LiquiFi Security Recheck

by Pessimistic

This report is public.

Published: December 2, 2020

Abstract

In this report, we consider the security of the smart contracts of LiqiuiFi project. Our task is to find and describe security issues in smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of the smart contracts of LiqiuiFi project. We performed our audit according to the procedure described below.

The initial analysis showed a critical issue that allows a user to vote several times with same tokens; two issues of medium severity: an ERC20 standard violation and hardly securable quorum; also, there are lots of low-severity issues that arise from complex design of the project and Solidity limitations. Besides the project complexity, the overall code quality is average.

After the audit, the code base was updated. In the latest version of the code, developers fixed most important issues (repeatable voting and hardly securable quorum issues) and several issues of low severity. For the rest of issues, the comments were provided. These issues do not endanger project security.

General recommendations

We recommend refactoring the code, simplifying its logic and splitting the codebase into smaller modules.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether code logic corresponds to the specification.
2. Whether the code is secure.
3. Whether the code meets best practices.

We perform our audit according to the following procedure:

- Automated analysis
  - We scan project's code base with automated tools: Crytic, MythX, and SmartCheck.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We inspect the specification and check whether the logic of smart contracts is consistent with it.
  - We manually analyze code base for security vulnerabilities.
  - We assess overall project structure and quality.
- Report
  - We reflect all the gathered information in the report.

Project overview

Project description

For the analysis, we were provided with the code base on private GitHub repository of LiqiuiFi project, commit 82224a05aa8a9fe598939b48b5c6391768b4acf8.

The project has the documentation as a separate file, whitepaper-liquifi.pdf, sha1sum 7a43fa2ee86c5dc1684360d82911f1e6db81418d.

The total LOC of audited sources is 2534.

Latest version of the code

After the initial audit, the code base was updated. For the recheck we were provided with commit d28c5ac6a24c3ca2672f2e84c16a7edbe750e685.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

Repeatable voting (fixed)

Users can use their tokens to vote multiple times.

To perform an attack, a user needs to repeat the following steps for desirable amount of times:

1. Cast a vote
2. Transfer tokens to a new address

We recommend inspecting existing voting systems and choosing a suitable solution.

> *Comment from developers: the new LiquifiInitialGovernor contract now locks LQF tokens of a voting user until the voting ends.*

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in current implementation. We highly recommend addressing them.

ERC20 standard violation

`transfer()` and `transferFrom()` functions of **LiquifiToken** contract violate ERC20 MUST requirement:

```
Transfers of 0 values MUST be treated as normal transfers and fire the
Transfer event.
```

There are also `SHOULD` recommendations ignored in several cases (see code quality section).

> *Comment from developers: though the issue is a MUST requirement in ERC20 standard, we see no consequences that could under some conditions cause loosing funds or other critical problems. We leave this issue open with the following mitigation plan:*
> *If (when) the standard violation leads to third party integration problems or unacceptance of Liquifi tokens to some exchanges, or earlier upon Liquifi governance decision, we will launch LQF tokens migration program to a new version. The new LQF tokens (LQFv2 tokens) will act as wrappers to the existing LQF tokens and will be freely exchangeable at 1:1 ratio. In addition, both versions of tokens will be accepted for Liquifi governance.*

Hardly securable quorum (fixed)

50% is an extremely high value for a quorum. Consider using a lower value.

> *Comment from developers: we have changed the initial quorum threshold to 10%. Future versions of governance will allow changing this threshold.*

The issue has been fixed and is not present in the latest version of the code.

Low severity issues

Low severity issues can influence project operation in future versions of code. We recommend taking them into account.

Project design

- In **LiquifiPoolFactory** and **LiquifiPoolRegister** contracts, we recommend enforcing strict order of tokens that are passed as arguments. Consider implementing require check to ensure that tokens are properly sorted. This will simplify the logic of the code.

> *Comment from developers: this is a tradeoff of flexibility vs simplicity. We think that simplification of code in this case is not significant enough and therefore will not change this.*

- We recommend moving all the WETH/ETH logic from **LiquifiLiquidityPool** and **LiquifiDelayedExchangePool** contracts to **LiquifiPoolRegister** contract, so that **LiquifiLiquidityPool** and **LiquifiDelayedExchangePool** contracts will not contain any ether logic and will work only with tokens.
- **LiquifiDelayedExchangePool** and **Liquifi** contracts delay writing to storage to save gas, which complicates code logic. However, since Constantinople hardfork, multiple writing to the same storage slot became significantly cheaper. As a result, gas savings are insignificant compared to complexity drawbacks. We recommend removing “dirty storage” optimization.
- **LiquifiDelayedExchangePool** contract uses sorted linked lists for orders, however their logic is mixed with other parts of the code. Consider moving linked list logic implementation into a library.
- In **LiquifiDelayedExchangePool** contract, developers try to reduce stack depth by localizing variables, params ordering, etc. However, this significantly complicates code logic and impairs code readability. Though, it is a serious limitation of Solidity compiler, splitting the code into smaller modules with low coupling usually helps to avoid `stack too deep` issue.
- **LiquifiDelayedExchangePool** contract pretty much implements `God object` anti-pattern. Consider splitting it into smaller modules, e.g., moving detached logic into libraries with internal functions.

> *Comment from developers: true but requires major refactoring that may cause smart contract size or gas consumption issues. We will consider this when working on Liquifi protocol V2.*

Gas consumption

- In **LiquifiPoolRegister** contract, `depositWithETH()` function converts all the available ether to WETH even if some of it is then converted back.

> *Comment from developers: it is not obvious whether changing this logic will lead to lower gas consumption, as it will cause additional calculations and checks. We will consider this when working on Liquifi protocol V2.*

- In **LiquifiPoolFactory** contract, `getPool()` function deploys a very large pool contract. Creating a single contract with all the logic and deploying lightweight proxies that use its logic, will use significantly less gas for deployment.

> *Comment from developers: it is not obvious whether changing this logic will lead to lower gas consumption, as using proxies will cause additional calculations and checks. On the other hand, pool creation is a relatively rare operation and gas economy is more important at usual swap operations. Anyway, we will consider this when working on Liquifi protocol V2.*

Code quality

- Consider using expression like `type(uint64).max` to get the maximum value of a particular `uint` type.

> *Comment from developers: PARTIALLY FIXED (in **LiquifiInitialGovernor**, **LiquifiProposal** contracts). We will also consider this when working on Liquifi protocol V2.*

- In `_require()` function of **Liquifi** contract, consider specifying constants as strings to improve readability:

```
bytes memory message = bytes("FAIL https://err.liquifi.org/__");
```

> *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

- `SHOULD` recommendations of ERC20 standard are ignored in:

  `transfer()` function of **LiquifiToken**:

```
The function SHOULD throw if the message caller's account balance
does not have enough tokens to spend.
```

  LiquifiMinter and **LiquifiLiquidityPool** contracts:

```
A token contract which creates new tokens SHOULD trigger a Transfer
event with the _from address set to 0x0 when tokens are created.
```

> *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

- **LiquifiMinter** and **LiquifiLiquidityPool** contracts are susceptible to ERC20 approve issue.

> *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

- In **LiquifiProposal** contract, consider using `enum` instead of `uint8` at line 14.

  The issue has been fixed and is not present in the latest version of the code.

- In **LiquifiInitialGovernor** contract, `proposalFinalization()` function receives `0` value for `_option` in case if the proposal fails for any reason. However, in case of successful proposal, the function also receives `0` for `_option`. We recommend using another value (e.g., 1) for successful proposals.

  The issue has been fixed and is not present in the latest version of the code.

- In `checkIfEnded()` function of **LiquifiProposal** contract, return value is uninitialized.

  The issue has been fixed and is not present in the latest version of the code.

- If `LiquifiDAO.ProposalStatus.IN_PROGRESS` is not the first element of `ProposalStatus` enum, function `checkIfEnded()` of **LiquifiProposal** will work incorrectly.

  The issue has been fixed and is not present in the latest version of the code.

- Consider using `encodeWithSignature()` or providing a selector as `ERC20.transferFrom.selector` as an argument for `abi.encodeWithSelector()` function in
  - **LiquifiPoolRegister** at line 36
  - **LiquifiPoolRegister** at line 44
  - **LiquifiLiquidityPool** at line 79

> *Comment from developers: FIXED in **LiquifiPoolRegister**. For **LiquifiLiquidityPool** we will consider this when working on Liquifi protocol V2.*

- Consider using `""` instead of `new bytes(0)` to improve code readability in
  - **LiquifiPoolRegister** at line 52
  - **LiquifiLiquidityPool** at line 75

> *Comment from developers: code readability improvement in this case is arguable. We have decided not to change this.*

- In **LiquifiLiquidityPool** contract, consider separating initialization logic and workflow. E.g., moving `_totalSupply == 0` logic from `mint()` function to `constructor`.

> *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

Code style

In **LiquifiPoolFactory** contract, the name of `getPool()` function is misleading since it deploys a contract and writes to storage.

Also, Solidity Style Guide is often ignored.

> *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

Project-related issues

- **package-lock.json** file is missing in the repository.

  We recommend adding this file to repository.

  The issue has been fixed and is not present in the latest version of the code.

- The contracts use `pragma solidity = 0.7.0`. However, there were many bug fixes since 0.7.0, so we recommend using later version, e.g. 0.7.4.

  > *Comment from developers: true. We will consider this when working on Liquifi protocol V2.*

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

December 2, 2020