# Azuro V3

Protocol Audit Report

Prepared by: Maxim Timofeev (@max_timo)

10.03.25-25.03.25

# Table of Contents



# Protocol Summary

Azuro is a decentralized betting protocol. Anyone can launch a frontend service that connects to the smart contracts and receive an affiliate bonus for each bet made through the given frontend. Different betting events can be hosted, for example a football game. Odds are provided by a Data Feed provider (Oracle) for every bet. A user bet gets automatically converted to an NFT in the user's wallet. Different types of bets can be made, e.g. ordinary, combo bets.

New version of the protocol combines prematch and live betting, ordinary and combo bets into one LiveCore contract. The odds are no more calculated on the contract, but are decided for each bet by an oracle. There are also new options like gasless transactions via Relayer, cashing out the bet via Cashout, bet rejection by privileged role.

# Disclaimer

Auditor makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the auditor is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

# Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
| --- | --- | --- | --- |

| Severity | Impact: High | Impact: Medium | Impact: Low |
| --- | --- | --- | --- |
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

### Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - leads to a moderate material loss of assets in the protocol or moderately harms a group of users or the protocol functioning.
- Low - leads to a minor material loss of assets in the protocol or harms a small group of users or slightly harms the protocol functioning.

### Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

## Audit Details

Review commit hash: 1edde89cf9cebc6ee62cdf75aabe38e3463ac79e

### Scope

```
contracts\extensions\*.sol
contracts\interface\*.sol
contracts\libraries\*.sol
contracts\utils\*.sol
contracts\AzuroBet.sol
contracts\Factory.sol
contracts\LiveCore.sol
contracts\LP.sol
contracts\Vault.sol
```

## Executive Summary

### Issues found

| Severity | Number |
| --- | --- |
| Critical | 4 |
| High | 4 |
| Medium | 5 |
| Low | 1 |
| Informational | 2 |
| Gas | 3 |

# Findings

# Critical

## [C-1] LiveCore.putOrder can be called bypassing LP

### Severity

**Impact:** High, it lets anyone to make bets without funding `amount` of token, and to get full payout after, practically stealing Vault funds.

**Likelihood:** High, anyone can perform an attack.

### Description

`LiveCore.putOrder` function does not have a restriction for anyone to make a bet and not pay any funds as an `amount`. Existing modifier `restricted(order.oracle, this.putOrder.selector)` ensures only that the oracle from `order` structure has access for `putOrder` function, but no rule for `msg.sender`. The `_checkSignatures` function is also not checking sender, but only addresses in order structure.

The consequence is that a bettor can make a bet for free, and after resolving condition withdraw real token payout via `LP.withdrawPayout`. This leads to stealing the protocol funds and potentionally draining the protocol.

### Recommendation

Restrict `msg.sender` of `LiveCore.putOrder` to be only `LP`.

### Status

Fixed

## [C-2] Lack of signatures data consistency verification leads to betting with unrestricted parameters

## Severity

**Impact:** High, it lets anyone to make bets with arbitrary odds on any condition, letting to game the protocol to win a lot of money with unfairly high chance.

**Likelihood:** High, anyone can perform an attack.

## Description

`_checkSignatures` in `LP.betOrder` flow doesn't check that hashes in `data` array are equal to hashes of `order` structure, which brings the opportunity to provide arbitrary values in `order`, except nonce, oracle and bettor signature. The signatures may be any of the correct ones, corresponded to needed bettor and oracle, and signed earlier.

The impact is not only that signatures may be replayed, but the most crucial is that `odds` parameter can be manipulated to any value. An attacker can get correct signatures for the bet with supposedly very high chance of winning (being low odds), change `odds` parameter to the highest allowed value by condition payout limit, change nonce if needed, and put the order in `LP.betOrder`. The result will be an arbitrarily high payout with a winning chance tending to 100%, and repeated many times attack can drain protocol funds.

## Recommendation

`_checkSignatures` should check that `getHashes(order) == data`, where `getHashes` calculates correct message hashes according to EIP712 standard. Another approach may be restricting `LP.betOrder` to be called only by `Relayer.betFor`, which sends correct message hashes to LP, removing the need for checking equality inside.

## Status

Fixed

# [C-3] `LP.migrateDeposits` allows for anyone to mint free deposits

## Severity

**Impact:** High, it lets anyone to mint deposits for free, spending LP funds, and withdraw them after, stealing the funds.

**Likelihood:** High, anyone can perform an attack.

## Description

`LP.migrateDeposits` allows to provide arbitrary `otherLp` address, which opens an attack vector. Consider scenario:

1. Attacker creates a contract, which has function `transferFrom` doing nothing, and `withdrawLiquidity` doing nothing except returning some `amount`.

2. Attacker calls `migrateDeposits` with a contract as `otherLp` and an array with a single value as `oldDepositIds`.
3. `_addDeposit` inside calls `vault.addDeposit`, which takes funds from LP contract and creates new deposit for `account`, which is `msg.sender` (attacker), and with `amount` value.

The result is creating deposit for an attacker, but with LP funds. Attacker then can withdraw the deposit, which means theft of LP funds. The attack may be repeated and LP may get drained.

### Recommendation

Restrict `otherLp` to be only pre-confirmed LP.

### Status

Fixed

## [C-4] `LP.withdrawLiquidityFor` allows for anyone to steal other users' funds

### Severity

**Impact:** High, it lets anyone to withdraw someone else's deposit, stealing his funds.

**Likelihood:** High, anyone can perform an attack.

### Description

`LP.withdrawLiquidityFor` doesn't have a check for `msg.sender`. `isActive(core)` doesn't involve sender aswell. Vault allows for LP to do any operations with deposits via `vault.withdrawLiquidityFor`. An attacker can provide himself in `to` parameter, and another's `depositId`, and the deposit will be withdrawn to attacker address, stealing the funds.

### Recommendation

The function is not used anywhere, thus it can be simply deleted. Another way is to check `msg.sender` to be deposit's owner.

### Status

Fixed

# High

## [H-1] User can make cashout for rejected bet

### Severity

**Impact:** High, the attack can be repeated to game the protocol in a high amount of value.

**Likelihood:** Medium, the odds for a cashout do not always exceed 1, thus not always bringing profit, though the possibility is high.

## Description

If a bet is rejected, there is still an opportunity to make cashout for it because in `_processCashOut`, the call to `IBetting(item.bettingContract).viewPayout(item.betId)` will revert since the condition is not yet finalized. As a result, the `catch` block will be triggered, and the function execution will continue:

```
        try IBetting(item.bettingContract).viewPayout(item.betId) {
            revert BetAlreadyResolved();
        } catch (bytes memory error) {
            if (bytes4(error) == ILiveCore.AlreadyPaid.selector) {
                revert BetAlreadyPaid();
            }
        }
```

Then, the following lines will execute:

```
        uint256 amount = Core(item.bettingContract).bets(item.betId).amount;
        payout = amount.mul(odds);

        IERC721(betToken).transferFrom(betOwner, address(this), item.betId);
```

Thus, the payout will be withdrawn from the `Cashout` contract and sent to the user. If the odds provided by the oracle for the cashout are greater than 1, the user will make a profit, whereas they were only supposed to get the `amount` back due to the rejection.

This can be exploited for an attack. `LiveCore` supports live betting, during which situations like "post-goal" betting can arise. This occurs when, after an event such as a goal in a match, the outcome odds of a condition change drastically, and a bet can be placed before the oracle updates the odds accordingly.

The bet rejection mechanism is used in this case — to remove bets placed immediately after a goal.

The attack could involve placing a "post-goal" bet, taking advantage of the sudden shift in odds and prior knowledge of their direction. Then, when the oracle rejects the bet, the attacker can cash it out at the most favorable odds provided by the oracle.

## Recommendation

Disable cashout for rejected bets, for example by using the `isRejected` flag, or by using the `bet.payout == bet.amount` comparison, since this condition is only met for rejected bets.

## Status

Fixed

# [H-2] Possibility to make cashout for combo bet with losing subbets

## Severity

**Impact:** High, the attack can be repeated to game the protocol in a high amount of value.

**Likelihood:** Medium, all conditions need to be resolved in a correct order, and cashout odds are not always profitable, though the possibility is high.

## Description

In the `viewPayout` function, within the section of code handling combo bets, there are the following lines:

```
                if (_isConditionCanceled(condition)) continue;
                if (!_isConditionResolved(condition))
                    revert OneOfConditionsInComboNotFinished(conditionId);
                if (bet.timestamp >= condition.settledAt) continue;
                // odds > FixedMath.ONE is not rejected subBet
                if (
                    comboPart.odds > FixedMath.ONE &&
                    !isOutcomeWinning(
                        comboPart.conditionId,
                        comboPart.outcomeId
                    )
                ) return 0;

                comboOdds = comboOdds.mul(comboPart.odds);
```

Based on this logic, if a combo bet contains both a lost sub-bet and a sub-bet with an unresolved condition, and during the loop, the unresolved sub-bet is encountered before the lost sub-bet, the `viewPayout` function will revert. However, if the lost sub-bet is encountered first, the function will return 0.

Since a cashout for a bet can be paid out if `viewPayout` reverts, this behavior can be exploited to manipulate the protocol. If a combo bet contains a lost sub-bet that appears later in the list than a sub-bet with an unresolved condition, it will be possible to execute a cashout for the combo bet. This means that if any subbet of a combo bet is losing, then a bettor knows in advance that a combo bet is guaranteed to lose, but they can still cash it out and receive a payout, making the process risk-free and solely dependent on the cashout odds.

By structuring a combo bet strategically, one can arrange the sub-bets so that the conditions with the earliest resolution appear at the end of the list. This ensures that `viewPayout` does not return 0 for as long as possible in case of a loss, allowing for a cashout to be executed. Using multiple such combo bets, an attacker can exploit the protocol for significant gains.

## Recommendation

Modify `viewPayout` so that if any sub-bet in a combo bet is lost, the algorithm always returns 0, preventing the possibility of a revert.

## Status

Fixed

# [H-3] Unnecessary liquidity may be locked for ordinary bets, potentionally leading to funds frozen in LP

## Severity

**Impact:** High, exceeding of locked liquidity may lead to funds forever frozen in LP after a condition resolve. The repeated situation may lead to a very high loss of funds.

**Likelihood:** Medium, redundant liquidity may be locked quite often, by about half of a chance or by a malicious bettor making a big bet in the beginning of condition.

## Description

In a `_changeFunds` function there is an update of a locked liquidity for a condition:

```
        int128 newReserve = _calcReserve(
            payouts,
            totalNetBets,
            winningOutcomesCount
        ).toInt128();
        if (newReserve > oldReserve) {
            condition.maxReserved = newReserve;
            lp.changeLockedLiquidity(newReserve - oldReserve);
        }
```

The liquidity is updated only if `newReserve > oldReserve`, so it is supposed to be only increasing. In an opposite case, when it is smaller, it will not be updated, and the redundantly locked liquidity will remain locked for the condition. If the condition gets resolved right after that, LP will withdraw more liquidity from the vault than needed. That will mean, that after withdrawing all payouts, there will still be an excess of funds left on the LP. Since they are not supposed to be used anywhere else and can't be withdrawn by any other way, they will stuck forever frozen in LP contract.

This case may happen quite often, and the excess may accumulate massively after time. There is also possibility for a malicious bettor to intentionally create the discrepance, for example, making a very big bet in the beginning of a condition lifetime, demanding a very high liquidity for payout. The condition funds will later be "equalized" by the other users' bets, and needed liquidity will most probably get lower than one saved by the first bet, so the chance of unlocking excessive liquidity will be higher than usual.

## Recommendation

Remove the `newReserve > oldReserve` check to let liquidity be updated unconditionally.

## Status

Fixed

# [H-4] Bettor can spend other user's approve via Relayer

## Severity

**Impact:** High, it allows to make arbitrary bets for any user who has approved some funds to the `Relayer`, potentionally losing his funds, and enabling DoS possibility.

**Likelihood:** Medium, attacker needs to sandwich between approve and bet by a user, which is not always possible (e.g. it is made in a single transaction).

## Description

`Relayer.betFor` allows delegating the bet creation transaction to another account so that the `order.bettor` can sign and send a transaction that places a bet on behalf of `betOwner` and deducts funds from them:

```
            // bet owner (bettor or its wallet) send bet amount and fee for the
bet
            TransferHelper.safeTransferFrom(
                token,
                betOwner,
                address(this),
                amount + bettorsFee
            );

            // prepare data for signature check and bet
            lp.betOrder(
                clientData.core,
                order,
                order.betOwner,
                getHashes(order)
            );
```

The issue is that the signature is validated against `order.bettor`, not `betOwner`, while the funds are deducted from `betOwner`. This means that `bettor` can sign a transaction for any arbitrary bet in the name of `betOwner` and send it to the Relayer contract. If `betOwner` has previously approved funds for transfer to the Relayer contract, the funds will be deducted from `betOwner` when the transaction is executed by `bettor`, even though `betOwner` may not have explicitly authorized this bet.

In general, if any account has approved funds for the Relayer, any malicious bettor can sign a transaction and place any bet on behalf of that account. This creates a risk of exposing players' funds by signing them up for losing bets. Additionally, it enables a denial-of-service (DoS) attack by front-running bets and overriding them with malicious transactions that consume approved funds, preventing the original bet from being executed. This can disrupt the ability to place desired bets altogether.

### Recommendation

Remove the ability for an external bettor to sign a signature for placing a bet on behalf of another bettor. All bets should be signed only by `betOwner`, whose funds are being deducted. However, external accounts should still be able to send signed signatures to the contract, ensuring the implementation of gasless transactions as originally intended.

### Status

Fixed

# Medium

## [M-1] Combo bet may contain subbets with the same `conditionId`

### Severity

**Impact:** High, attack allows to multiply possible payout massively, leaving the same chance to win.

**Likelihood:** Low, the oracle is most likely to reject such bet, still situation is possible in case of an error or a third party frontend.

### Description

A combo bet resulting odds and corresponding payout are calculated by multiplying odds from all of the subbets. If a combo bet consists of, for example, 2 subbets on the same condition (and the same outcome, otherwise it's meaningless), its odds will be squared (and the payout), but the chance to win will remain the same as for the single bet on this condition. This opens a possibility to win a lot more money but with the same winning chance.

The situation is possible, because there is no check during creation of a combo bet in a `LiveCore`, that it doesn't contain subbets with the same `conditionId`. However, an oracle is the entity to provide odds and sign the final transaction, and `azuro-sdk` implementation works the way that it checks against subbets condition similarity. Still, the attack could be possible in case of an error and for a third party frontend and oracle, which can forget to add this check.

### Recommendation

Make a check in `_putComboBet` that `data.comboParts` doesn't contain `subBet`-s with the same `conditionId`.

### Status

Fixed

# [M-2] Flawed payout limit check for combo bets allows uncontrollable liquidity locking

## Severity

**Impact:** Medium, some conditions may get flooded by combo bets, leaving no free liquidity to combo bet on other conditions, harming user experience.

**Likelihood:** Medium, combo bets don't need a lot of money to get DOSed, or a situation may happen spontaneously in a moment of a hype for some conditions.

## Description

In the `_putComboBet` function, there is a line that limits the maximum payout for a combo bet:

```
        // check COMBO payout limit
        if (payout > conditionData.payoutLimit) revert PayoutLimit();
```

This code restricts the payout for a combo bet, but multiple bets can be placed, having their payout just below the maximum limit. The system does not account for the total volume of already placed bets or the liquidity spent on each conditions. As a result, the protocol allows all available liquidity to be locked into combo bets on a few specific conditions, leaving no free liquidity for combo on other conditions. This significantly reduces liquidity efficiency and negatively impacts the user experience.

## Recommendation

When limiting the payout for combo bets, consider the liquidity already allocated to each condition by combo bets, similar to how the `_checkPayoutLimit` function works for ordinary bets.

## Status

Acknowledged, the situation is OK.

# [M-3] Combo bets may lock liquidity intended for ordinary bets

## Severity

**Impact:** Medium, combo bets can flood the pool, leaving no liquidity for ordinary bets, though ordinary bets limits were not yet exceeded.

**Likelihood:** Medium, combo bets don't need a lot of money to get DOSed, or a situation may happen spontaneously in a moment of a hype for some conditions.

## Description

When a combo bet is created, a certain amount of liquidity is reserved in the pool for its payout:

```
        bet.payout = payout;
        bet.lastDepositId = lp.getLastDepositId();
        lp.changeLockedLiquidity((payout - data.amount).toInt128());
```

The combo bet mechanism is integrated with the ordinary bet mechanism, and both are funded from the same pool. As a result, the maximum amount of liquidity locked for all combo bets is limited only by the total funds in the vault. This can lead to a situation where users place so many combo bets that no free liquidity remains for ordinary bets, even though the limits for ordinary bets themselves have not yet been reached. In this case, users will temporarily not be able to make ordinary bets, until the liquidity is unlocked. This behavior negatively impacts user experience and reduces the protocol's liquidity efficiency.

## Recommendation

Separate liquidity for single and combo bets so that each type has its own liquidity usage limit.

## Status

Fixed

# [M-4] `LP.withdrawPayout` may revert in case of `token` having blacklist functionality, leading to liquidity locked forever

## Severity

**Impact:** High, in case of a bettor blocked for receiving token transfers after making a combo bet, some amount of liquidity, which is unlocking in `resolvePayout`, will never be unlocked, leading to eternal freezing of the protocol funds.

**Likelihood:** Low, since no "blacklist for receive" tokens (like USDC or TUSD) are used in the protocol currently. Though, they may be utilized in the future or by a third party frontend, deploying and configuring its own set of contracts with `Factory`.

## Description

Inside the `resolvePayout` function, which is called within `withdrawPayout`, there are lines that release the liquidity locked for a specific combo bet:

```
        // if bet is combo - change reserve
        if (bet.comboParts.length > 0) {
            uint128 initialPayout = bet.payout;
            lp.addReserve(
                initialPayout - bet.amount,
                initialPayout - actualPayout,
```

```
                bet.lastDepositId
            );
        }
```

Then, within the same transaction, the bettor receives their payout:

```
        (account, amount) = IBetting(core).resolvePayout(tokenId);
        if (amount > 0) _withdraw(account, amount);
```

If, for any reason, the transfer of funds to the account reverts (e.g., due to an account block in the token contract), the entire transaction will revert, preventing the release of the combo bet's liquidity through `addReserve`. There will be no alternative way to unlock this liquidity, meaning the funds will remain frozen and effectively lost.

If the token has a blocking mechanism, specifically one that prevents receiving funds (such as USDC), the following attack scenario becomes possible:

1. The bettor places as many combo bets as possible with the minimum amount but the highest possible odds to lock as much liquidity as possible.
2. The bettor triggers a block for his account in the token contract.
3. It becomes impossible to execute `withdrawPayout` for this bettor and their combo bets, causing the liquidity they locked to remain permanently frozen.

Any user could exploit this vulnerability, potentially leading to significant losses for the protocol. Since payout for a combo bet is significantly higher than spent `amount`, the attack is economically viable.

In the current version of the protocol, tokens with a receive-blocking mechanism are not used. However, since third-party developers integrating the protocol may choose to use such tokens, this vulnerability could still emerge and put the protocol at risk.

## Recommendation

Allow `resolvePayout` to be called separately from `withdrawPayout` so that liquidity can be unlocked without executing a token transfer. Alternatively, the documentation should explicitly state that tokens with account-blocking mechanisms for receiving transfers should not be used.

## Status

Acknowledged, the protocol is not using "blacklist on receive" tokens.

# [M-5] Incorrect locked liquidity accounting after ordinary bet rejection

## Severity

**Impact:** Medium, insufficient amount of funds will be locked for payouts, thus some bettors may never withdraw their payout, or withdraw it only with delay or a donation involvement.

**Likelihood:** Medium, it needs privileged role to reject a bet before condition resolve, which is supposed to happen moderately often but not always.

### Description

In a `_rejectConditionBets` function, there is a problem with calculating `newReserve` parameter, which is used to update locked liquidity value for a condition.

There is this code:

```
        // recalculate locked reserves
        int128 newReserve = _calcReserve(
            payouts,
            condition.totalNetBets,
            condition.winningOutcomesCount
        ).toInt128();
        lp.changeLockedLiquidity(newReserve - condition.maxReserved);
        condition.maxReserved = newReserve;
```

`payouts` is an array containing all summed payouts from rejected bets, but doesn't contain all existing payouts of the condition. Being that `payouts` values are much less than `condition.totalNetBets`, `newReserve` will most likely become 0. Hence, `changeLockedLiquidity` will unlock all liquidity reserved earlier, and `maxReserved` will become 0.

This leads to a situation when a condition may get resolved right after the bet rejection and may have not enough liquidity to pay bettors their payouts.

### Recommendation

Change `_calcReserve` parameter `payouts` to `condition.payouts`.

### Status

Fixed

## Low

## [L-1] Condition with `winningOutcomesCount` being 0 can be created

### Severity

**Impact:** Low, it allows to create a broken condition, which will not have winners.

**Likelihood:** Low, situation may happen in case of an oracle error, but can be mitigated by condition canceling.

### Description

`LiveCore._createCondition` function allows to create condition which will have `winningOutcomesCount` equal to 0. Such condition will be able to accept bets and to be resolved, but only with empty `winningOutcomes_` array. In this case all users' bets will be losing. It is also possible to cancel the condition and return the bets. Overall, the scenario seems to be strange and can lead to confusion and misuse.

### Recommendation

Check that `winningOutcomesCount == 0` before updating state variable of condition during creation.

### Status

Fixed

## Informational

## [I-1] `ConditionState.PAUSED` is unused

### Description

Option `PAUSED` in `ConditionState` enum is not used anywhere in the protocol and is a redundant code.

### Recommendation

The option can be safely deleted from the enum.

### Status

Fixed

## [I-2] Array lengths may be different

### Description

`data.comboParts` and `order.conditionDatas` arrays in `_putComboBet` may have different lengths, which is not checked. It doesn't have consequences for now, but may be mitigated to avoid further possible problems.

### Recommendation

Check the equality of these arrays' lengths.

### Status

Acknowledged

# Gas

# [G-1] Storage array length multiple read

## Description

Some storage arrays lengths are read in a cycle, bringing redundant gas spending:

- `bet.comboParts.length` in `LiveCore.rejectComboBets`
- `subBets.length` in `LiveCore._resetSubBetOdds`

## Recommendation

Extract the lengths into variables before cycle, and read the variable value.

## Status

Fixed

# [G-2] `treeNode[rChild].amount` storage multiple read

## Description

In `LiquidityTree._getPushView` there is a double storage read:

```
uint256 rAmount = treeNode[rChild].amount;
uint256 sumAmounts = lAmount + treeNode[rChild].amount;
```

## Recommendation

Replace `treeNode[rChild].amount` by `rAmount` in the second line.

## Status

Fixed

# [G-3] `LiquidityTree` binary tree forming can be optimized

## Description

Binary deposit tree currently has a fixed depth of 40 levels. This means that on any new deposit creation or withdrawal there happens at least 40 storage writings, which update all tree nodes from the deposit leaf to

the root. The total number of possible deposits is 2^40, which was reserved for the least possible chance of overflow, but seems to not be needed for a meaningful amount of time, so the most nodes of the tree remain unused. Currently in the protocol there are ~10k deposits, which corresponds to ~0.0000009% of deposit tree used. The possibility of optimizing the tree and reducing storage writings is desirable to be considered.

## Recommendation

The suggested optimization is to have a minimal tree size from the beginning and to double it gradually, when it's needed.

For example, with 5 deposits it's needed to have an 8-leaf tree, which has depth 3 (+1 root). It will spend only 4 storage writings on deposit and withdraw.

For N deposits it's needed to have a tree with `[log2(N)] + 1` levels and `2^[log2(N)]` leaves, where `[x]` means the rounded up integer part of x. For example, the tree with 10000 deposits will have a depth of 15 and leaf number of 16384, which means 15 storage writing for deposit and withdraw, and it is ~3 times lower than current gas spending. So, the gas economy is expected to be significant with such optimization.

The easiest way to implement the approach is to have initial `root` value equal to LIQUIDITYNODES. It will correspond to tree with 1 leaf - the root. All the other nodes should be considered as "out of the tree".

The next step, when the number of deposits exceeds 1, is to double the tree, dividing the root `root /= 2`, so it opens the tree with 2 leaves - [LIQUIDITYNODES, LIQUIDITYNODES + 1]. The higher numbers of leaves should be considered invalid aswell. Existing operations for updating the tree will simply dive into a tree until the `root`, and update `root` when a number of new deposits is exceeded. No other change to the algorithm is needed.

Denote LIQUIDITYNODES as L. So, on the step N, the binary tree in an array should look like this:

```
| L/2^N    | 0 | ... | 0 | L/2^(N-1) | L/2^(N-1) + 1 | 0 | ... | 0 |
| -------- | --------- | ------------------------ | ---------- | ...
|(root #1) |  (zeroes)   | (node #2)   (node #3)     |  (zeroes)   |

     | L/2^(N-k)   | L/2^(N-k) + 1 | ...    | L/2^(N-k) + 2^k-1 |
 ... | ------------------------------------------------------- | ...
     | (node #2^k)  (node #2^k+1)   (nodes)   (node #2^k+2^k-1) |

     | 0 | ... | 0 | L | L+1 | ... | L+2^N-1 |
 ... | --------- | ---------------------- |
     |  (zeroes)   |    (leaves)             |
```

Lower node numbers are "virtual" and only here for clarity about tree arrangement. The real array indexes are upper values.

## Status

Acknowledged