# Audit Report for Azuro Protocol V1

Branch `release-1.5` / Commit `08d3f27852773b24e41be63fe71ea57fcedd6135`

Audited by: jack @6a61636b

The Azuro protocol is a platform that allows a DAO to facilitate users placing bets on events, with the results provided by oracles. For this audit, I've assumed two things:

- Oracles are honest: there is some mechanism outside these contracts incentivising them to report accurate results
- The `token` that will be specified by the liquidity pool contract does not have any ERC777 style send/receive hooks and is not deflationary

## Contracts/libraries covered

- `AzuroBet.sol`
- `Core.sol`
- `LP.sol`
- `Math.sol`
- `utils/LiquidityTree.sol`
- `interface/IAzuroBet.sol`
- `interface/ICore.sol`
- `interface/ILP.sol`

## Vulnerability Classification

Vulnerabilities have been analyzed based **severity** and **difficulty**: how much damage they could do and how easy they are to exploit.

## Summary of Findings

Two vulnerabilities were found, detailed breakdowns can be found on the following pages. Both have been patched in branch `release-1.6`.

1. In some cases liquidity providers (LPs) can withdraw their liquidity when it should be locked to payout potential winning bets. This results in oracles being unable to resolve a condition: **medium** severity, **low** difficulty.
2. If an oracle resolves a condition right at the timestamp, there is a small chance their update is included in an uncled block, and user could place a risk-free bet in a block with a slightly lower timestamp than the uncled one: **medium** severity, **very high** difficulty.

## *Vulnerability 1*

In the following situation an LP is able to withdraw more liquidity than they should, which also results in the oracle being unable to resolve the condition:

```
// User A deposits liquidity
lp.connect(userA).addLiquidity(100_000);

// Oracle creates condition with reinforcement of 20_000
core.connect(oracle).createCondition(…);

// Locked liquidity is now 20_000 and the liquidity tree has 100_000
lp.lockedLiquidity == 20_000;

// User B deposits liquidity >= the locked liquidity/reinforcement level
lp.connect(userB).addLiquidity(20_000);

// Locked liquidity is now 20_000 and the liquidity tree has 120_000

// User C places a bet
lp.connect(userC).bet(...);

// User A sees that resolving the condition will cause them a loss and pulls
// their liquidity
lp.connect(userA).withdrawLiquidity(100_000);

// We only check withdrawal amount >= liquidity tree total – locked liquidity
// in LP.sol, so this succeeds as 100_000 >= 120_000 - 20_000

// Oracle now tries to resolve a condition that would cause a loss for LPs
core.connect(oracle).resolveCondition(…);

// This fails. When LP.addReserve() from Core.sol is called with a loss,
// LP.sol tries to deduct from the liquidity tree leaves that deposited before
// the condition. In this case that is one leaf which is at zero so we get
// an underflow.
```

I rated this as medium severity since this state can be recovered from by canceling the condition. This reduces the locked liquidity, so other LPs can still withdraw. However, the original LP would still have avoided the loss, and the bet winners would have their wins canceled out. It’s fairly easy to execute for an LP: just wait for a situation that is beneficial to them and withdraw their liquidity.

To fix, I think there are two solutions:

1. Add a `lockedAmount` variable to the `Node` struct in `LiquidityTree.sol`. Update it for all required leaves lazily in the `lockReserve()` function in `LP.sol`, and check against it in `withdrawLiquidity()`.
2. Actually remove the amounts from the required leaves in `lockReserve()` of `LP.sol` using `removeLimit()`, then add them back again in `addReserve()` using `addLimit()` once the condition is resolved.

Implemented fix: `removeLimit()` has been changed out for `remove()`, which removes liquidity from the whole tree rather than specific leaves. LPs could still potentially withdraw early to avoid a loss from a condition (would shift the loss to later LPs). The reinforcement level per condition will be set so that each resolved outcome is a low enough percentage of the overall liquidity pool that this is not worth it for LPs: they would be giving up claims on several conditions that will potentially be resolved for a profit to avoid one small loss.

## *Vulnerability 2*

I want to start by prefacing: this would be very hard to execute, but the mitigation is simple so I thought it would be worth mentioning.

If an oracle resolves a condition at the exact first timestamp allowed, if that block was uncled, and a user could find a block with a lower timestamp (or get a miner to manipulate the timestamp), they could place a risk-free bet up to the max allowed. The block after would then be the one where the timestamp would be valid to resolve the condition.

Uncled blocks/small fork attacks are rare, but we have seen a few in the wild:

https://twitter.com/sbetamc/status/1263220679937265671
https://medium.com/alchemy-api/unmasking-the-ethereum-uncle-bandit-a2b3eb694019

To mitigate, introduce a buffer between the end of betting and the start of resolving the condition. A value of one minute would stop any realistic exploit scenario.

Implemented fix: one minute delay added between end of betting and when condition can be resolved.