CD SECURITY

AUDIT REPORT

Azuro
February 2024

Introduction

A time-boxed security review of the **Azuro** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

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.

About **Azuro**

Azuro is a decentralized betting protocol that enables peer-to-pool betting. A `Pool` can be created through the `Factory` contract. The `Pool` consists of:

- `Access` contract which enables to owner to have access control over the `Pool`
- `LP` contract which is the main entry point and where the key functionalities of the protocol are stored
- `Betting Engine` which is a term for the contracts that take care of the betting operations

Any frontend(affiliate) can be integrated with the Azuro protocol and earn % of the profit. The users that place bets through the Azuro protocol receive an NFT that represents each bet.

The full documentation can be found here.

Threat Model

Privileged Roles & Actors

- `Owner` - Entity that can call specific methods such as `changeFee` , `changeLiquidityManager` , `changeMinDepo`
- `Affiliates` - Frontend apps that earn rewards
- `Bettors` - The users that access the markets and place bets
- `Liquidity Providers` - The LPs earn through the spread embedded in the odds on which bettors place bets.
- `Data Providers` - An entity with the necessary access in a Pool to create and cancel events, create, resolve or cancel Conditions and update the odds

Security Interview

Q: What in the protocol has value in the market?

A: All the bets placed on different market, the liquidity and the NFTs that represent the bets.

Q: In what case can the protocol/users lose money?

A: If the liquidity is drained or the users bets are voided.

Q: What are some ways that an attacker achieves his goals?

A: If an attacker is able to manipulate the outcome of sporting events or bypass the restrictions of the protocol.

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - bcb411afee212cff25ca89c06c7a643946281603**

Scope

The following smart contracts were in scope of the audit:

- `BetExpress.sol`
- `CoreBase.sol`
- `libraries/CoreTools.sol`
- `libraries/Math.sol`
- `PrematchCore.sol`
- `LP.sol`
- `StakingPool.sol`
- `extensions/ProxyFont.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 3 issues
- Low: 3 issues

Findings Summary

ID	Title	Severity
[M-01]	Liquidity providers' deposits may not be withdrawable for a long time	Medium
[M-02]	The Pool's `owner` can cancel any game for no reason	Medium
[M-03]	Missing input validation can brick the functionality of the protocol	Medium
[L-01]	Discrepancy between implementation and docs	Low
[L-02]	Lack of two-step role transfer	Low
[L-03]	Wrong Custom Error is used	Low

[M-01] Liquidity providers' deposits may not be withdrawable for a long time

Severity

Impact: High, as liquidity providers' deposits can be left stuck in the contract

Likelihood: Low, as it requires a malicious or a compromised owner

Description

Lets picture the following scenario:

1. Couple of people have deposited liquidity via `addLiquidity()` in `LP.sol`
2. A malicious or compromised person is now the owner of the contract who can call `onlyOwner` methods
3. He calls `changeWithdrawTimeout()` with the biggest value possible of uint64 `newWithdrawTimeout`
4. Now when users want to withdraw their deposited liquidity via `withdrawLiquidity()` it will always revert with `WithdrawalTimeout(_withdrawAfter - time);` custom error as you can see from the below snippet:

```
function withdrawLiquidity(uint48 depositId, uint40 percent)
        external
        returns (uint128 withdrawnAmount)
    {
        uint64 time = uint64(block.timestamp);
        uint64 _withdrawAfter = withdrawAfter[depositId];
        if (time < _withdrawAfter)
            revert WithdrawalTimeout(_withdrawAfter - time);
    }
```

Recommendations

Add a reasonable constrain for the `newWithdrawTimeout` value. Here is an example code to add in `changeWithdrawTimeout`:

```
+ uint256 maxValueTimeout = 30 days;
+ if (newWithdrawTimeout > maxValueTimeout) revert IncorrectNewTimeout();
```

[M-02] The `owner` can cancel any game for no reason

Severity

Impact: High, as users will not be able to profit from bets and this will be against the core principal of decentralization

Likelihood: Low, as it requires a malicious or a compromised owner/data provider

Description

The `cancelGame` function inside `LP` can be called only by the `owner` of the `Pool`. However, the only check is that the game has not been cancelled yet:

```
    function cancelGame(uint256 gameId)
        external
        restricted(this.cancelGame.selector)
    {
        Game storage game = _getGame(gameId);
        if (game.canceled) revert GameAlreadyCanceled();

        lockedLiquidity -= game.lockedLiquidity;
        game.canceled = true;
        emit GameCanceled(gameId);
    }
```

Consider the following scenario:

1. There is a football game between Team A and Team B. Team B is a massive underdog (let's say the odds for them is 10 for simplicity).
2. A group of user place bets on Team B winning the game with total amount of 10 000$.
3. Team B is wining 2 minutes before the end of the game. The `owner` sees that and realizes the `Pool` will have to pay out 100K $ to the users who placed a bet on Team B.
4. The `owner` cancels the event without any reason and `voids` the bets.

Recommendations

The solution for this issue is not an easy one as there may be a lot of different reasons in real life events (fans behaviour, weather conditions etc.) that could lead to a cancelled events. However, consider adding additional appropriate checks or allowing the `Data Provider` to cancel the events based on a live data.

[M-03] Missing input validation can brick the functionality of the protocol

Severity

Impact: High, as the protocol will not work as intended

Likelihood: Low, as it requires a malicious or a compromised owner

Description

There are setter functions where input parameters are not validated. The `shiftGame` function can set new start time for games but it is not checking if it is after `block.timestamp`. This could lead to setting start time in the past. The `changeFee` function allows the `owner` to set the fee for the Liquidity pool but it allows to be set to up to 100% which should not be the case.

Recommendations

Add appropriate validation for the setter functions. Also, for the `changeFee` consider adding `MAX_FEE_ALLOWED` check (e.g. 15%). If 100% fee is allowed to be set, consider adding a time lock so users have the chance to react and decide wether they want to continue to use the protocol.

[L-01] Discrepancy between implementation and docs

In the documentation, we can see the following statement about the adding of liquidity:

```
There is a lockup period of 7 days after liquidity is deposited. This
means you can provide liquidity for 7 days or more. Not less.
```

But in the LP contract from where liquidity is added, the lockup period which is `withdrawTimeout` is not set to anything. It can only be changed but even there it is not constrained to be minimum 7 days. This is misleading and can lead to errors - either update the implementation or the NatSpec accordingly.

[L-02] Lack of two-step role transfer

All of the contracts in scope have imported the `OwnableUpgradeable.sol` contract forked from OZ which means they lack two-step role transfer. The ownership transfer should be done with great care and two-step role transfer should be preferable.

Use Ownable2StepUpgradeable by OpenZeppelin.

[L-03] Wrong Custom Error is used

Inside `StakingPool`, the `beforeAddLiquidity` is checking if the user has reached a `depositLimit` :

```
        if (deposited[account] > depositLimits[account])
            revert NotEnoughStake();
```

However, if the limit is reached it reverts with `NotEnoughStake` error which naming can be frustrating for the user. Consider changing it to `DepositLimitReached`.