# Unbound Finance (UniswapV2) Ethereum Contracts Security Audit

June 14th, 2021 (last updated July 5th, 2021)

@lucash-dev

https://hackerone.com/lucash-dev

https://github.com/lucash-dev

## Disclaimer

Security audits are inherently limited -- there is no possibility of strict proofs that a given system doesn't contain further vulnerabilities. The best that can be offered is a good faith effort to find as many vulnerabilities as possible within the resource constraints of the project. Likewise, validation of fixes for issues found can't be guaranteed to be correct.

Given the above, this audit report is provided "AS IS", without any guarantee of correctness or completeness. The author takes no responsibility for any loss or damage caused by the use, misuse or failure to use the information contained in this document, as well as for any information that might be missing from it (e.g. missed vulnerabilities).

By using the information contained in this report, you agree to do so at your own risk, and not to hold the authors liable for any consequences of such use.

## Summary

This document contains the results of the audit conducted on the Unbound Ethereum smart contracts.

- **Methodology**: a general description of the methodology used to find possible issues.
- **Exploitable Vulnerabilities**: a list of immediately exploitable issues found.
- **Trust Issues**: a quick note on the trust model for the contracts.
- **Attack Scenarios**: a brief description of the main attack scenarios considered during this audit, even though no means of performing the attack were found.
- **Common Weaknesses**: a brief description of the common weaknesses that the code was searched for, though they weren't found in the audited code.
- **Improvement Suggestions**: a list of issues that, if corrected, might lead to increased security, though there is no evidence they lead to any vulnerability as they were found.

## Scope

The scope of this audit are the Ethereum smart contracts found at https://github.com/unbound-finance/UnboundContracts/tree/adf8b2fdbc9d8f7dd5d9386c52769b9f01738878.

The audit is focused on identifying implementation defects that can lead to security vulnerabilities, and does not constitute an appraisal of the code's value proposition.

## Methodology

The audit was conducted manually by an experienced security researcher, by inspecting the code in two different ways:

- Systematic review of every function in the contracts, spotting missing best practices and logical flaws.
- Free exploration of interaction points and execution flows, searching for concrete attack vectors and invalid assumptions.

While the first approach mimics a more traditional code review, the second one tries to reproduce the kind of coverage you would obtain from a "bug bounty" program.

It's the author's belief that combining these approaches offer much higher likelihood of catching high-severity issues than using either individually.

## Exploitable Vulnerabilities Found

One *critical* vulnerability was found in the `LiquidityLockContract` contract.

### Excess collateral returned in `unlockLPT` in the undercollateralization scenario

The `unlockLPT` code deals with three different scenarios:

- Complete unlocking by repaying the entire loan.
- Partial unlocking, when the amount of LPT is more than necessary for collateralization.
- Partial unlocking, when the amount of LPT locked is less than necessary for collateralization

In the third scenario, the code is supposed to return only the excess collateral after the loan is partially repaid, so as to keep the loan fully collateralized.

A mathematical error, however, makes the value returned be wrong, returning almost the entire value locked regardless of the collateralization level or amount of unbound dollares repaid. This allows for users to keep both their LPT tokens and their unbound dollars, as they can repay the minimum unit of unbound dollars and obtain back all of their LPT short of a rounding error.

The root cause is that two numbers in a subtraction operation are represented at different scales. The variable `valueStart` holds a value scaled by `base`, i.e. 1e18. see https://github.com/unbound-finance/UnboundContracts/blob/adf8b2fdbc9d8f7dd5d9386c52769b9f01738878/contracts/LiquidityLockContracts/LiquidityLockContract.sol#L25

Compare to https://github.com/unbound-finance/UnboundContracts/blob/adf8b2fdbc9d8f7dd5d9386c52769b9f01738878/contracts/LiquidityLockContracts/LiquidityLockContract.sol#L18

Even though the scale is corrected in the return statement (by dividing by `valueOfSingleLPT`) the subtraction by `valueAfter` is incorrect, as that value isn't scaled by `base`. See https://github.com/unbound-finance/UnboundContracts/blob/adf8b2fdbc9d8f7dd5d9386c52769b9f01738878/contracts/LiquidityLockContracts/LiquidityLockContract.sol#L25

The result is that `valueStart` is always orders of magnitude larger than `valueAfter`, which leads to a calculation of an amount of tokens to return extremely close to the total LPT locked, regardless of the remaining debt.

This results in an inflation bug as the LPT can be unlocked and locked again without burning the corresponding unbound dollars -- causing excess supply and price depreciation.

A unit test demonstrating this issue can be shared with developers upon request.

*Suggested Remedy* the issue should be corrected by multiplying `valueAfter` by `base` before the subtraction.

**Update by Development Team**: this issue was corrected in the way suggested above. As far as the author can see, the fix is correct and eliminates the issue found.

## Trust Issues

For completeness, the author notes that the good functioning of the contracts partially depends, at least in their present form, on centralized trusted third-parties with privileged access, such as governance and oracle updaters.

This seems to be in accordance to the design and project documentation, which states the intent to migrate to a more distributed trust model in the future, such as a DAO.

No undocumented trust reliance was found by the author.

## Attack Scenarios

This section describes the most important attack scenarios (non-exhaustive list) considered during the audit of the code. This would be direct attacks on the smart contracts, not on economic value of the token.

- **Unauthorized user can perform operations on positions.**

  A malicious user, without proper authorization, is able to perform any operation on a position created by another user, e.g. moving user funds.

- **Oracle Manipulation** A malicious user can change the state of the Oracle (e.g. Uniswap pair manipulation in a flashloan) allowing for performing actions with amounts in excess of what is expected. In particular, such a scenario would allow the attacker to obtain loans with values in excess of provided collateral.

- **Preventing processing of legitimate operations (DoS)**

  A malicious user can change the state of the contracts in a way to prevent the processing of legitimate operations by users, such as obtaining loans or unlocking collateral.

- **Arbitrary Transfer of External Funds**

  A malicious user can make the contracts perform transfers of funds on behalf of users who previously approved them. In particular this could mean an attacker using the vulnerable contract to steal or freeze user balance not locked in the vulnerable contract.

## Common Weaknesses

This section describes some of the common weaknesses (non-exhaustive list) the code was searched for during the audit. *None of these weaknesses* were found in the code.

- **Incorrect Method Access Level**

  This weakness is found when a method in a contract was assigned the wrong access level (private/public/external) allowing an attacker to perform operations that shouldn't have been authorized.

- **Lack of Sender Validation for Restricted Functionality**

  This weakness is found when methods that should be restricted fail to validate `msg.sender` and thus allow any user to access it's functionality.

- **Lack of Callee Validation**

  This weakness is found when methods make contract calls to addresses provided by users in a trusted manner, without validating that address points to a non-malicious contract, or that the call data perform authorized operations.

- **Contract Reentrancy**

  This weakness is found when external/public methods that allow calls to user-specified methods are vulnerable to calling other methods in the original smart contract, leading to possible inconsistency in contract state that might be exploitable. While there are contracts in the code base that could be affected by reentrancy, no way to turn those into exploitable attacks was found.

- **Improper Validation of Values Provided by Sender**

  This weakness happens when parameters in general to an external/public method are trusted to be correct without any validation and are used for determining the state or behavior of the contract.

- **Rounding Errors**

  This weakness happens when division operations aren't consistently rounded, leading to inconsistent contract state that might be exploitable.

- **Overflow/Underflow Errors**

  This weakness happen when the contracts implements mathematical calculations in such a way as to cause the results of operations to exceed the domain of the variable (example, negative values in unsigned integers) thus leading to unexpected or inconsistent state.

- **Trust of External State**

  This weakness happen when the contract code assumes the state of external contracts can't be controlled by the attacker, and use that state as input for it's own functioning. A good example of this would trusting the token balance of an address -- which can be manipulated by sending tokens to that address. Oracle manipulation is a specific, particularly dangerous case of this.

## Improvement Suggestions

This section contains suggestions for improvement of the contracts that might make them more robust against security issues not found in this audit, or that could be introduced in the future. This includes issues with code maintainability, documentation, comments, etc.

### Code duplication between `LockLPT` and `LockLPTWithPermit`

Both functions share most of the logic, except for the call to the permit function. The common logic could be refactored in a reusable internal function, making the code more maintainable.

### Confusing variable names

Certain variable names don't properly describe the value held in it, which makes the code harder to reason about and maintain.

Exmaples:

- `rateBalance` in `valuing_01.sol`: this variable is a base/factor rather than a balance.
- `listOfLLC` in `valuing_01.sol`: this variable is a map, rather than a list.
- `checkLoan` in `UnboundDollar.sol`: this function returns a loan amount, rather than checking for loan existence or state.

# Conclusion

This document described the methodology for conducting the audit of the code, the potential attacks and weaknesses considered.

One critical issue was found, and some potential issues and possible improvements were discussed.

**The critical issue found was addressed by the development team.**