LimitRanger Audit Report 19.08.2022

Process

One independent security expert audited the LimitRanger codebase and verified its correct integration with all third-party contracts.

Scope

The codebase was delivered as a GitHub repository:
https://github.com/LimitRangerOrg/contract
The audited commit hash is `a42472914202eefe207fbbd6b32957a6f2703f10` with the following contract files in scope:

```
contracts
├── LimitRanger.sol
```

Intended Behavior

LimitRanger allows users to create limit orders on Uniswap V3. The user provides a price range above the current spot price. When the Uniswap tick price is above the upper limit of the range, everyone can close the position, after which the funds are transferred to the creator of the position. A fee is charged by the protocol and there is a reward for closing positions, which incentivizes closing positions.

Issues

The following issues were found during the audit of LimitRanger:

Issue #	Description	Severity	Status
1	ETH Handling	Major	Resolved
2	System Susceptible To Flash Loan Attacks	Major	Acknowledged
3	Non-Compliant ERC20 Tokens Not Supported	Minor	Resolved
4	User Can Avoid Paying Fees In Certain Circumstances	Minor	Resolved
5	Opening Positions With Zero Liquidity Possible	Informational	Resolved
6	Solidity Version	Informational	Resolved
7	WETH Handling	Informational	Resolved

The issues are described in more detail below.

Critical Issues

No critical issues were found.

Major Issues

1. ETH Handling

In `mintNewPosition`, there are two issues regarding the handling of ETH (when the relevant token is WETH):

- It is not validated that the user has called the function with the sufficient ETH amount.
- When the user (accidentally) pays more ETH than `params.token0Amount` / `params.token1Amount`, he is not reimbursed.

These two issues combined can lead to the following behavior: When the balance of the contract is larger than 0 for some reason (for instance because a previous user paid too much), an attacker can call `mintNewPosition` with WETH as token0 and the balance as `token0Amount`, without providing any ETH. The ETH of the contract will be used and the attacker can immediately cancel the position to get this ETH. This also makes the `retrieveEth` function somewhat unnecessary, as everyone can retrieve ETH of the contract in practice.

Recommendation
Check that `msg.value == ethAmount`.

2. System Susceptible To Flash Loan Attacks

Because the current tick price of the pool is used for determining if a position can be closed, the system is susceptible to flash loan attacks. An attacker can take out a flash loan, manipulate the tick price of the pool, call `stopPosition` on the relevant position(s) to get the reward(s) and pay back the flash loan. In such a scenario, the user's tokens are converted although the upper tick was not reached.
This attack will not be economically feasible for pools with a lot of liquidity, but could be for low-liquidity pools.

Recommendation
This attack can be avoided by using time-weighted average prices (TWAP), see https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles for more details on how to build a TWAP oracle based on a Uniswap pool. Using a TWAP would generally be beneficial for pools with low liquidity, as the spot prices of those can deviate from the market prices over short periods of time.
If using a TWAP oracle is not desirable, users should be informed about the risk with low-liquidity pools.

Update
The team acknowledged the issue:

We acknowledge that a position could be force closed by using a flash loan attack when using a low liquidity pool which is problematic if the main intent of the position was to accrue fees. However closing a position during a short price spike can also be beneficial to a user if he primarily wants his target price to be reached and is less interested in earning fees. For now the team will include a warning about flash loans in the protocol description and in the user interface. Implementing flash loan protection using time weighted average prices as a user selectable option is being considered for a future release.

Minor Issues

3. Non-Compliant ERC20 Tokens Not Supported

There are ERC20 tokens that do not strictly adhere to the ERC20 standard and return no value on success (and revert on failure), see https://soliditydeveloper.com/safe-erc20 for a discussion of the issue. Calling `mintNewPosition` with such a token will fail, as no return value after the `transferFrom` can be decoded (in case the transfer is successful). However, those tokens are supported by Uniswap, meaning that the creation unnecessarily fails and limits the applicability of LimitRanger.

Recommendation
As you are already using the TransferHelper library in other places, it would make sense to also use `TransferHelper.safeTransferFrom` (which supports these tokens) in `mintNewPosition`.

4. User Can Avoid Paying Fees In Certain Circumstances

In mintNewPosition, no upper bound on `params.protocolFee` is enforced. This can be misused by providing a very high protocol fee, such that the fee calculation within `_payoutPosition` (`fee0 = (collectedAmount0 / 1000) * position.fee`) overflows. When the fee is chosen intelligently, it will overflow to 0 (or a low value). However, because `position.fee` is a `uint16`, the applicability of this attack is somewhat limited. The maximum value is 65535, meaning that the amount can only be multiplied by ~65.5. Therefore, the amount would need to be a large value such that it overflows when multiplied by 65.5
Moreover, protocol fees that are larger than 1000 are also problematic when they are provided accidentally by the user (and do not cause an overflow). In such a case, paying out the position will not be possible, as `collectedAmount - fee` within `_payoutToken` will fail, causing the transfer to revert.

Recommendation
Set an upper limit for the `protocolFee`.

Informational Notes

5. Opening Positions With Zero Liquidity Possible

mintNewPosition can be called with `params.token0Amount = 0` and `params.token1Amount = 0`, which will open a position with zero liquidity on Uniswap. While this is not a security vulnerability, you might want to check that at least one amount is larger than 0 to avoid that the user accidentally creates such positions.

Recommendation
Consider checking that at least one amount is larger than 0.

6. Solidity Version

Instead of using Solidity version 0.7.6 in combination with `SafeMath`, version 0.8.x could be used, where `SafeMath` is no longer necessary.

Recommendation
Consider switching the Solidity version.

7. WETH Handling

In contrast to a withdrawal, a user has no choice when minting a new position if WETH or ETH should be used. This may be undesirable for certain use cases (e.g., when LimitRanger is used from another smart contract) where the funds already exist as WETH and therefore need to be withdrawn first (meaning that everyone who integrates with LimitRanger needs some special logic for handling ETH / WETH).
Note that there is no requirement by Uniswap that the ETH needs to be transferred for WETH. In the function `pay` within `PeripheryPayments.sol` (https://github.com/Uniswap/v3-periphery/blob/1d69caf0d6c8cfeae9acd1f34ead30018d6e6400/contracts/base/PeripheryPayments.sol#L58), where the payment happens, the first option (when the ETH balance is sufficient) is to use ETH directly, but it can also pull in WETH when the ETH balance is not sufficient.

Recommendation
Consider allowing minting positions with WETH.