hexens × azuro

MAR.25

SECURITY REVIEW REPORT FOR AZURO

CONTENTS

- An Ordinary Bet Should Not Be Rejected More Than Once
- Incorrect Description of the Vault._deposit() Function
- The Math.maxSum() function should consider the case where n is zero
- Combo bets should be verified to ensure they are not duplicates when created
- Bets with odds less than ONE should not be allowed
- Unused Variable bettor in Relayer.betFor() Function
- Incorrect condition for skipping child nodes in LiquidityTree::_isNeedUpdateWholeLeaves()

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit evaluates the Azuro protocol, a decentralized prediction market protocol that provides tooling and infrastructure for EVM chains to support robust applications and betting interfaces.

Our security review spanned two weeks and included a comprehensive analysis of Azuro's third version.

During the audit, we discovered five critical-severity vulnerabilities that could lead to financial losses for both liquidity providers and players. Additionally, we identified three high-severity, three medium-severity, two low-severity, and four informational issues.

All reported vulnerabilities were either resolved or acknowledged by the development team and subsequently verified by us.

As a result, we can confidently affirm that the protocol's security and overall code quality have improved following our audit.

SCOPE

The analyzed resources are located on:

https://github.com/Azuro-protocol/Azuro-v3/tree/6bbe97a11774cd392da15506e5f860b930883bff

The issues described in this report were fixed in the following commits:

https://github.com/Azuro-protocol/Azuro-v3/commit/30ff8c6ad769049047adf7b3aab5cddcd90ac203

https://github.com/Azuro-protocol/Azuro-v3/commit/ab0521971c5fe5018480e49e509b42ab600bb0ac

AUDITING DETAILS

STARTED
10.03.2025

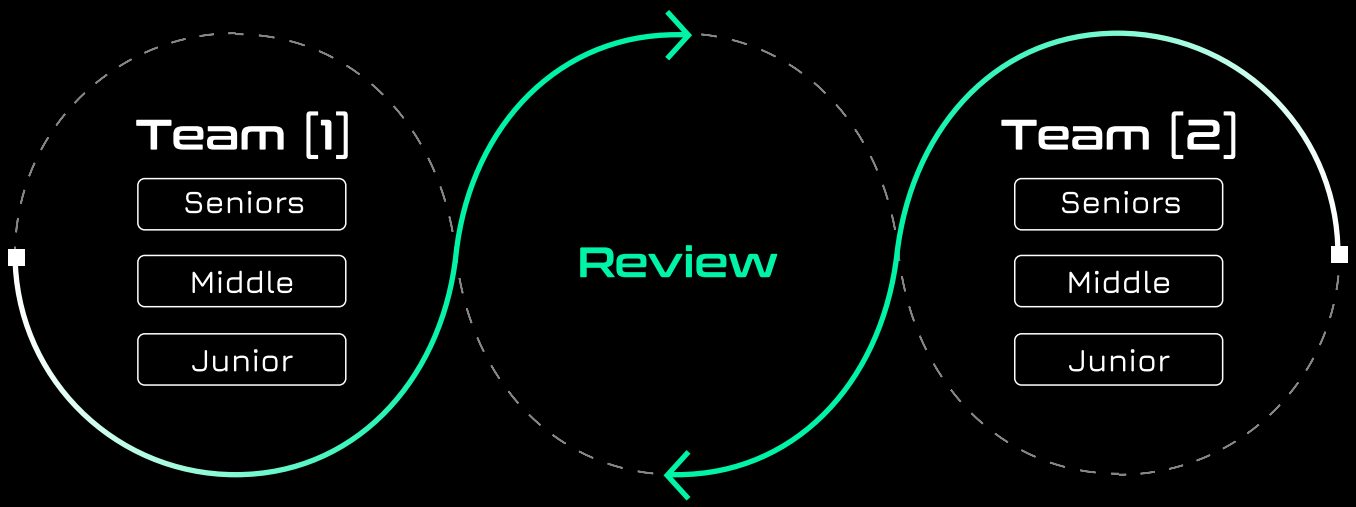
DELIVERED
24.03.2025

Review Led by

TRUNG DINH
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

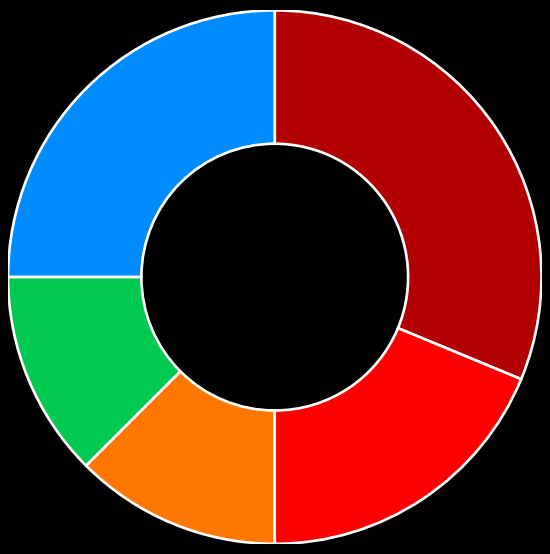
ISSUE SYMBOLIC CODES

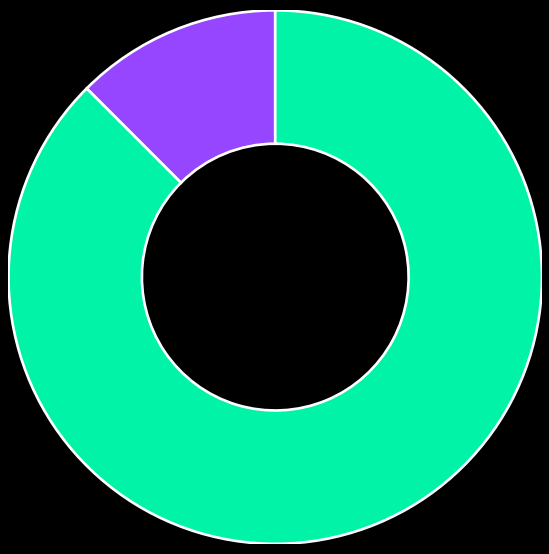
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	5
High	3
Medium	2
Low	2
Informational	4

Total: 16





WEAKNESSES

This section contains the list of discovered weaknesses.

AZUR01-3

INCORRECT NEWRESERVE CALCULATION WHEN REJECTING AN ORDINARY BET

SEVERITY: Critical

PATH:

contracts/LiveCore.sol#L423-L430

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **LiveCore._calcReserve()** function is responsible for calculating the amount of liquidity that needs to be reserved. It takes payouts[] as input, an array that stores the total payout for each outcome index of all placed "Ordinary" bets.

Within the **LiveCore._rejectConditionBets()** function, **_calcReserve()** is called with the **payouts[]** array, which is introduced at line 397 and calculated within the loop at line 409. However, this **payouts[]** array is used to store the total payout amount for each outcome of the rejected bets. In other words, it represents the amount that should be deducted from the storage array **condition.payouts[]**, rather than the expected input format for **LiveCore._calcReserve()** that we described earlier.

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

Consider using **condition.payouts** instead of payouts when calling the function **_calcReserve()** in the function **_rejectConditionBets()**

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

AZURO1-5

INCORRECT NEWPAYOUT CALCULATION IN REJECTCOMBOBETS() DUE TO IMPROPER SETTLEDAT CHECK

SEVERITY: Critical

PATH:

contracts/LiveCore.sol#L140

REMEDIATION:

Consider modifying line 140 of the contract LiveCore as follows:

- if (bet.timestamp >= condition.settledAt) continue;
+ if (condition.settledAt != 0 && bet.timestamp >= condition.settledAt) continue;

STATUS: Fixed

DESCRIPTION:

The **LiveCore.rejectComboBets()** function is responsible for rejecting sub-bets within a list of combo bets. For each combo bet, after resetting the rejected sub-bet to "ONE," the function iterates using the **k** iterator to recalculate the bet's **comboOdds**, which is then used to determine the new payout as **newPayout = comboOdds * amount**.

```
function rejectComboBets(
    RejectedComboBet[] calldata bets_
) external restricted(msg.sender, this.rejectComboBets.selector) {
    // --snip--

    for (uint256 i; i < bets_.length; ++i) {
        // --snip--

        // Recalculate new payout after sub-bet changes
        for (uint256 k; k < bet.comboParts.length; ++k) {
            ComboPart storage comboPart = bet.comboParts[k];
            condition = _getCondition(comboPart.conditionId);

            if (_isConditionCanceled(condition)) continue;
            if (bet.timestamp >= condition.settledAt) continue;

            comboOdds = comboOdds.mul(comboPart.odds);
        }
        newPayout = comboOdds.mul(bet.amount).toUint128();
        lp.changeLockedLiquidity(
            newPayout.toInt128() - bet.payout.toInt128()
        );
        bet.payout = newPayout;
        emit BetRejected(bets_[i].tokenId);
    }
}
```

Within each iteration, two conditions are checked to determine whether a sub-bet's **odds** should be skipped:

- **Line 139**: If the condition has been canceled.
- **Line 140**: If the condition has been resolved and the bet was created after the settlement time (**bet.timestamp >= condition.settledAt**).

However, simply checking whether **bet.timestamp >= condition.settledAt** is insufficient. If the condition has not yet been resolved, **condition.settledAt** remains **0**. As a result, **bet.timestamp** will always be greater than **condition.settledAt**, causing the function to unexpectedly skip the odds of unresolved condition sub-bets. This leads to an incorrect calculation of **comboOdds**, ultimately resulting in an inaccurate **payout**.

```
// recalculate new payout, after sub bet changes
for (uint256 k; k < bet.comboParts.length; ++k) {
    ComboPart storage comboPart = bet.comboParts[k];
    condition = _getCondition(comboPart.conditionId);


    if (_isConditionCanceled(condition)) continue;
    if (bet.timestamp >= condition.settledAt) continue;


    comboOdds = comboOdds.mul(comboPart.odds);
}
newPayout = comboOdds.mul(bet.amount).toUint128();
lp.changeLockedLiquidity(
    newPayout.toInt128() - bet.payout.toInt128()
);
bet.payout = newPayout;
```

Proof of Concept:
Place the following test inside **test > live-test.js** under **describe("Live test") > context("Combo (express) bets")**.

```
it.only("issue AZURO1-5", async() => {
  /// --hexens: reject the second bet
  await makeRejectComboBets(core, maintainer, [
    {
      tokenId: res.bets[0].tokenId,
      subBets: [
        {
          conditionId: condId2,
          outcomeId: OUTCOMEWIN,
        },
      ],
    },
  ]);

  /// --hexens: locked liquidity = 0, even the combo bet still contain 1 sub
bet
  expect(await vault.lockedLiquidity()).to.be.eq(0);
  console.log(await vault.lockedLiquidity());

  /// --hexens: resolve the first and second conditions
  time = await shiftTime(ethers, FIVE_MINUTES);
  await makeClientLiveResolveConditions(core, oracle, conditionId,
[OUTCOMEWIN], time, []);
  await makeClientLiveResolveConditions(core, oracle, condId2, [OUTCOMEWIN],
time, []);

  /// --hexens: underflow because of lockedLiquidity = 0
  await expect(
    lp.withdrawPayout(coreAddress, res.bets[0].tokenId)
  ).to.be.revertedWithPanic(0x11);
});
```

AZUR01-6

LACK OF ACCESS CONTROL IN THE LP.WITHDRAWLIQUIDITYFOR() FUNCTION

SEVERITY: Critical

PATH:

contracts/LP.sol#L498-L508

REMEDIATION:

Consider implementing access control for the LP.withdrawLiquidityFor() function or removing the function if it is not needed.

STATUS: Fixed

DESCRIPTION:

The **LP.withdrawLiquidityFor()** function calls **Vault.withdrawLiquidityFor()**, which handles withdrawing tokens from the Vault contract and transferring them to the specified recipient. However, **LP.withdrawLiquidityFor()** lacks proper access control, allowing anyone to call it and withdraw all funds from the Vault contract for themselves.

```
/**
 * @notice transfer liquidity by the 'core' for use by 'to'
 */
function withdrawLiquidityFor(
    address core,
    address to,
    uint128 amount,
    uint48 depositId
) external isActive(core) {
    vault.withdrawLiquidityFor(to, amount, depositId);
}
```

Proof of Concept:

Place the following test inside **test > lp-test.js** under **describe("LP test")**.

```
it.only("issue AUZURO1-6", async () => {
    await changeReinforcementAbility(lp, liveCore, poolOwner, MULTIPLIER);

    const deposit = 1000n;
    await lp.connect(poolOwner).changeMinDepo(deposit);

    const betAmount = 100n; // max bet with `deposit` payoutLimit
    const lpNFT = await addDeposit(lp, lpSupplier, deposit);

    const attackerAccount = USER_B;
    const balanceBefore = await wxDAI.balanceOf(attackerAccount.address);
    await lp.connect(attackerAccount).withdrawLiquidityFor(
      liveCore.target,
      attackerAccount.address,
      betAmount,
      lpNFT
    );
    const balanceAfter = await wxDAI.balanceOf(attackerAccount.address);
    expect(balanceAfter - balanceBefore).to.equal(betAmount);
    console.log(balanceAfter - balanceBefore);
  });
```

REJECTED ORDINARY BET FAILS TO RETURN PAYOUT TO USER IF THE OUTCOME IS A LOSS

SEVERITY: Critical

PATH:

contracts/LiveCore.sol#L612-L625

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

We analyze this issue in the context of an "Ordinary" bet.

In the **LiveCore.viewPayout()** function, if the condition is resolved and the bet's outcome is a loss, the function immediately returns **0** at line 619:

```
function viewPayout(uint256 tokenId) public view virtual returns (uint128) {
    // --snip--

    // If the bet is a combo
    uint256 length = bet.comboParts.length;
    if (length > 0) {
       // --snip--
    } else {
        uint256 conditionId = bet.conditionId;
        condition = _getCondition(conditionId);
        if (_isConditionResolved(condition)) {
            if (bet.timestamp >= condition.settledAt) return bet.amount;
            if (isOutcomeWinning(bet.conditionId, bet.outcomeId))
                return bet.payout;
            else return 0; //@audit Returns 0 without checking if the bet
was rejected
```

```
        }
        if (_isConditionCanceled(condition)) return bet.amount;

        revert ConditionNotFinished();
    }
}
```

At first glance, this behavior appears correct, as a losing bet should not return any payout to the token owner. However, it fails to account for bets that were rejected beforehand. If a bet was rejected, the owner should receive their original bet amount regardless of the condition's outcome.

As a result, rejected bets do not return funds to the owner, causing the bet amount to become stuck in the LP contract.

Consider returning **bet.payout** if the bet has been rejected in the function **LiveCore.viewPayout()**.

```
function viewPayout(uint256 tokenId) public view virtual returns (uint128) {
    // --snip--

    // If the bet is a combo
    uint256 length = bet.comboParts.length;
    if (length > 0) {
       // --snip--
    } else {
        uint256 conditionId = bet.conditionId;
        condition = _getCondition(conditionId);
        if (_isConditionResolved(condition)) {
            if (bet.timestamp >= condition.settledAt) return bet.amount;
            if (isOutcomeWinning(bet.conditionId, bet.outcomeId))
                return bet.payout;
            else {
++              // if the bet is rejected
++              if (bet.payout == bet.amount)
++                  return bet.payout
                else
                    return 0;
            }
        }
        if (_isConditionCanceled(condition)) return bet.amount;

        revert ConditionNotFinished();
    }
}
```

Proof of concept:

Place the following test inside **test > live-test.js** under **describe("Live test") > context("Single bet")**.

```
it.only("issue AUZURO1-7", async () => {
    await timeShiftBy(ethers, FIVE_MINUTES);
    let nonce = 1n;
    const betOdds = 1500000000000n;
    const payout = (BET_100 * betOdds) / MULTIPLIER;
    betParams.clientBetData.bets[0].odds = betOdds;
    betParams.clientBetData.clientData.expiresAt = (await
getBlockTime(ethers)) + 1000;

    let b1 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);
    let b2 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);

    betParams.clientBetData.bets[0].outcomeId = OUTCOMELOSE;
    let b3 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);
    await timeShiftBy(ethers, FIVE_MINUTES);

    /// -- hexens: b4's outcome loses
    betParams.clientBetData.bets[0].outcomeId = OUTCOMELOSE;
    let b4 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);
    let token4 = b4.bets[0].tokenId;

    // -- hexens: reject bet b4
    await makeRejectBets(core, maintainer, [{ conditionId: conditionId,
tokenIds: [token4] }]);

    // resolve
    await makeClientLiveResolveConditions(core, oracle, conditionId,
[OUTCOMEWIN], b4.eventTime + 1, []);

    /// -- hexens: b4 returns 0 payout even it's cancelled
    expect((await makeWithdrawPayout(lp, core, bettor, token4))
[0]).to.be.eq(0);
});
```

AZUR01-16

UNFAIR DISTRIBUTION OF PROFITS AND LOSSES FOR LPS OF THE VAULT WHO DEPOSIT WHEN THE CONDITIONS ARE RUNNING

SEVERITY: Critical

PATH:

contracts/LiveCore.sol#L680

contracts/LiveCore.sol#L728

REMEDIATION:

The liquidity added after the condition's creation shouldn't be reserved and locked in the same way as the liquidity added before. Therefore, you should create a mechanism to handle locking liquidity across a range of LPs for each condition.

STATUS: Acknowledged

DESCRIPTION:

When a condition is created, **condition.lastDepositId** is set to the current last deposit ID of the vault in the **LiveCore::_createCondition()** function.

```
condition.lastDepositId = lp.getLastDepositId();
```

It will be used to determine the range of liquidity IDs in the vault that are associated with that condition. The profit or loss for that condition will be updated to the corresponding segment of the vault's liquidity via **LP::addReserve()**.

```
function _resolveCondition(
    Condition storage condition,
    uint256 conditionId,
    ConditionState result,
    uint128[] memory winningOutcomes_,
    uint128 payout,
    uint64 settledAt
) internal {
    ...

    lp.addReserve(lockedReserve, profitReserve, condition.lastDepositId);

    ...
}
```

The point is, after a condition is created, any liquidity added to the vault will be used as reserve for that condition. Ordinary bets for that condition can have enough liquidity to lock and then place an order. So, the liquidity added after the condition is created is treated the same as the liquidity added before the condition is resolved.

However, if the condition generates profit for the liquidity (the bet is losing), the liquidity added after the condition's creation will not receive any profit. Only the liquidity deposited before the condition's creation will receive this profit. But if the condition results in a loss for the liquidity (the bet is winning), the liquidity added after the condition's creation will also face the risk of losing funds (locked liquidity).

```
function _createCondition(
    ConditionData memory conditionData,
    address oracle,
    Condition storage condition
) internal {
    (uint128 value, bool isNotUnique) = Math.isNotUniq(
        conditionData.outcomes
    );
    if (isNotUnique) revert DuplicateOutcomes(value);

    if (conditionData.winningOutcomesCount >= conditionData.outcomes.length)
        revert IncorrectWinningOutcomesCount();

    condition.lastDepositId = lp.getLastDepositId();
    condition.winningOutcomesCount = conditionData.winningOutcomesCount;
    condition.payouts = new uint128[](conditionData.outcomes.length);
    condition.oracle = oracle;
    for (uint256 i; i < conditionData.outcomes.length; ++i) {
        outcomeNumbers[conditionData.conditionId][
            conditionData.outcomes[i]
        ] = i + 1;
    }

    emit ConditionCreated(
        conditionData.gameId,
        conditionData.conditionId,
        conditionData.outcomes,
        conditionData.odds,
        conditionData.winningOutcomesCount
    );
}
```

```
function _resolveCondition(
    Condition storage condition,
    uint256 conditionId,
    ConditionState result,
    uint128[] memory winningOutcomes_,
    uint128 payout,
    uint64 settledAt
) internal {
    (uint128 value, bool isNotUnique) = Math.isNotUniq(winningOutcomes_);
    if (isNotUnique) revert DuplicateOutcomes(value);

    condition.state = result;
    for (uint256 i; i < winningOutcomes_.length; ++i) {
        winningOutcomes[conditionId][winningOutcomes_[i]] = true;
    }

    uint128 lockedReserve = uint128(condition.maxReserved);
    uint128 profitReserve = lockedReserve + condition.totalNetBets - payout;

    lp.addReserve(lockedReserve, profitReserve, condition.lastDepositId);

    emit ConditionResolved(
        conditionId,
        uint8(result),
        winningOutcomes_,
        profitReserve.toInt128() - lockedReserve.toInt128(),
        settledAt
    );
}
```

Commentary from the client:

“ If the bet is accepted, its payout is guaranteed by the blocked liquidity, and if the payout implies a loss, it will be distributed. Then, if the used deposit range is not enough to cover the loss, the deposit range will be expanded to all. This is an extreme case (not common), but probable."

Proof of Concept:

```
const { expect } = require("chai");
const {
  addDeposit,
  addDepositFor,
  getBlockTime,
  timeShiftBy,
  multipliedPercents,
  tokens,
  prepareEmptyStand,
  createCondition,
  getWithdrawnAmount,
  makeWithdrawPayout,
  changeReinforcementAbility,
  prepareAccessLiveCore,
  makeLiveBet,
  makeClientLiveResolveConditions,
} = require("../utils/utils");
const { MULTIPLIER } = require("../utils/constants");
const { ethers } = require("hardhat");

const LIQUIDITY = tokens(2_000_000);
const ONE_WEEK = 604800;
const ONE_DAY = 86400;
const OUTCOMEWIN = 1n;
const OUTCOMELOSE = 2n;

const WITHDRAW_100_PERCENT = MULTIPLIER;
const WITHDRAW_80_PERCENT = multipliedPercents(80, 100);
const WITHDRAW_50_PERCENT = multipliedPercents(50, 100);
const WITHDRAW_20_PERCENT = multipliedPercents(20, 100);
const WITHDRAW_10_PERCENT = multipliedPercents(10, 100);
const TOKENS_1K = tokens(1_000);
const TOKENS_3K = tokens(3_000);
const TOKENS_4K = tokens(4_000);
const TOKENS_5K = tokens(5_000);
const TOKENS_20K = tokens(20_000);
const TOKENS_100K = tokens(100_000);
const FIRST_DEPO = tokens(100);
const SECOND_DEPO = tokens(100);

const payoutLimit = TOKENS_20K; // 10%
```

```
const minDepo = tokens(10);
const daoFee = multipliedPercents(9, 100); // 9%
const dataProviderFee = multipliedPercents(1, 100); // 1%
const affiliateFee = multipliedPercents(60, 100); // 60%

const approveAmount = tokens(999_999_999_999_999);
const odds = MULTIPLIER * 2n;

const DEPO_A = tokens(120_000);
const DEPO_B = tokens(10_000);
const DEPO_SMALL = tokens(1);

let dao, poolOwner, dataProvider, affiliate, lpSupplier, lpSupplier2,
lpOwner, oracle, oracle2, maintainer;
let access, liveCore, relayer, wxDAI, lp, vault;
let roleIds, lpnft0;

let conditionId = 0n;
let conditions = [];
let lpNFT_A, lpNFT_B;

describe("issue AZURO1-16", function () {
  before(async () => {
    [
      dao,
      poolOwner,
      dataProvider,
      affiliate,
      lpOwner,
      lpSupplier,
      lpSupplier2,
      oracle,
      oracle2,
      maintainer,
      USER_B,
      USER_C,
      USER_D,
    ] = await ethers.getSigners();
  });
  beforeEach(async () => {
    ({ access, liveCore, relayer, wxDAI, lp, vault, roleIds } = await
prepareEmptyStand(
      ethers,
      dao,
```

```
      poolOwner,
      dataProvider,
      affiliate,
      lpSupplier,
      minDepo,
      daoFee,
      dataProviderFee,
      affiliateFee
    ));
    await prepareAccessLiveCore(access, poolOwner, [oracle], [maintainer],
roleIds);

    await wxDAI.connect(poolOwner).approve(relayer, approveAmount);
    await wxDAI.connect(poolOwner).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier2).approve(lp, approveAmount);
  });
  it.only("test poc case 1: lps get loss", async () => {
      //lp1 deposits 1k
      await addDeposit(lp, lpSupplier, TOKENS_1K);
      let lp1 = await vault.getLastDepositId();

      //create a condition and put a bet of 1k
      const condition = createCondition(oracle, ++conditionId, [OUTCOMEWIN,
OUTCOMELOSE], LIQUIDITY);
      const tokenId = await makeLiveBet(liveCore, relayer, poolOwner,
condition, OUTCOMEWIN, odds, TOKENS_1K);

      //lp2 deposits 1k
      await addDeposit(lp, lpSupplier, TOKENS_1K);
      let lp2 = await vault.getLastDepositId();

      //put a new bet of 1k
      const tokenId2 = await makeLiveBet(liveCore, relayer, poolOwner,
condition, OUTCOMEWIN, odds, TOKENS_1K);

      //resolve condition with outcomewin, all bets are winning
      await makeClientLiveResolveConditions(liveCore, oracle, conditionId,
[OUTCOMEWIN], await getBlockTime(ethers));

      //both lp1 and lp2 get loss
      expect(await vault.nodeWithdrawView(lp1)).to.equal(0);
      expect(await vault.nodeWithdrawView(lp2)).to.equal(0);
    });
```

```
    it.only("test poc case 2: lps get profit", async () => {
      //lp1 deposits 1k
      await addDeposit(lp, lpSupplier, TOKENS_1K);
      let lp1 = await vault.getLastDepositId();
      let beforeNodeWithdraw1 = await vault.nodeWithdrawView(lp1);

      //create a condition and put a bet of 1k
      const condition = createCondition(oracle, ++conditionId, [OUTCOMEWIN,
OUTCOMELOSE], LIQUIDITY);
      const tokenId = await makeLiveBet(liveCore, relayer, poolOwner,
condition, OUTCOMELOSE, odds, TOKENS_1K);

      //lp2 deposits 1k
      await addDeposit(lp, lpSupplier, TOKENS_1K);
      let lp2 = await vault.getLastDepositId();
      let beforeNodeWithdraw2 = await vault.nodeWithdrawView(lp1);

      //put a new bet of 1k
      const tokenId2 = await makeLiveBet(liveCore, relayer, poolOwner,
condition, OUTCOMELOSE, odds, TOKENS_1K);

      //resolve condition, all bets are losing
      await makeClientLiveResolveConditions(liveCore, oracle, conditionId,
[OUTCOMEWIN], await getBlockTime(ethers));

      //only lp1 gets profit, lp2 gets nothing
      expect(await
vault.nodeWithdrawView(lp1)).to.be.gt(beforeNodeWithdraw1);
      expect(await
vault.nodeWithdrawView(lp2)).to.equal(beforeNodeWithdraw2);
    });
});
```

Explanation of PoC:

- **Before resolving the condition:**
  - 1st LP deposits 1000 tokens.
  - The condition is created with a bet of 1000 tokens, odd = 2.
    => All 1000 tokens of LP1 are locked.
  - 2nd LP deposits 1000 tokens.
  - A new bet is placed with 1000 tokens, odd = 2.
    => Since 1000 tokens of LP2 are available as reserve in the vault, all LP2's tokens are locked in the same way as LP1's and become part of the reserve for the condition.
- **The first scenario**:
  When the bets are winning, LP2 loses their liquidity, even though it was deposited after the condition was created.
  - The condition is resolved with the result that both bets are winning, causing a loss of 2000 tokens in liquidity.
    => **LP.addReserve()** will trigger **vault.withdrawLiquidityFor()** with a loss of 2000 tokens, and the deposit ID is LP1's deposit ID.
  - However, liquidity from start to LP1 is not enough to cover 2000 tokens (LP1 only has 1000 tokens in liquidity).
    => **LiquidityTree::_isNeedUpdateWholeLeaves** returns true, causing an update to all LPs in the vault.
  - Therefore, the loss from the condition is also distributed to LP2, causing LP2 to lose all 1000 tokens of liquidity.
- **The second scenario**:
  When the bets are losing, LP2 doesn't receive any profit.
  - The condition is resolved with the result that both bets are losing, making a profit of 2000 tokens for liquidity.
    => **LP.addReserve()** will trigger **vault.addLiquidity()** with an amount of 2000 tokens, and the deposit ID is LP1's deposit ID.
  - Only LPs in the range from start to LP1 are distributed rewards through the liquidity tree, so LP2 doesn't receive anything.

ATTACKERS CAN LOCK LIQUIDITY BY PLACING MULTIPLE LONG COMBO BETS WITH MINIMAL STAKE

SEVERITY: High

PATH:

contracts/LiveCore.sol#L218-L290

REMEDIATION:

Consider setting a higher default value for the minimum bet amount.

Additionally, impose a limit on the combo odds (the product of all sub-bet odds) to prevent the exploitation of large payouts with minimal cost.

STATUS: Fixed

DESCRIPTION:

In the **_putComboBet()** function, a combo bet has no limit on its length. The only limitation is the payout for a combo bet, which is governed by the payoutLimit in the last condition of the sub-bets array:

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

However, the minimum bet amount is set to 1 (wei) in the **LP::addCore()** function:

```
function addCore(address core) external override onlyFactory {
    CoreData storage coreData = _getCore(core);
    coreData.minBet = 1;
    coreData.reinforcementAbility = uint64(FixedMath.ONE);
    coreData.state = CoreState.ACTIVE;

    emit CoreSettingsUpdated(
        core,
        CoreState.ACTIVE,
        uint64(FixedMath.ONE),
        1
    );
}
```

An attacker can place a combo bet with only 1 wei as the bet amount and use a very long list of sub-bets to multiply many odds and achieve a large payout. They can accept the loss by using different outcomes of the same condition as sub-bets since the cost is minimal, thereby locking a large amount of liquidity funds when creating a bet.

For example, if the last condition has a payout limit of $25,000 (25,000e18 wei), and there are 2 currently active conditions: the first condition has 10 outcomes, each with odds of 8, and the second condition has 13 outcomes, each with odds of 10.

The attacker can choose all 10 outcomes of the first condition and all 13 outcomes of the second condition to create a combo bet with 23 sub-bets. The payout of this combo bet will be 1 * 8^10 * 10^13 ≈ 10,737e18.

Although the attacker will never win this bet, they can successfully lock more than $10,000 of liquidity reserves with a cost of just 1 wei. They can continue creating combo bets in a similar manner to lock all the liquidity funds.

```
function _putComboBet(
    OrderData memory order,
    address betOwner,
    uint256 minBet
) internal returns (uint256 tokenId) {
    ClientComboBetData memory data = abi.decode(
        order.clientBetData,
        (ClientComboBetData)
    );
    if (data.amount < minBet) revert SmallBet();
    ConditionData memory conditionData;
    tokenId = azuroBet.mint(betOwner);
    uint256 comboOdds = FixedMath.ONE;

    Bet storage bet = bets[tokenId];
    bet.amount = data.amount;
    bet.timestamp = uint64(block.timestamp);

    SubBetData[] memory subBetData = new SubBetData[](
        data.comboParts.length
    );

    for (uint256 i; i < data.comboParts.length; ++i) {
        ComboPart memory subBet = data.comboParts[i];
        conditionData = order.conditionDatas[i];

        // get live core specific data
        if (conditionData.conditionId != subBet.conditionId)
            revert IncorrectConditionIds();

        _safeCreateCondition(
            conditionData.conditionId,
            conditionData,
            order.oracle
        );

        subBetData[i] = SubBetData({
            gameId: conditionData.gameId,
            conditionId: conditionData.conditionId,
            conditionKind: conditionData.conditionKind,
            outcomeId: subBet.outcomeId,
            odds: subBet.odds
        });
```

```
            comboOdds = comboOdds.mul(subBet.odds);
            bet.comboParts.push(subBet);
        }

        // nonce only for the first subBet
        uint256 nonce = data.nonce;
        uint128 payout = comboOdds.mul(data.amount).toUint128();

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

        // it must be unique for the bettor and bet conditions
        _registerNonce(order.bettor, nonce);

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

        emit NewLiveBet(
            tokenId,
            order.bettor,
            data.clientData.affiliate,
            BetType.COMBO,
            nonce,
            data.amount,
            subBetData,
            conditionData.payoutLimit
        );
    }
```

AZUR01-15

LATE ORDINARY BETS ARE NOT PROPERLY HANDLED, LEADING TO INCORRECT PROFIT/LOSS CALCULATION

SEVERITY: High

PATH:

contracts/LiveCore.sol#L709-L737

REMEDIATION:

Consider rejecting all ordinary bets placed after the condition's settlement time.

To ensure this process is handled correctly, we recommend splitting the resolveConditions() function into two separate functions, each to be called independently:

1. The first function should set the condition's settledAt timestamp and update its state to RESOLVED. After this step, no further bets should be allowed for the resolved condition due to the check at line 474 of the function _safeCreateCondition().
2. The backend should then identify and reject all bets placed after the settlement time.
3. Finally, the second function should be called to propagate the corresponding loss or profit to the LP.

STATUS: Fixed

DESCRIPTION:

At line 599 in the **function LiveCore.viewPayout()**, the function returns **bet.amount** if **bet.timestamp >= condition.settledAt**. In other words, the user receives their original bet amount back if the bet was placed **after the condition was resolved**.

This behavior implies that bets **can be placed after a condition is resolved**, which is understandable due to the inherent delay between real-world events and their reflection on the blockchain. As a result, the protocol must properly handle these "**after-game**" bets to ensure accurate liquidity management.

For **Ordinary** bets, the protocol correctly refunds the original bet amount if the bet was placed after the settlement time as we described earlier. However, this alone is not sufficient, because the bet amount is still included in **condition.totalNetBets**, which affects liquidity calculations.

To prevent liquidity from being locked incorrectly, the protocol must also **exclude** the late bet amount from **condition.totalNetBets**. A similar approach should be applied to **condition.payouts[]**, ensuring that the late bet does not contribute to the payout distribution.

Consider the following example:

1. We have a Yes/No condition that asks: "Will Hexens and Azuro form a strong partnership?"
2. At timestamp = 1, Alice places a $2000 bet on "Yes" with **odds = 2.0**, resulting in a payout of:
   - **payout = 2000 * 2.0 = 4000$**
   - Condition storage updates:
     - **totalNetBets = 0 + 2000 = 2000$**
     - **maxReserved = 4000 - 2000 = 2000$**
3. At timestamp = 10, Bob places a $1000 bet on "Yes" with **odds = 4.0**, resulting in a payout of:
   - **payout = 1000 * 4.0 = 4000$**
   - Condition storage updates:
     - **totalNetBets = 2000 + 1000 = 3000$**
     - **maxReserved = (4000 + 4000) - 5000 = 3000$**
4. However, in reality, the condition is already resolved with the outcome "Yes" at timestamp = 5, but the **oracle** submits this result later, at timestamp = 15.
   - When **resolveConditions()** is triggered, the function propagates a loss = (5000 + 3000 - 8000) - 3000 = -3000$ (following the formula in line 726 of **_resolveCondition()**) to the LP.

    - However, the actual loss should be -2000$ because Bob's bet should be ignored:
        - **viewPayout(Bob) = bet.amount = 1000$** (since it was placed late)
        - **viewPayout(Alice) = bet.payout = 2000 * 2.0 = 4000$**
        - Correct loss calculation: **(1000 - 1000) + (4000 - 2000) = -2000$**
- As we can see, if Bob's bet is not excluded from the condition's storage accounting, it leads to unnecessary fund loss for the LP.

```
function _resolveCondition(
    Condition storage condition,
    uint256 conditionId,
    ConditionState result,
    uint128[] memory winningOutcomes_,
    uint128 payout,
    uint64 settledAt
) internal {
    (uint128 value, bool isNotUnique) = Math.isNotUniq(winningOutcomes_);
    if (isNotUnique) revert DuplicateOutcomes(value);

    condition.state = result;
    for (uint256 i; i < winningOutcomes_.length; ++i) {
        winningOutcomes[conditionId][winningOutcomes_[i]] = true;
    }

    uint128 lockedReserve = uint128(condition.maxReserved);
    uint128 profitReserve = lockedReserve + condition.totalNetBets - payout;

    lp.addReserve(lockedReserve, profitReserve, condition.lastDepositId);

    emit ConditionResolved(
        conditionId,
        uint8(result),
        winningOutcomes_,
        profitReserve.toInt128() - lockedReserve.toInt128(),
        settledAt
    );
}
```

AZUR01-19

THE ORDER IN WHICH THE CONDITION IS RESOLVED AFFECTS THE DISTRIBUTION OF PROFIT AND LOSS TO THE LPS OF THE VAULT

SEVERITY: High

PATH:

contracts/LP.sol#L385-L423

REMEDIATION:

The order in which conditions are resolved should be fixed in some way, such as by using the settled time.

STATUS: Acknowledged

DESCRIPTION:

When a condition is resolved, the **LP::addReserve()** function will distribute the profit or loss of the condition to the LPs from the start node to **condition.lastDepositId**.

However, it distributes profit and loss to the LPs based on their current liquidity in the liquidity tree, without considering the reserves of other conditions. Therefore, the order in which conditions are resolved affects the distribution, causing LPs to receive unfair profit or loss.

Consider the scenario:

- There are two LP nodes: LP1 and LP2. Each has 1000 tokens in liquidity.
- There are two conditions created after LP1 and before LP2 (**condition1** and **condition2**), each with one bet of 500 tokens and an odd of 2.
- **condition1** will cause a loss of 500 tokens for the LPs (bet is winning).

- **condition2** will cause a profit of 500 tokens for the LPs (bet is losing).

There is one condition created after LP2 (**condition3**), with one bet of 1000 tokens and an odd of 2.

- **condition3** will cause a loss of 1000 tokens for the LPs (bet is winning).

Case 1: resolve condition1 → condition2 → condition3

- After resolving **condition1**, LP1 = 500, LP2 = 1000 (remove 500 tokens of liquidity from start to LP1).
- After resolving **condition2**, LP1 = 1000, LP2 = 1000 (add 500 tokens of liquidity from start to LP1).
- After resolving **condition3**, LP1 = 500, LP2 = 500 (remove 1000 tokens of liquidity from start to LP2).

Case 2: resolve condition3 → condition1 → condition2

- After resolving **condition3**, LP1 = 500, LP2 = 500 (remove 1000 tokens of liquidity from start to LP2).
- After resolving **condition1**, LP1 = 0, LP2 = 500 (remove 500 tokens of liquidity from start to LP1).
- When resolving **condition2**, it attempts to add 500 tokens of liquidity from start to LP1. However, the liquidity from start to LP1 is currently 0, so **LiquidityTree::_isNeedUpdateWholeLeaves()** returns true.
  => This results in updating the profit to all LPs in the vault, but since LP1 has 0 liquidity, LP2 will receive all the rewards.
- In this case, because profit is updated for all LPs, any new LPs that deposit before resolving **condition2** will also receive profit. Therefore, an attacker could front-run **resolveCondition()** of **condition3** by depositing a large amount of liquidity, effectively claiming most of the profit from that condition.

```
function addReserve(
    uint128 lockedReserve,
    uint128 finalReserve,
    uint48 depositId
) external override isCore(msg.sender) {
    Reward storage daoReward = rewards[factory.owner()];
    Reward storage dataProviderReward = rewards[dataProvider];
    Reward storage affiliateReward = rewards[affiliate];

    if (finalReserve > lockedReserve) {
        uint128 profit = finalReserve - lockedReserve;
        // add profit to liquidity (reduced by dao/data provider/affiliates
rewards)
        profit -= (_chargeReward(daoReward, profit, FeeType.DAO) +
            _chargeReward(
                dataProviderReward,
                profit,
                FeeType.DATA_PROVIDER
            ) +
            _chargeReward(affiliateReward, profit, FeeType.AFFILIATES));

        vault.addLiquidity(profit, depositId);
    } else {
        // remove loss from liquidityTree excluding canceled conditions
(when finalReserve = lockedReserve)
        if (lockedReserve - finalReserve > 0) {
            uint128 loss = lockedReserve - finalReserve;
            // remove all loss (reduced by data dao/data provider/affiliates
losses) from liquidity
            loss -= (_chargeFine(daoReward, loss, FeeType.DAO) +
                _chargeFine(
                    dataProviderReward,
                    loss,
                    FeeType.DATA_PROVIDER
                ) +
                _chargeFine(affiliateReward, loss, FeeType.AFFILIATES));
            vault.withdrawLiquidityFor(address(this), loss, depositId);
        }
    }
    if (lockedReserve > 0)
        _reduceLockedLiquidity(msg.sender, lockedReserve);
}
```

Commentary from the client:

“ The order of resolution is determined by objective circumstances in the game event. When distributing profit/loss in different sequences, it can (this is a kind of terminal case!) lead to the distribution of the loss to **new deposits**."

Proof of Concept:

```
describe("issue AZURO1-19", function () {
  before(async () => {
    [
      dao,
      poolOwner,
      dataProvider,
      affiliate,
      lpOwner,
      lpSupplier,
      lpSupplier2,
      oracle,
      oracle2,
      maintainer,
      USER_B,
      USER_C,
      USER_D,
    ] = await ethers.getSigners();
  });

  beforeEach(async () => {
    ({ access, liveCore, relayer, wxDAI, lp, vault, roleIds } = await
prepareEmptyStand(
      ethers,
      dao,
      poolOwner,
      dataProvider,
      affiliate,
      lpSupplier,
      minDepo,
      daoFee,
      dataProviderFee,
      affiliateFee
    ));
    await prepareAccessLiveCore(access, poolOwner, [oracle], [maintainer],
roleIds);

    await wxDAI.connect(poolOwner).approve(relayer, approveAmount);
    await wxDAI.connect(poolOwner).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier2).approve(lp, approveAmount);
  });
```

```
describe("issue AZURO1-19", function () {
  before(async () => {
    [
      dao,
      poolOwner,
      dataProvider,
      affiliate,
      lpOwner,
      lpSupplier,
      lpSupplier2,
      oracle,
      oracle2,
      maintainer,
      USER_B,
      USER_C,
      USER_D,
    ] = await ethers.getSigners();
  });

  beforeEach(async () => {
    ({ access, liveCore, relayer, wxDAI, lp, vault, roleIds } = await
prepareEmptyStand(
      ethers,
      dao,
      poolOwner,
      dataProvider,
      affiliate,
      lpSupplier,
      minDepo,
      daoFee,
      dataProviderFee,
      affiliateFee
    ));
    await prepareAccessLiveCore(access, poolOwner, [oracle], [maintainer],
roleIds);

    await wxDAI.connect(poolOwner).approve(relayer, approveAmount);
    await wxDAI.connect(poolOwner).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier).approve(lp, approveAmount);
    await wxDAI.connect(lpSupplier2).approve(lp, approveAmount);
  });
```

```
  let conditionId1;
  let conditionId2;
  let conditionId3;
  let lp1;
  let lp2;

  async function setupScenario() {
    conditionId1 = ++conditionId;
    conditionId2 = ++conditionId;
    conditionId3 = ++conditionId;

    await addDeposit(lp, lpSupplier, tokens(1000));
    lp1 = await vault.getLastDepositId();

    const condition1 = createCondition(oracle, conditionId1, [OUTCOMEWIN,
OUTCOMELOSE], LIQUIDITY);
    const tokenId1 = await makeLiveBet(liveCore, relayer, poolOwner,
condition1, OUTCOMEWIN, odds, tokens(500));

    const condition2 = createCondition(oracle, conditionId2, [OUTCOMEWIN,
OUTCOMELOSE], LIQUIDITY);
    const tokenId2 = await makeLiveBet(liveCore, relayer, poolOwner,
condition2, OUTCOMELOSE, odds, tokens(500));

    await addDeposit(lp, lpSupplier, tokens(1000));
    lp2 = await vault.getLastDepositId();

    const condition3 = createCondition(oracle, conditionId3, [OUTCOMEWIN,
OUTCOMELOSE], LIQUIDITY);
    const tokenId3 = await makeLiveBet(liveCore, relayer, poolOwner,
condition3, OUTCOMEWIN, odds, tokens(1000));
  }

  it.only("test case 1: cond1 -> cond2 -> cond3", async () => {
    await setupScenario()

    await makeClientLiveResolveConditions(liveCore, oracle, conditionId1,
[OUTCOMEWIN], await getBlockTime(ethers));
    await makeClientLiveResolveConditions(liveCore, oracle, conditionId2,
[OUTCOMEWIN], await getBlockTime(ethers));
    await makeClientLiveResolveConditions(liveCore, oracle, conditionId3,
[OUTCOMEWIN], await getBlockTime(ethers));

    expect(await vault.nodeWithdrawView(lp1)).to.be.equal(tokens(500));
```

```
    expect(await vault.nodeWithdrawView(lp2)).to.be.equal(tokens(500));
  });

  it.only("test case 2: cond3 -> cond1 -> cond2", async () => {
    await setupScenario()

    await makeClientLiveResolveConditions(liveCore, oracle, conditionId3,
[OUTCOMEWIN], await getBlockTime(ethers));
    await makeClientLiveResolveConditions(liveCore, oracle, conditionId1,
[OUTCOMEWIN], await getBlockTime(ethers));
    await makeClientLiveResolveConditions(liveCore, oracle, conditionId2,
[OUTCOMEWIN], await getBlockTime(ethers));

    expect(await vault.nodeWithdrawView(lp1)).to.be.equal(tokens(0));
    expect(await vault.nodeWithdrawView(lp2)).to.be.equal(tokens(1000));
  });
});
```

AZURO1-1

INCORRECT IMPLEMENTATION OF THE MATH.ISNOTUNIQ() FUNCTION

SEVERITY: Medium

PATH:

contracts/libraries/Math.sol#L12-L23

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **Math.isNotUniq()** function checks whether the given array **a[]** contains any repeated values. It accomplishes this by iterating through the entire array using the iterator **i**. For each element **a[i]**, the function runs a nested loop with iterator **j** to compare **a[i]** against all other elements in the array.

However, the issue occurs in the second loop, where the function incorrectly increments **i** instead of **j**. This causes **j** to remain at **0** after each iteration. As a result, the function only compares **a[i]** with **a[0]** rather than checking all elements properly.

```
function isNotUniq(
    uint128[] memory a
) internal pure returns (uint128 notUniqueValue, bool isNotUnique) {
    uint256 length = a.length;
    for (uint256 i; i < length; ++i) {
        for (uint256 j; i < length; ++i) {
            if (i == j) continue;
            if (a[i] == a[j]) return (a[i], true);
        }
    }
    return (0, false);
}
```

Consider increasing **j** instead of **i** in the second loop as follows:

```
function isNotUniq(
    uint128[] memory a
) internal pure returns (uint128 notUniqueValue, bool isNotUnique) {
    uint256 length = a.length;
    for (uint256 i; i < length; ++i) {
--      for (uint256 j; i < length; ++i) {
++      for (uint256 j; j < length; ++j) {
            if (i == j) continue;
            if (a[i] == a[j]) return (a[i], true);
        }
    }
    return (0, false);
}
```

AZUR01-8

AN ORDINARY BET SHOULD NOT BE REJECTED MORE THAN ONCE

SEVERITY: Medium

PATH:

contracts/LiveCore.sol#L389-L431

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **LiveCore._rejectConditionBets()** function is responsible for rejecting a list of **Ordinary** bets. However, it does not check whether a bet has already been rejected, allowing the same bet to be rejected multiple times.

After rejecting the bets, the function deducts the total bet amount (**amounts**) from the storage variable **condition.totalNetBets** in line 418. If a bet is rejected more than once, its amount is deducted multiple times, leading to an incorrect calculation of **totalNetBets**.

A similar issue also affects the storage array **condition.payouts[]**, where multiple rejections cause repeated deductions, resulting in inaccurate payout calculations.

Since **_rejectConditionBets()** can be triggered by multiple entities—either through **rejectBets()** by holders of the corresponding access NFT or via **resolveConditions()** by the **oracle**—there is a possibility that multiple entities may detect the same bet as needing rejection and call **_rejectConditionBets()** on it multiple times. This scenario can lead to the issue described above, where a bet is rejected more than once, resulting in incorrect deductions from **totalNetBets** and **condition.payouts[]**.

```
function _rejectConditionBets(
    Condition storage condition,
    uint256 conditionId,
    uint256[] memory betTokens
) internal {
    uint256 outcomes = condition.payouts.length;
    uint128 amounts;
    uint128 amount;
    uint128[] memory payouts = new uint128[](outcomes);
    Bet storage bet;
    for (uint256 i; i < betTokens.length; ++i) {
        bet = bets[betTokens[i]];

        // Only for ordinary bet (not combo)!
        if (bet.comboParts.length > 0) revert IncorrectBetType();
        if (bet.conditionId != conditionId) revert IncorrectConditionId();
        if (bet.isPaid) revert AlreadyPaid();

        amount = bet.amount;
        amounts += amount;
        payouts[getOutcomeIndex(bet.conditionId, bet.outcomeId)] += bet
            .payout;
        // reject payout
        bet.payout = amount;

        emit BetRejected(betTokens[i]);
    }

    // remove bets, payouts
    condition.totalNetBets -= amounts;
    for (uint256 i; i < outcomes; ++i) {
        condition.payouts[i] -= payouts[i];
    }

    // recalculate locked reserves
    int128 newReserve = _calcReserve(
        payouts,
        condition.totalNetBets,
```

```
        condition.winningOutcomesCount
    ).toInt128();
    lp.changeLockedLiquidity(newReserve - condition.maxReserved);
    condition.maxReserved = newReserve;
}
```

Consider reverting the transaction if a bet is already rejected.

```
function _rejectConditionBets(
    Condition storage condition,
    uint256 conditionId,
    uint256[] memory betTokens
) internal {
    // --snip--

    for (uint256 i; i < betTokens.length; ++i) {
        bet = bets[betTokens[i]];

        // Only for ordinary bet (not combo)!
        if (bet.comboParts.length > 0) revert IncorrectBetType();
        if (bet.conditionId != conditionId) revert IncorrectConditionId();
        if (bet.isPaid) revert AlreadyPaid();
++      if (bet.payout == bet.amount) revert AlreadyRejected();

    // --snip--
```

Proof of Concept:

Place the following test inside **test > lp-test.js** under **describe("Live test") > context("Single bet")**.

```
it.only("issue AZURO1-8", async () => {
    await timeShiftBy(ethers, FIVE_MINUTES);
    let nonce = 1n;
    const betOdds = 1500000000000n;
    const payout = (BET_100 * betOdds) / MULTIPLIER;
    betParams.clientBetData.bets[0].odds = betOdds;
    betParams.clientBetData.clientData.expiresAt = (await
getBlockTime(ethers)) + 1000;

    let b1 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);
    let b2 = await makeRelayerBetLiveGetTokenId(relayExecutor1, betParams,
nonce++);
    let token1 = b1.bets[0].tokenId;
    let token2 = b2.bets[0].tokenId;

    await timeShiftBy(ethers, FIVE_MINUTES);

    // reject the bet first time
    await makeRejectBets(core, maintainer, [{ conditionId: conditionId,
tokenIds: [token1] }]);
    const oldLockedLiquidity = await vault.lockedLiquidity();

    // reject the bet second time
    await makeRejectBets(core, maintainer, [{ conditionId: conditionId,
tokenIds: [token2] }]);

    expect(oldLockedLiquidity != (await vault.lockedLiquidity()));
});
```

INCORRECT DESCRIPTION OF THE VAULT._DEPOSIT() FUNCTION

SEVERITY: Low

PATH:

contracts/Vault.sol#L180-L187

REMEDIATION:

Consider updating the description as follows:

-- * @notice Deposits `amount` of `token` tokens from `msg.sender` balance to the contract.
++ * @notice Deposits `amount` of `token` tokens from `from` balance to the contract.

STATUS: Fixed

DESCRIPTION:

The **Vault._deposit()** function is responsible for transferring tokens from the specified **from** address to the vault contract. However, the function's description incorrectly states that tokens are deposited from **msg.sender**, rather than from the **from** address.

This discrepancy can cause confusion and reduce the readability of the contract.

```
/**
 * @notice Deposits `amount` of `token` tokens from `msg.sender` balance to
the contract.
 * @param from address to get liquidity from
 * @param amount The amount of tokens to deposit.
 */
function _deposit(address from, uint128 amount) internal {
    TransferHelper.safeTransferFrom(token, from, address(this), amount);
}
```

THE MATH.MAXSUM() FUNCTION SHOULD CONSIDER THE CASE WHERE N IS ZERO

SEVERITY: Low

PATH:

contracts/libraries/Math.sol#L67

REMEDIATION:

If n is zero, Math::maxSum() should return zero to represent the sum of the empty subset.

STATUS: Fixed

DESCRIPTION:

The **Math::maxSum()** function is used to calculate the sum of the **n** largest items in an array **a**. When **n** is less than 2, it will always return the maximum item in **a**, even if **n** is zero.

```
function maxSum(
    uint128[] memory a,
    uint256 n
) internal pure returns (uint256 sum_) {
    if (n < 2) return max(a);
    ...
}
```

The **LiveCore** contract doesn't verify that the **winningOutcomesCount** of a condition is not zero. If the **winningOutcomesCount** is zero, **LiveCore::_calcReserve()** will calculate the reserve based on the maximum payout and use it to lock liquidity. However, it should be zero because this condition won't return any payout.

```
function _calcReserve(
    uint128[] memory payouts,
    uint256 totalNetBets,
    uint8 winningOutcomesCount
) internal pure returns (uint128) {
    return
        Math
            .diffOrZero(
                Math.maxSum(payouts, winningOutcomesCount),
                totalNetBets
            )
            .toUint128();
}
```

```
function maxSum(
    uint128[] memory a,
    uint256 n
) internal pure returns (uint256 sum_) {
    if (n < 2) return max(a);

    uint256 length = a.length;
    if (length <= n) return sum(a);

    uint128[] memory sorted = new uint128[](length);
    for (uint256 i; i < length; ++i) {
        sorted[i] = a[i];
    }
    sort(sorted, 0, length - 1);

    for (uint256 i; i < n; ++i) {
        sum_ += sorted[length - 1 - i];
    }
}
```

COMBO BETS SHOULD BE VERIFIED TO ENSURE THEY ARE NOT DUPLICATES WHEN CREATED

SEVERITY: Informational

PATH:

contracts/LiveCore.sol#L218-L290

REMEDIATION:

The _putComboBet function should verify the bets in the combo parts to ensure there are no duplicate bets with the same condition and outcome.

STATUS: Fixed

DESCRIPTION:

The **LiveCore::_putComboBet** function doesn't have any check to verify that the **comboParts** don't contain duplicate bets with the same condition and outcome. If this happens (due to missing verification from the oracle before signing the bet), a combo bet can result in a huge payout while the winning chance remains the same.

```
function _putComboBet(
    OrderData memory order,
    address betOwner,
    uint256 minBet
) internal returns (uint256 tokenId) {
    ClientComboBetData memory data = abi.decode(
        order.clientBetData,
        (ClientComboBetData)
    );
    if (data.amount < minBet) revert SmallBet();
    ConditionData memory conditionData;
    tokenId = azuroBet.mint(betOwner);
    uint256 comboOdds = FixedMath.ONE;

    Bet storage bet = bets[tokenId];
    bet.amount = data.amount;
    bet.timestamp = uint64(block.timestamp);

    SubBetData[] memory subBetData = new SubBetData[](
        data.comboParts.length
    );

    for (uint256 i; i < data.comboParts.length; ++i) {
        ComboPart memory subBet = data.comboParts[i];
        conditionData = order.conditionDatas[i];

        // get live core specific data
        if (conditionData.conditionId != subBet.conditionId)
            revert IncorrectConditionIds();

        _safeCreateCondition(
            conditionData.conditionId,
            conditionData,
            order.oracle
        );

        subBetData[i] = SubBetData({
            gameId: conditionData.gameId,
            conditionId: conditionData.conditionId,
            conditionKind: conditionData.conditionKind,
            outcomeId: subBet.outcomeId,
            odds: subBet.odds
        });
```

```
            comboOdds = comboOdds.mul(subBet.odds);
            bet.comboParts.push(subBet);
        }

        // nonce only for the first subBet
        uint256 nonce = data.nonce;
        uint128 payout = comboOdds.mul(data.amount).toUint128();

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

        // it must be unique for the bettor and bet conditions
        _registerNonce(order.bettor, nonce);

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

        emit NewLiveBet(
            tokenId,
            order.bettor,
            data.clientData.affiliate,
            BetType.COMBO,
            nonce,
            data.amount,
            subBetData,
            conditionData.payoutLimit
        );
    }
```

AZUR01-10

BETS WITH ODDS LESS THAN ONE SHOULD NOT BE ALLOWED

SEVERITY: Informational

PATH:

contracts/LiveCore.sol::_putOrdinaryBets()

contracts/LiveCore.sol::_putComboBet()

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

When a bet's odds are set to **FixedMath.ONE**, it is considered rejected, as seen in line 455 of the **_resetSubBetOdds()** function.

```
function _resetSubBetOdds(
    ComboSubBet[] calldata inputSubBets,
    ComboPart[] storage subBets
) internal {
    // --snip--

    if (subBet.odds != one64) subBet.odds = one64;

    // --snip--
}
```

However, when creating combo or ordinary bets, the contract does not enforce a requirement for the odds to be greater than **FixedMath.ONE**. As a result, bettors can place bets with no potential profit or loss, which serves no meaningful purpose for either the players or the protocol.

Consider requiring the odds of the created bet to be greater than **FixMath.ONE**.

```
function _putComboBet(
    OrderData memory order,
    address betOwner,
    uint256 minBet
) internal returns (uint256 tokenId) {
    // --snip--
    for (uint256 i; i < data.comboParts.length; ++i) {
        ComboPart memory subBet = data.comboParts[i];
        conditionData = order.conditionDatas[i];

++      if (subBet.odds <= FixedMath.ONE) revert TooSmallOdds();

        // --snip--
}
```

AZUR01-14

UNUSED VARIABLE BETTOR IN RELAYER.BETFOR() FUNCTION

SEVERITY: Informational

PATH:

contracts/extensions/Relayer.sol#L71

REMEDIATION:

Consider removing the variable if not necessary.

STATUS: Fixed

DESCRIPTION:

In the function **Relayer.betFor()**, the variable **bettor** is declared on line 71 but remains unused throughout the function. Its only purpose is to receive the value of **order.bettor** on line 79, without being utilized elsewhere. This results in an unnecessary variable declaration, which could be removed to improve code clarity and efficiency.

```
address bettor;
```

AZUR01-18

INCORRECT CONDITION FOR SKIPPING CHILD NODES IN LIQUIDITYTREE::_ISNEEDUPDATEWHOLELEAVES()

SEVERITY: Informational

PATH:

contracts/utils/LiquidityTree.sol#L535-L538

REMEDIATION:

Since _isNeedUpdateWholeLeaves returns true if the liquidity in the considered range is insufficient for subtraction, the shortcut condition should check the liquidity within that range. Therefore, the above condition can be fixed as follows:

```
if (
    (isSub && lAmount < forLeftAmount
    && (sumAmounts - lAmount < amount - forLeftAmount))
) return true;
```

STATUS: Fixed

DESCRIPTION:

In the **LiquidityTree** contract, **_isNeedUpdateWholeLeaves** is used to determine whether the entire tree needs to be updated when adding or subtracting an amount in the range from l to r. This function will return true when the liquidity in the range from l to r is insufficient to remove in the case of subtraction, and when the liquidity in the range from l to r is zero in the case of addition.

When l <= mid and r > mid, this function has a shortcut to return true in the case of subtraction:

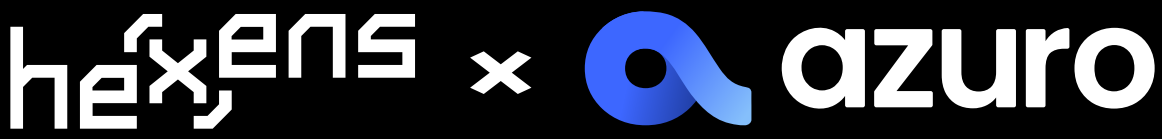
```
uint48 lChild = node * 2;

uint256 lAmount = treeNode[lChild].amount;
uint256 rAmount = treeNode[lChild + 1].amount;
uint256 sumAmounts = lAmount +
    // get right amount excluding unused leaves
    (rAmount -
        _getLeavesAmount(lChild + 1, mid + 1, end, r + 1, end));
if (sumAmounts == 0) return true;
uint128 forLeftAmount = uint128(
    (amount * lAmount).div(sumAmounts) / FixedMath.ONE
);

// if reduced amount is not sufficient for each child - need to update whole
tree
if (
    (isSub && (rAmount < amount - forLeftAmount)) &&
    lAmount < forLeftAmount
) return true;
```

However, this condition will never occur because at the beginning of this function, the case when **treeNode[node].amount < amount** is already considered. **treeNode[node].amount = lAmount + rAmount**, and **treeNode[node].amount** is already greater than **amount**, which bypassed that check.

```
if (
    (isSub && treeNode[node].amount < amount) ||
    (!isSub && treeNode[node].amount == 0)
) return true;
```


hexens
azuro