as possible, in order to preserve the original behavior of Uniswap v3. The following is a list of the modifications applied to the various contracts:

- `VinuSwapPool`:
  - The constructor retrieves an additional parameter `feeManager` from `VinuSwapDeployer.parameters()`;
  - In `swap`, the contract first calls `feeManager.computeFee(fee)` to retrieve the actual fee to be applied;
  - `initialize` can only be called by the owner;
  - `flash` is removed in order to reduce the contract size;

- `VinuSwapFactory`:
  - `createPool` accepts two additional parameters, `tickSpacing` and `feeManager`;
  - `createPool` can only be called by the owner;
  - `enableFeeAmount` is removed, as well as its corresponding event `FeeAmountEnabled`;
- `VinuSwaPoolDeployer`:
  - In order to reduce the contract size, `parameters()` is no longer saved as a struct, but as a list of variables. For the same reason, such parameters are not deleted at the end of the deployment;
- `NonfungiblePositionManager`:
  - The struct `Position` has an additional field `lockedUntil`;
  - `positions` returns an additional value `lockedUntil`;
  - `tokensOwed0` and `tokensOwed1` are returned by a new function `tokensOwed`, instead of `positions`;
  - A new function `lock` allows users to lock a position until a deadline. A locked position cannot have its liquidity decreased (which is a requirement for exiting the position) until the deadline;
  - `decreaseLiquidity` cannot be called if the position is locked.

5 Deployment Process

The recommended steps to deploy the contracts are:

1. Deploy `Controller`;
2. Deploy `VinuSwapFactory`;
3. Transfer ownership of `VinuSwapFactory` to `Controller` by calling `VinuSwapFactory.setOwner`;
4. Deploy `SwapRouter`;
5. Deploy the `NFTDescriptor` library;
6. Deploy `NonfungibleTokenPositionDescriptor`, using `NFTDescriptor` as linked library;
7. Deploy `NonfungiblePositionManager`;
8. Deploy a contract that supports `IFeeManager` (e.g. `TieredDiscount`);
9. For each pool:
   (a) Deploy `VinuSwapPool` by calling `Controller.createPool`;
   (b) Initialize the pool by calling `Controller.initialize`;
   (c) Set the protocol fee of the pool by calling `Controller.setFeeProtocol`.

The file `scripts/deploy.ts` contains an example deployment that follows this checklist.

6 Security-Related Design Decisions

This section lists several design decisions that are related to contract security. The intention is to allow future maintainers and/or developers to understand the rationale behind some design decisions in order to avoid accidentally creating vulnerabilities. For the sake of brevity, design decisions that were already present in Uniswap's original implementation are not discussed.

6.1 Contract Version Is 0.7.6

In order to reduce the number of modifications to the contract, the contract version was not changed and was thus kept at 0.7.6. The most noteworthy difference between the current Solidity version (0.8.21, at the time of writing) is the lack of integrated overflow checking, which is however handled by the `SafeMath` library, developed by OpenZeppelin[5].

6.2 Controller Dust Goes to First Payee

When splitting the token revenue between `Controller` payees, the dust (i.e. the remainder of the division, caused by approximation errors) is transferred to the first payee.

6.3 Discounts Use tx.origin

`TieredDiscount`, which gives a discount based on an ERC20 balance, uses `tx.origin` to identify the user. This design choice has the effect of allowing users to obtain discounts regardless of the chain of calls, but also implies that per-contract discounts are not allowed.

6.4 Fee Cannot Be Higher Than Base Fee

As a sanity check, the fee computed by the fee manager cannot be higher than the base fee used by `VinuSwapPool`. This provides a form of insurance against potential fee manager attacks.

6.5 Pools Can Only Be Deployed by Controller

Since the deterministic pool address computation implies that there can be only one pool for each (`TOKEN0`, `TOKEN1`, `fee`) triple, and since some pool parameters (e.g. the type of fee manager) are highly dependent on the business decisions of the `Controller` owners, the `VinuSwapPoolDeployer` contract is designed to only allow its owner (i.e. `Controller`) to deploy pools.

6.6 Controller Fees Are Distributed Using Pull Payments

In order to bound the gas of a `Controller.collectProtocolFees` call, the fees are distributed using pull payments: each payee has its own balance, and in order to withdraw the funds the payee has to call `Controller.withdraw`. This approach also ensures that payees cannot interfere with the fund distribution of each other.

6.7 collectProtocolFees Can Only Be Called by Controller Shareholders

Since `Controller.collectProtocolFees` allows the contract to interact with any arbitrary contract, in order to limit potentially malicious contract calls, only `Controller` shareholders (i.e. the payees and the owner) can call `Controller.collectProtocolFees`.

6.8 setFeeProtocol Can Only Be Called by the Controller Owner

Unlike `Controller.collectProtocolFees`, due to its sensitive nature `Controller.setFeeProtocol` can only be called by the contract owner.

6.9 VinuSwapPool.initialize Can Only Be Called by the Factory Owner

Since the onus of deploying a pool has been transferred to the owner of `VinuSwapFactory` (which is often `Controller`), in keeping in line with this philosophy, the initialization is also assigned to the owner, which thus has full control over the setup process of the pool.

[5] https://docs.openzeppelin.com/contracts/2.x/api/math

6.10 Controller.collectProtocolFees Does Not Use the Return Value

`VinuSwapPool.collectProtocol`, the function called by `Controller.collectProtocolFees`, returns the amount of `token0` and `token1` that were collected. However, to ensure greater compatibility with non-standard (and potentially unaudited) contracts, `Controller.collectProtocolFees` will not use the return values, but rather check the difference in the token balances of `Controller` between before and after the call to `VinuSwapPool.collectProtocol`.

7 Potential Security Concerns

This section details potential aspects of the contracts that users and maintainers need to be aware of. While none of these can be considered vulnerabilities, it is still important to keep in mind these phenomena while interacting with, deploying and upgrading the contracts.

7.1 Fee Manager Can Block Pool Execution

Since `VinuSwapPool` calls a fee manager to compute the actual fee of a swap, it is possible for the fee manager to revert and prevent all swaps. For this reason, fee managers should be as simple as possible and avoid reverting in `computeFee`.

7.2 Modifying the VinuSwap Contract Changes the Pool Address

A hard-to-debug mistake when upgrading or modifying the contract is caused by the address computation system in *contracts/periphery/libraries/PoolAddress.sol*, which uses a hardcoded value for the init code hash. This value must be recomputed every time a modification is performed to `VinuSwapPool` or to any of its parent contracts. In addition to the existing init pool hash computation code in the original Uniswap v3 test files, a helper contract `PoolInitHelper` is provided, which will return the init code hash of the `VinuSwapPool` contract it is compiled with.

7.3 Token Ordering

When creating a pool, the contract addresses `tokenA` and `tokenB` will be automatically reordered into `token0` and `token1`, based on the address ranking (e.g. if `tokenA` = `0xFFF...` and `tokenB` = `0xEEE...`, then `token0` will be `0xEEE...` and `token1` will be `0xFFF...`). Therefore, both interacting contracts and Web3 applications should not assume that the token order will be the same as the order in which `tokenA` and `tokenB` were passed to the deployer.