Azuro Security Review

Pashov Audit Group

Conducted by: ast3ros, dirk_y

February 20th 2024 - February 23rd 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Azuro-protocol/Azuro-v2** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Azuro

The BetExpress contract is a Betting Engine that combines outcomes from different Azuro conditions in a single bet. The BetExpress contract allows users to place combo bets.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 5d36c9cfc98afb1910692df106275cae81750652

fixes review commit hash - a4ec8c7edaceae35c0f44b19eeb58a13d1929c9d

Scope

The following smart contracts were in scope of the audit:

- `BetExpress`

7. Executive Summary

Over the course of the security review, ast3ros, dirk_y engaged with Azuro to review Azuro. In this period of time a total of **4** issues were uncovered.

Protocol Summary

Protocol Name	Azuro
Repository	https://github.com/Azuro-protocol/Azuro-v2
Date	February 20th 2024 - February 23rd 2024
Protocol Type	Betting Engine

Findings Count

Severity	**Amount**
High	1
Medium	2
Low	1
Total Findings	**4**

Summary of Findings

ID	Title	Severity	Status
[H-01]	BetExpress is broken when a bet has a condition with more than one outcome	High	Resolved
[M-01]	Payout not resolved for losing bet	Medium	Resolved
[M-02]	A carefully timed express bet can lock up the whole liquidity pool	Medium	Acknowledged
[L-01]	Unsafe numeric casting practices	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] BetExpress is broken when a bet has a condition with more than one outcome

Severity

Impact: High, having a condition with two or more winning outcomes, the functionality of BetExpress doesn't work.

Likelihood: Medium, it happens when a condition has two or more winning outcomes.

Description

In the scenario where a deposit is made for a condition supporting two or more outcomes, BetExpress's construction logic improperly handles the `condition.payouts` array by pushing only one `subBetPayout`, despite multiple outcomes being possible.

```
if (outcomeOrdinal == 0) {
        condition.payouts.push(subBetPayout);
```

The variable `winningOutcomesCount` is designated to count the winning outcomes for a condition. If a condition is configured with two or more outcomes, this count exceeds 2, indicating multiple winning outcomes.

```
if (condition.winningOutcomesCount == 0)
        condition.winningOutcomesCount = winningOutcomesCounts[
            i
        ];
```

During liquidity reservation calculation, the `Math.maxSum` function is invoked with `condition.payouts` and `condition.winningOutcomesCount` as arguments, aiming to ensure sufficient liquidity is reserved.

```
function _calcReserve(
        Condition storage condition
    ) internal view returns (uint128) {
        return
            uint128(
                Math.diffOrZero(
                    Math.maxSum(
                        condition.payouts,
                        condition.winningOutcomesCount
                    ),
                    condition.fund
                )
            );
    }
```

However, this logic fails for conditions with multiple outcomes as the `payouts` array contains only a single element, while `n >= 2`.

```
/**
    * @notice Get the sum of `n` max items of `a`.
    */
    function maxSum(
        uint128[] memory a,
        uint256 n
    ) internal pure returns (uint256 sum_) {
        if (n < 2) return max(a);

        uint256 length = a.length;

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

        for (uint256 i = 0; i < n; ++i) {
            sum_ += sorted[length - 1 - i];
        }
    }
```

This operation leads to a revert due to underflow in the summing loop, where `length - 1 - i` becomes negative when `length = 1` and `i = 1` (since `n = 2`).

```
for (uint256 i = 0; i < n; ++i) {
            sum_ += sorted[length - 1 - i];
        }
```

Recommendations

Handling the case there is more than one winning outcome.

8.2. Medium Findings

[M-01] Payout not resolved for losing bet

Severity

Impact: Medium, affects core function resolvePayout, leading to delays in resolving bets, which in turn affects the efficient use of funds.

Likelihood: Medium, there's a common scenario where a sub-bet, not necessarily the first in the sequence, is resolved first, without it being the winning outcome.

Description

When an express bet has any of its sub-bets lost, the entire bet is considered lost and should be resolved accordingly. We can see that if `condition.state` = RESOVLED then the payout returns 0.

```
function _calcPayout(Bet storage bet) internal view returns (uint128) {
        ...
        for (uint256 i = 0; i < length; ++i) {
            ICoreBase.CoreBetData storage subBet = subBets[i];
            ICondition.Condition memory condition = core.getCondition(
                subBet.conditionId
            );

            if (condition.state == IConditionState.ConditionState.RESOLVED) {
                if (core.isOutcomeWinning(subBet.conditionId, subBet.outcomeId))
                    winningOdds = winningOdds.mul(bet.conditionOdds[i]);
                else return 0;
            } else if (
                !(condition.state == IConditionState.ConditionState.CANCELED ||
                    lp.isGameCanceled(condition.gameId))
            ) {
                revert ConditionNotFinished(subBet.conditionId);
            }
        ...
    }
```

The problem occurs because subBets are checked in order, starting from the first element. If there's a non-resolved sub-bet before a losing sub-bet, the entire express bet remains unresolved. For instance, consider an express bet consisting of three sub-bets with the following conditions:

- 1: ConditionState.CREATED
- 2: ConditionState.RESOLVED (loss)
- 3: ConditionState.RESOLVED (loss) This bet cannot be resolved.

This scenario prevents the express bet from being resolved, leading to funds being unnecessarily locked and delaying payouts to the DAO and oracles. In extreme cases, resolution awaits the final sub-bet's outcome. Each prediction engine has a locked liquidity limit level, unresolved bets can lead to other betters cannot use the engine if it is in high demand.

Recommendations

Iterate through all sub-bets to identify any losing bet promptly and return 0, allowing for the express bet to be resolved without waiting for all sub-bets to finish.

[M-02] A carefully timed express bet can lock up the whole liquidity pool

Severity

Impact: High, since no other bets can be made by any other users, and if the express bet is successful the whole liquidity pool will be drained

Likelihood: Low, since the attack can only be performed during a short timeframe and it requires significant upfront capital with a significant risk of losing that capital

Description

As detailed in the documentation and implemented in the code, there are limits on the max express bets at various levels:

- There is a maximum odds for an express bet
- Each condition in an express bet has a reinforcement limit (the max amount that can be reserved from the pool for that condition)
- There is a cap on the amount of liquidity the BetExpress contract (a core) can lock from the liquidity vault

The first two protections are adequate, however, the cap on the `reinforcementAbility` for the BetExpress contract isn't sufficiently strict. When a new BetExpress contract is plugged into a core via the Factory contract, the `addCore` method is called which sets the `reinforcementAbility` of the core to 1 by default:

```
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
        );
    }
```

This by default allows the core to lock all the liquidity from the LP. This can be seen in the `changeLockedLiquidity` method:

```
function changeLockedLiquidity(
        uint256 gameId,
        int128 deltaReserve
    ) external override isActive(msg.sender) {
        if (deltaReserve > 0) {
            uint128 _deltaReserve = uint128(deltaReserve);
            if (gameId > 0) {
                games[gameId].lockedLiquidity += _deltaReserve;
            }

            CoreData storage coreData = _getCore(msg.sender);
            coreData.lockedLiquidity += _deltaReserve;

            vault.lockLiquidity(_deltaReserve);

            if (coreData.lockedLiquidity > getLockedLiquidityLimit(msg.sender))
                revert LockedLiquidityLimitReached();
        } else
            _reduceLockedLiquidity(msg.sender, gameId, uint128(-deltaReserve));
    }
```

This method is called from BetExpress when new express bets are made to reserve liquidity from the LP for potential successful bet payouts, and it calls `getLockedLiquidityLimit` under the hood:

```
function getLockedLiquidityLimit(
        address core
    ) public view returns (uint128) {
        return
            uint128(
                _getCore(core).reinforcementAbility.mul(vault.getReserve())
            );
    }
```

So, by default, a core can lock 100% of the liquidity pool until the LP owner explicitly updates the settings of a core by calling `updateCoreSettings`.

This is particularly important for express bets because the odds can be compounded up to 1000 by default. So, a malicious user could wait for a BetExpress contract to be plugged into an existing core, at which point they could immediately place an express bet with a selection of sub bets that maximize the odds at close to 1000 without hitting the reinforcement limit for the conditions in the sub bets. An attacker that wanted to cause the maximum grief would pick these conditions to be ones that resolve as far in the future as possible to maximize the amount of time that the liquidity is locked up and the contracts are unusable for other bettors.

At this point, the admin could either:

1. Wait for the conditions to resolve and hope one of them fails the bet (otherwise the whole pool would be drained)
2. Cancel all the conditions in all the sub bets in order to force the bet id to be paid out and liquidity released back to the LP. This could negatively impact other bettors in other contracts using the same conditions in existing (non-express) bets.

Although the attacker does have to risk their capital, a sufficiently financed attacker may be willing to take that risk since there is the potential to capture the whole pool and it's likely that the LP owner would choose option 2 above to reduce downtime for other customers (which would mean the attacker gets their funds back).

Recommendations

When a new `BetExpress` contract is plugged into a liquidity pool, the factory `plugExpress` method should allow the LP owner to specify the `reinforcementAbility` of the contract. By specifying this during registration, there isn't a window after registration and before a call to `updateCoreSettings` where the BetExpress contract can lock up the liquidity pool.

8.3. Low Findings

[L-01] Unsafe numeric casting practices

Some of the casting instances from uint256 to uint128 are unsafe. They have the potential to overflow and lead to wrong accounting of the BetExpress.

There are instances when uint128() is used to convert uint256 numbers such as:

```
subBetAmount = uint128(amount_.mul(subBetReserveShare));
    subBetPayout = uint128(payout.mul(subBetReserveShare));
```

The maximum of uint128 number is in e39. Therefore, the possibility is very low for an amount with 18 decimals. However, if the system uses other tokens with higher decimals in the future then the overflow can happen. The token can be any token chosen by Liquidity Pool creator.

It is recommended to use safe casting consistently:
amount_.mul(subBetReserveShare).toUint128()