hexens

AZURO SMART CONTRACT AUDIT REPORT

01.11.2022

CONTENTS

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	2
HIGH	2
MEDIUM	1
LOW	0
INFORMATIONAL	3

TOTAL: 8

SCOPE

The analyzed resources are located on:
https://github.com/Azuro-protocol/Azuro-v2/tree/release-2.2
(commit 83763067c87946d1179cf7962820b1ac36bd9f41)

The issues described in this report were fixed in the following commit:
https://github.com/Azuro-protocol/Azuro-v2/tree/release-2.4

WEAKNESSES

This section contains the list of discovered weaknesses.

1. REENTRANCY

SEVERITY: Critical

PATH: Livecore.sol:129

REMEDIATION: follow the Checks-Effects-Interaction pattern systematically

STATUS: fixed

DESCRIPTION:

In **claimBetToken**, mint comes before the bet is deleted from the bets mapping, opening up to reentrancy, because minting an **ERC1155** calls the destination address. This allows a bettor to recursively call **claimBetToken** over the same bet after it was resolved and mint unlimited **AzuroBet ERC1155** tokens.

```
function claimBetToken(uint256 betId) external {
    if (bets[betId].bettor != msg.sender) revert OnlyBetOwner();

    BetGroup storage bg = betGroups[betId];
    if (_isBetGroupRejected(bg)) revert BetRejected();
...

    azuroBet.mint(
        msg.sender,
        getTokenId(conditionId, outcomeIndex),
        amount,
        uint128(batchOdds[bg.batchId][outcomeIndex].mul(amount))
    );
    delete bets[betId];
}
```

2. REENTRANCY

SEVERITY: Critical

PATH: LP.sol:643

REMEDIATION: follow the Checks-Effects-Interaction pattern systematically

STATUS: fixed

DESCRIPTION:

In **_addLiquidity**, the mint function comes before **withdrawAfter** mapping is updated, allowing a reentrancy attack that ignores withdrawal time limitations.

Since this function can be called by any user via **addLiquidity** it is possible to reenter, perform operations that exploit the added liquidity, possibly inflated by a flashloan, and then reclaim that liquidity before the current timestamp is used to update **withdrawAfter** and inhibit reclaims.

```
function _addLiquidity(uint128 amount) internal {
    if (amount < minDepo) revert AmountNotSufficient();

    uint48 leaf = _nodeAddLiquidity(amount);

    // make NFT
    _mint(msg.sender, leaf);
    withdrawAfter[leaf] = uint64(block.timestamp) + withdrawTimeout;
    emit LiquidityAdded(msg.sender, leaf, amount);
}
```

3. REENTRANCY

SEVERITY: High

PATH: LP.sol:339

REMEDIATION: follow the Checks-Effects-Interactions pattern and make the external call come after the (internal) effects have been processed, meaning after the reward has been zeroed out and claim timestamped

STATUS: fixed

DESCRIPTION:

This **claimReward()** function can be reentered by a malicious smart contract to drain LP's rewards.

The issue would arise when a liquidity pool is initialised with a token that has a transfer hook which calls the destination address (such as **ERC777**). A malicious user can build a smart contract to interact with the protocol, generate some rewards, then claim them multiple times through reentrancy, draining the contract.

```
function claimReward() external {
    Reward storage reward = rewards[msg.sender];
    if (reward.amount <= 0) revert NoReward();
    if ((block.timestamp - reward.claimedAt) < claimTimeout)
        revert ClaimTimeout(reward.claimedAt + claimTimeout);
    TransferHelper.safeTransfer(token, msg.sender, uint128(reward.amount));
    reward.amount = 0;
    reward.claimedAt = uint64(block.timestamp);
  }
```

4. REENTRANCY

SEVERITY: High

PATH: AzuroBet.sol:86

REMEDIATION: follow the Checks-Effects-Interaction pattern systematically

STATUS: fixed

DESCRIPTION:

The call to **_mint** comes before the **_balancePayouts**`mapping is increased by the payout. **ERC1155** mints should be considered like potential threats, since the destination will be called. Following the CEI pattern systematically is a safer pattern, meaning that all calls to **ERC1155** mint should come last in a function, and that function should also only be invoked last.

```
function mint(
      address to,
      uint256 id,
      uint256 amount,
      uint256 payout
  ) external override onlyCore {
      super._mint(to, id, amount, "");
      _balancePayouts[id][to] += payout;
  }
```

5. UNSAFE TYPE CONVERSION

SEVERITY: Medium

PATH: multiple contracts mentioned in the description

REMEDIATION: replace all casts to a smaller type by calls to a function that checks for overflow/truncation first (see description for further details)

STATUS: acknowledged

DESCRIPTION:

The code contains an abundance of type conversions that represent potential risks because of the way **Solidity** handles conversion when the destination type consists of less bits than the source. This truncation might result in unexpected code behaviour, yet some cases may not be able to reach the problematic values, making any blanket mitigation strategy a risk of gas waste. If gas consumption control is a priority, cases should be individually analysed and mitigated. Otherwise a valid mitigation strategy would be to replace all casts to a smaller type by calls to a function that checks for overflow/truncation first.

References:

LP.sol:

- addReserve():535,

- addReserve():536,

- addReserve():553,

- addReserve():554,

- _addDelta():753,

- _reduceDelta():762

CoreBase.sol:

- _resolveCondition():432,

- _resolveCondition():401,

- _calcDeltaReserve():509,

- _calcDeltaReserve():511,

- _calcOdds():543,

- viewPayout():251,

- viewPayout():259,

- _cancel():303,

- _createCondition():358,

- _createCondition():361

LiveCore.sol:

- _executeBatch():401

Core.sol:

- putBet():67

AffiliateHelper.sol:

- updateContribution():91,

- updateContribution():94

CoreTools.sol:

- getFundsRatioLive():71

6. COMMENTS

SEVERITY: Informational

PATH: LP.sol:692

REMEDIATION: add missing natspec and correct the comments

STATUS: acknowledged

DESCRIPTION:

Some comments are ambiguous or unclear, proofreading and adaptation by a native English speaker would be beneficial to future developers inheriting the code.

Several functions also lack full natspec comments, for example a missing description of its parameters.

```
/**
   * @notice Indicate if `core` is an active Core or not.
   */
  function _updateCore(address core, bool active) internal {
      cores[core] = active ? CoreState.ACTIVE : CoreState.INACTIVE;
      emit CoreUpdated(core, active);
  }

  /**
   * @notice Resolve payout for liquidity deposit.
   * @param  depNum deposit token ID
   * @param  percent payout share to resolve where `FixedMath.ONE` is 100%
of deposit payout
   */
```

7. PERMISSIONLESS FACTORY

SEVERITY: Informational

PATH: Factory.sol:58

REMEDIATION: If this is not intended behaviour, it should be addressed to reduce the protocol's attack surface

STATUS: acknowledged

DESCRIPTION:

Anyone can deploy a Liquidity Pool with any token, and arbitrary fees, even if a pool already exists associated with that token.

This seems to be a design decision which fosters healthy competition, but it also has the potential to facilitate scam schemes. If this is not intended behaviour, then it should be addressed to reduce the protocol's attack surface.

```
function createPool(
    address token,
    uint64 daoFee,
...
    string calldata coreType,
    address oracle
  ) external {
...

    emit NewPool(lpAddress);

    _plugCore(lpAddress, beaconCore);
  }
```

8. UNUSED MODIFIER

SEVERITY: Informational

PATH: LP.sol:61

REMEDIATION: remove the modifier or add functionality to it

STATUS: fixed

DESCRIPTION:

The **onlyMaintainer** modifier is defined but never used in the LP contract. The function it calls (**checkMaintainer**) is also called externally from **CoreBase** child contracts, and is crucial to the correct operation of the protocol, but this modifier itself is dead code and might be confusing to developers working on the protocol in the future.

```
/**
   * @notice Only permits calls by Maintainers.
   */
  modifier onlyMaintainer() {
      checkMaintainer(msg.sender);
      _;
  }
```