SMART CONTRACTS REVIEW

zokyo

August 30th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING XYRO

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 7 Critical issues: 1 acknowledged and 6 resolved = - 5 points deducted
- 5 High issues: 5 resolved = 0 points deducted
- 11 Medium issues: 2 acknowledged and 9 resolved = - 4 points deducted
- 9 Low issues: 9 resolved = 0 points deducted
- 8 Informational issues: 3 acknowledged and 5 resolved = 0 points deducted

Thus, 100 - 5 - 4 = 91

TECHNICAL SUMMARY

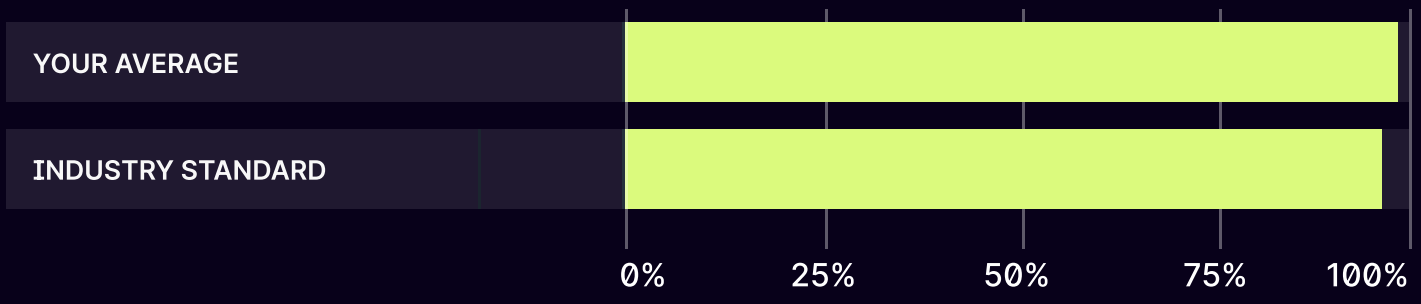
This document outlines the overall security of the Xyro smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Xyro smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 7 critical issues found during the review. (See Complete Analysis)

Testable Code



99,2% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Xyro team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Xyro repository:
Repo: https://github.com/xyro-io/smart-contracts

Last commit: b7f77decbaae02efbba0281fd8fb30d869b5d3f6

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- Bullseye.sol
- OneVsOneExactPrice.sol
- Setup.sol
- Treasury.sol
- UpDown.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Xyro smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The Xyro project features a suite of competitive betting games where players predict market movements. The primary game modes—**Up/Down, Bull's Eye, 1vs1 Exact Price**, and **Setups**—offer various betting strategies, from guessing price directions to pinpointing exact market values.

Several games are implemented:

In Up/Down, players wager on whether a price will rise or fall, with bets ranging from 5 to 100. Two pools are created, one for those betting on a price increase and another for those betting on a decrease. After a betting period, the starting price is recorded, and the game concludes by comparing the final price to the starting price. The losing pool's funds are distributed proportionally among the winners after deducting the platform's commission.

In Bull's Eye, players guess the actual market price, and the one closest to the final price wins. The game uses a single betting pool with uniform bets, and the prize distribution typically allocates 50 percent to the first place, 35 percent to the second, and 15 percent to the third, with an alternative distribution of 75 percent for the first place, 15 percent for the second, and 10 percent for the third. If a player's guess is within 0.01 percent of the actual price, the prize is increased.

The 1vs1 Exact Price game mode is similar to Bull's Eye but involves direct competition between two players who can engage in either public or private matches, with flexible betting amounts.

Setups is similar to placing an order on an exchange, where players specify take profit and stop loss levels based on the market price at the start of the game. Players can bet any amount, and the game can end when the price crosses the specified take profit or stop loss, or at a predetermined end time. Each game is assigned a unique ID for backend management, ensuring clear differentiation and effective tracking of game history, with the backend also handling bet processing, pool creation, and prize distribution, incorporating commission deductions where applicable.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Xyro team and the Xyro team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Users will lose their funds if enough players take place in a game due to push pattern	Critical	Acknowledged
2	Anyone can drain the treasury	Critical	Resolved
3	Users can lose funds due to `safeTransferFrom` after approval	Critical	Resolved
4	Funds can be stolen after wrong approve	Critical	Resolved
5	Lock of fund due to the precision loss	Critical	Resolved
6	withdrawRakeback() function will always be failed	Critical	Resolved
7	`stopLoss` and` takeProfit` can be set in the same direction, leading to `stopLoss` voters be always the winners	Critical	Resolved
8	Users Would Lose Rewards If A Huge Majority Bets On The Winning Side Due To Rounding	High	Resolved
9	Finalizing a game after 10 minutes will lock user's earned funds	High	Resolved
10	Final prices within a threshold of 10 minutes are accepted, leading to a different result than the real one and causing winners not earning funds	High	Resolved
11	One side players have more probability of winning than the other	High	Resolved
12	The opponent has a higher probability of winning than the initiator of the game	High	Resolved
13	Attacker Can Steal Initiator Fees	Medium	Resolved

#	Title	Risk	Status
14	Reward Distribution Or Refunds Can Be Griefed If One Of The Address Gets Blacklisted	Medium	Resolved
15	If Approved Token Is A Fee-On-Transfer Token Then The Refund/Distribution Would Be Broken	Medium	Resolved
16	‘DISTRIBUTOR` AND `DEFAULT_ADMIN` ROLES CAN DRAIN THE WHOLE TREASURY	Medium	Resolved
17	`collectedFee` can be easily inflated	Medium	Resolved
18	Playing is restricted for an unlimited amount of users	Medium	Resolved
19	`endTime` can be set higher than `stopPredictAt` leading to a wrong behavior of the protocol.	Medium	Resolved
20	Calculated commission cut does not align with the expected percentages	Medium	Resolved
21	Games can get closed without a reason, leading to players not being paid	Medium	Acknowledged
22	`startTime` is not checked, allowing games with past prices to be started	Medium	Resolved
23	There may exist a discordance between price units and a precision loss	Medium	Acknowledged
24	Reentrance To Drain Funds If Approved Token Is A Callback Token (ERC777)	Low	Resolved
25	CollectedFees Is Not Paid Out Anywhere	Low	Resolved
26	Users Can Make A Safe Bet In UpDown Game And Increase Their Chances Of Winning	Low	Resolved
27	`depositAmounts` marks an incorrect amount of funds after closing a game which may lead to wrong implications in future updates or external parts relying on this information	Low	Resolved
28	The `calculateUpDownRate` function does not distribute fees	Low	Resolved

#	Title	Risk	Status
29	The `withdrawFees` and `withdrawRakeback` functions are not following the decimal notation from the rest of functions, which may lead to an incorrect usage	Low	Resolved
30	Game initiator is allowed to set his own address as opponent, leading to creating a non-payable game	Low	Resolved
31	Missing zero address check	Low	Resolved
32	Overflow on updating the deposited amount	Low	Resolved
33	Created Check Can Be Bypassed	Informational	Acknowledged
34	Add A Sanity Check For End Time	Informational	Resolved
35	User can play knowing the starting price	Informational	Resolved
36	`endTime` can be set to a lower value than `startTime`	Informational	Resolved
37	A user can initiate a game with a non-reliable price	Informational	Acknowledged
38	Should use local variables instead of storage variables	Informational	Acknowledged
39	Lack of events	Informational	Resolved
40	The `refuseGame` functionality is useless	Informational	Resolved

CRITICAL-1 | ACKNOWLEDGED

Users will lose their funds if enough players take place in a game due to push pattern.

The `UpDown.sol`, Bullseye.sol, and `Setup.sol` smart contracts implement several logics using 'for loops'. These loops traverse the array of players participating in the games and execute different actions as refunding or paying earned funds. This implementation logic is known as 'push pattern' and can lead to critical situations where users are losing their funds. If the array of users grows enough so that the for loops consume enough gas for the transactions to not be able to get processed, then the users won't be able to receive their funds, and they will get stuck.

The 'push pattern' is used in several places within the protocol, being able to cause an important denial of service is critical funds like 'finalizeGame()` and 'closeGame()`. For a more in detail explanation on how to follow design pattern and why choose 'pull pattern' instead of 'push' please check our article 'Push vs Pull pattern' (https://medium.com/zokyo-io/understanding-smart-contract-design-push-vs-pull-pattern-in-evm-0108bbe673fc).

Recommendation:
Every logic that uses 'push pattern' should be changed by 'pull pattern'. Check in detail: (https://medium.com/zokyo-io/understanding-smart-contract-design-push-vs-pull-pattern-in-evm-0108bbe673fc).

Comment:
The team implemented a 'pull pattern' within the `Setup.sol` smart contract to mitigate the described issue. However, `Updown` and `Bullseye.sol` were modified with a partial fix by adding a dynamic maximum number of players that can participate in the game.

Anyone can drain the treasury.

The `Treasury.sol` smart contract implements a `calculateSetupRate()` function which should 'Calculates setup reward rate and distributes fee for setup creator` as stated in the NatSpec comment. However, this function can be called by anyone who can set `lostTeamTotal`, `wonTeamTotal` and `initiator` freely..

```
function calculateSetupRate(
      uint256 lostTeamTotal,
      uint256 wonTeamTotal,
      address initiator
  ) external returns (uint256, uint256) {
      lostTeamTotal *= 10 ** IERC20Mint(approvedToken).decimals();
      wonTeamTotal *= 10 ** IERC20Mint(approvedToken).decimals();
      uint256 withdrawnFee = (lostTeamTotal * fee) / FEE_DENOMINATOR;
      collectedFee += withdrawnFee;
      uint256 lostTeamFee = (lostTeamTotal * setupInitiatorFee) /
          FEE_DENOMINATOR;
      uint256 wonTeamFee = (wonTeamTotal * setupInitiatorFee) /
          FEE_DENOMINATOR;
      SafeERC20.safeTransfer(
          IERC20(approvedToken),
          initiator,
          lostTeamFee + wonTeamFee
      );
      //collect dust
      uint256 rate = ((lostTeamTotal - withdrawnFee - lostTeamFee) *
          FEE_DENOMINATOR) / (wonTeamTotal - wonTeamFee);
      return (rate, lostTeamFee + wonTeamFee);
  }
```

`lostTeamTotal` and `wonTeamTotal` are the parameters used for calculating the amount of fees.
`initiator` is the address of the fee receiver.

Anyone can set X values to these amounts and set himself as initiator to receive the funds and drain the whole treasury:

```
SafeERC20.safeTransfer(
          IERC20(approvedToken),
          initiator,
          lostTeamFee + wonTeamFee
      );
```

Recommendation:
If the same behavior is wanted to be kept, add an access restriction modifier to only allow certain game's contracts to call this function. If only the calculation is wanted to be kept, remove the `safeTransfer` functionality from the function.

Users can lose funds due to `safeTransferFrom` after approval.

The `deposit()` function within the `Treasury.sol` smart contract implements a `safeTransferFrom()` call:

```
function deposit(uint256 amount, address from) public {
      SafeERC20.safeTransferFrom(
          IERC20(approvedToken),
          from,
          address(this),
          amount * 10 ** IERC20Mint(approvedToken).decimals()
      );
  }
```

As can be seen, the `from` is specified within the function's parameters, and the function is public and callable by anyone.

Consider the following scenario:

1. Alice approves 1000 tokens to Treasury.sol for any reason.
2. Any directly calls `deposit()` setting Alice's address as `from`.
3. Step 2 can be executed as a front-running attack also.
4. Alice's funds are transferred to the Treasury without her being able to withdraw them.

The same issue is present within the `depositWithPermit()` function.

Recommendation:

This `deposit()` is used by other contracts to handle player's deposits, so there are several solutions:

1. Instead of directly calling Treasury's deposit function, implement a deposit() function directly in each game contract using msg.sender as `from` parameter.
2. Add an access restriction modifier to Treasury's deposit function, where only game's contracts are allowed to execute the function.

Funds can be stolen after wrong approve.

The `refund()`, `withdrawFees()` and `distributeWithoutFee()` functions within the `Treasury.sol` smart contract are used for transferring funds out from the Treasury to a receiver address. For this purpose, the functions implement a `safeTransfer` call. However, a non-necessary and risky `approve` has been added before the `safeTransfer` leading to a scenario where the receiver has been approved with X amount of tokens from the Treasury.

```
function refund(
      uint256 amount,

      address to
  ) public onlyRole(DISTRIBUTOR_ROLE) {
      IERC20(approvedToken).approve(to, amount);
      SafeERC20.safeTransfer(
          IERC20(approvedToken),
          to,
          amount * 10 ** IERC20Mint(approvedToken).decimals()
      );
  }

  /**
   * Withrad earned fees
   * @param amount amount to withdraw
   */
  function withdrawFees(
      uint256 amount,
      address to
  ) public onlyRole(DEFAULT_ADMIN_ROLE) {
      IERC20(approvedToken).approve(to, amount);
      SafeERC20.safeTransfer(IERC20(approvedToken), to, amount);
  }
```

Consider the following scenario:

1. refund() or withdrawFees() has been executed to a malicious address.
2. X amount of funds has been transferred to the malicious address.
3. X amount of funds has been approved to the malicious address.
4. If the malicious address is a smart contract that implements a function which calls a `safeTransferFrom` setting `from = treasury`, the malicious contract will be able to withdraw X amount of funds again from the treasury.
5. As a result, the malicious contract has received 2X of funds.

The same scenario is present within the `distributeWithoutFee()` function:

```
function distributeWithoutFee(
        uint256 rate,
        address to,
        uint256 initialDeposit
    ) public onlyRole(DISTRIBUTOR_ROLE) {
        initialDeposit *= 10 ** IERC20Mint(approvedToken).decimals();
        uint256 withdrawnFees = (initialDeposit * fee) / FEE_DENOMINATOR;
        uint256 wonAmount = (initialDeposit - withdrawnFees) +
            ((initialDeposit - withdrawnFees) * rate) /
            FEE_DENOMINATOR;
        IERC20(approvedToken).approve(to, wonAmount);
        SafeERC20.safeTransfer(IERC20(approvedToken), to, wonAmount);
        if (getRakebackAmount(to, initialDeposit) != 0) {
            earnedRakeback[to] += getRakebackAmount(to, initialDeposit);
        }
    }
```

Recommendation:

Remove the `approve` from `refund()`, `withdrawFees()` and `distributeWithoutFee()`.

Lock of fund due to the precision loss

In the finalizeGame() function of the Bullseye contract, if there are 2 players, then the deposited amount is divided with such rates (75/25 or 80/20).

wonAmountFirst and wonAmountSecond is calculated by multiplying the ratio into the 2 * depositAmount. However, there may be a loss of precision in this calculation and may result in locking of funds.

Consider following scenario

- depositAmount is 233 and distribution ratios are 75/25
- wonAmountFirst will be 349 and wonAmountSecond will be 116.
- The sum of 2 won amounts will be 465 and there is a loss of 1.

```
uint256 wonAmountFirst = (2 *
      game.depositAmount *
      (
           playerOneDiff <= exactRange
                ? twoPlayersExactRate[0]
                : twoPlayersRate[0]
      )) / DENOMINATOR;
ITreasury(treasury).distribute(
      wonAmountFirst,
      playerOne,
      game.depositAmount,

      fee
);
uint256 wonAmountSecond = (2 *
      game.depositAmount *
      (
          playerOneDiff <= exactRange
                 ? twoPlayersExactRate[1]
                 : twoPlayersRate[1]
       )) / DENOMINATOR;
ITreasury(treasury).distribute(
      wonAmountSecond,
      playerTwo,
      game.depositAmount,
      fee
);
```

Recommendation:

Calculate the wonAmountSecond by subtracting the wonAmountFirst from 2*depositAmount.

withdrawRakeback() function will always be failed

In the withdrawRakeback() function of the Treasury contract, it mints xyro tokens to the users. The current IERC20Mint interface has a mint() function which returns a boolean but the xyro token implementation's mint() function doesn't return any value. Due to this mismatch, the mint() function will be failed and the whole transaction will be reverted.

```
function withdrawRakeback(uint256 amount) public {
    require(
        earnedRakeback[msg.sender] >= amount,
        "Amount is greated than earned rakeback"
    );
    earnedRakeback[msg.sender] -= amount;
    IERC20Mint(xyroToken).mint(msg.sender, amount);
}
```

```
interface IERC20Mint {
    function decimals() external view returns (uint256);
    function mint(address to, uint256 value) external returns (bool);
}
```

Recommendation:

Update the interface based on the real implementation of the token contract or add a return value into the mint() function of the token implementation.

`stopLoss` and` takeProfit` can be set in the same direction, leading to `stopLoss` voters be always the winners.

The function `createSetup()` within the `Setup.sol` smart contract is used to create new Setup games. This function takes, `isLong`, `takeProfitPrice` and `stopLossPrice` as function parameters. For a correct behavior:

- If `isLong` is set to true, `stopLossPrice` should be lower than `startingPrice` and `takeProfitPrice` should be higher than `startingPrice`.
- If `isLong` is set to false, `stopLossPrice` should be higher than `startingPrice` and `takeProfitPrice` should be lower than `startingPrice`.

However, `createSetup()` does not ensure the mentioned behavior:

```
if (isLong) {
            require(
                uint192(startingPrice) / 1e14 > stopLossPrice ||

                    uint192(startingPrice) / 1e14 < takeProfitPrice,
                "Wrong tp or sl price"
            );
        } else {
            require(
                uint192(startingPrice) / 1e14 < stopLossPrice ||
                    uint192(startingPrice) / 1e14 > takeProfitPrice,
                "Wrong tp or sl price"
            );
        }
```

If can be observed that if `isLong` is set to true `startingPrice` should be greater than `stopLossPrice` or lower than `takeProfitPrice` but not both at the same time. This means that `startingPrice` can be 10, `takeProfitPrice` can be 20 and `stopLossPrice` can be 15. The same happens when `isLong` is set to false.

Recommendation:

Fix the `require` statement. Instead of using `||`, use `&&` for ensure that both conditions are met.

Users Would Lose Rewards If A Huge Majority Bets On The Winning Side Due To Rounding

Consider the following scenario for a UpDown Game (winning side is down side) →

totalDepositsDown = 100 tokens

totalDepositsUp = 100000000 tokens

Now when the up down game gets finalised the following snippet executes to calculate the rate (since down team won)→

```
} else {
            uint256 finalRate = ITreasury(treasury).calculateUpDownRate(
                _game.totalDepositsUp,
                _game.totalDepositsDown,
                fee
            );
```

And the calculateUpDownRate is as follows →

```
function calculateUpDownRate(
        uint256 lostTeamTotal,
        uint256 wonTeamTotal,
        uint256 updownFee
    ) external returns (uint256 rate) {
        lostTeamTotal *= 10 ** IERC20Mint(approvedToken).decimals();
        wonTeamTotal *= 10 ** IERC20Mint(approvedToken).decimals();
        uint256 lostTeamFee = (lostTeamTotal * updownFee) /
FEE_DENOMINATOR;
        uint256 wonTeamFee = (wonTeamTotal * updownFee) / FEE_DENOMINATOR;
        collectedFee += lostTeamFee + wonTeamFee;
        //collect dust
        rate =
            ((lostTeamTotal - lostTeamFee) * FEE_DENOMINATOR) /
            (wonTeamTotal - wonTeamFee);
```

Plugging in the values , lostTeamTotal = 100 tokens and wonTeamTotal = 100000000 tokens we get rate to be 0 due to rounding .

Now , since rate is calculated as 0 when the rewards are distributed to the winning side this snippet would be triggered →

```
for (uint i = 0; i < DownPlayers.length; i++) {
                ITreasury(treasury).distributeWithoutFee(
                    finalRate,
                    DownPlayers[i],
                    depositAmounts[DownPlayers[i]]
                );
function distributeWithoutFee(
        uint256 rate,
        address to,
        uint256 initialDeposit
    ) public onlyRole(DISTRIBUTOR_ROLE) {
        initialDeposit *= 10 ** IERC20Mint(approvedToken).decimals();
        uint256 withdrawnFees = (initialDeposit * fee) / FEE_DENOMINATOR;
        uint256 wonAmount = (initialDeposit - withdrawnFees) +
            ((initialDeposit - withdrawnFees) * rate) /
            FEE_DENOMINATOR;
        IERC20(approvedToken).approve(to, wonAmount);
        SafeERC20.safeTransfer(IERC20(approvedToken), to, wonAmount);
        if (getRakebackAmount(to, initialDeposit) != 0) {
            earnedRakeback[to] += getRakebackAmount(to, initialDeposit);
        }
```

Therefore ,wonAmount would be initialDeposit - withdrawnFees + `0` , meaning the user would actually receive less tokens than he put up to play the game . The rewards are lost and the user makes a loss instead.

A similar issue would arise for the setup game where rate is calculated via calculateSetupRate().

Recommendation:
Ensure there is sufficient precision when calculating the rate.

Finalizing a game after 10 minutes will lock user's earned funds.

The `finalizeGame()` function within the `UpDown.sol` and `OneVsOneExactPrice`, smart contract is used by the administrator to end a game, submit the results, and distribute funds to the winners.
This function implements a `require` that checks if the price submitted if within a threshold of 10 minutes:

```
require(
        priceTimestamp - game.endTime <= 10 minutes ||
        block.timestamp - priceTimestamp <= 10 minutes,
        "Old chainlink report"
    );
```

Therefore, if the function is not called within these 10 minutes the funds are not going to be distributed and winners will lose their rewards as the administrator would need to call `refund()` to return all the funds, even to losers. The function can be called after 10 minutes but with a priceTimestamp also moved in time not representing the real time when the game should have finished leading to a different result.

This situation can take place for several reasons like downtime of the chain, transactions delayed, displaced, etc.

Recommendation:
It should be possible to finish the game at any time to avoid downtimes. However, it should be assured that the reported submitted is a price's report at the exact time where the game should have finished.

Final prices within a threshold of 10 minutes are accepted, leading to a different result than the real one and causing winners not earning funds.

The `finalizeGame()` function within the `UpDown.sol` and `OneVsOneExactPrice` smart contract is used by the administrator to end a game, submit the results, and distribute funds to the winners.
Each game has an `endDate` timestamp that defines when the game finishes and when the price should be evaluated. However, the `finalizeGame()` admits a price within a 10 minutes threshold from the `endDate` timestamp.

```
require(
        priceTimestamp - game.endTime <= 10 minutes ||

            block.timestamp - priceTimestamp <= 10 minutes,

        "Old chainlink report"
    );
```

It is clear that the price can change a lot in 10 minutes and depending on which timestamp is taken to evaluate the price winners would be one side of the pool or the other.

Recommendation:
It should be assured that the reported submitted is a price's report at the exact time where the game should have finished.

One side players have more probability of winning than the other.

In the `UpDown` game, players can bet that an asset's price will be up or down compared to the current price. If the price is up, players that voted for it will win and vice versa. However, the current implementation of the `finalizeGame()` function within the `UpDown.sol` smart contract allocates more probability of winning to the players that voted down, instead of allocating it 50/50.

```
if (uint192(finalPrice / 1e14) > _game.startingPrice) { // @audit why / 1e14 phase?
            uint256 finalRate = ITreasury(treasury).calculateUpDownRate(
                _game.totalDepositsDown,
                _game.totalDepositsUp,
                fee
            );
            for (uint i = 0; i < UpPlayers.length; i++) {
                ITreasury(treasury).distributeWithoutFee(
                    finalRate,
                    UpPlayers[i],
                    depositAmounts[UpPlayers[i]]
                );
            }
            emit UpDownFinalized(finalPrice, true, currentGameId);
        } else {
            uint256 finalRate = ITreasury(treasury).calculateUpDownRate(
                _game.totalDepositsUp,
                _game.totalDepositsDown,
                fee
            );
            for (uint i = 0; i < DownPlayers.length; i++) {
                ITreasury(treasury).distributeWithoutFee(
                    finalRate,
                    DownPlayers[i],
                    depositAmounts[DownPlayers[i]]
                );
            }
            emit UpDownFinalized(finalPrice, false, currentGameId);
        }
```

As it can be observed, if the price is greater, then 'up betters' would win. However, if the price is down or equal, 'down betters' would win. Integrating the possibility of 'equal price' as a win for 'down betters' gives them a major chance of winning the game. Therefore, betting down is, by probability, the best option and unfair to 'up betters'.

Recommendation:

Implement a third case which covers the third case scenario where the price remains the same and the users are refunded.

The opponent has a higher probability of winning than the initiator of the game.

The `OneVsOneExactPrice.sol` smart contract allows creating games where the exact result of an asset should be guessed. The game is played by the initiator and an opponent. The one who guess the closest price to the final price is the winner.

```
uint256 diff1 = game.initiatorPrice > uint192(finalPrice) / 1e14
        ? game.initiatorPrice - uint192(finalPrice) / 1e14
        : uint192(finalPrice) / 1e14 - game.initiatorPrice;
    uint256 diff2 = game.opponentPrice > uint192(finalPrice) / 1e14
        ? game.opponentPrice - uint192(finalPrice) / 1e14
        : uint192(finalPrice) / 1e14 - game.opponentPrice;
    if (diff1 < diff2) {
        ITreasury(treasury).distribute(
            game.depositAmount * 2,
            game.initiator,
            game.depositAmount,
            fee
        );
        emit ExactPriceFinalized(
            gameId,
            game.initiatorPrice,
            game.opponentPrice,
            finalPrice,
            Status.Finished
        );
    } else {
        ITreasury(treasury).distribute(
            game.depositAmount * 2,
            game.opponent,
            game.depositAmount,
            fee
        );
        emit ExactPriceFinalized(
            gameId,
            game.opponentPrice,
            game.initiatorPrice,
            finalPrice,
            Status.Finished
        );
    }
```

It can be observed that if diff < diff2, meaning that the first user's price is closer to the final price than the second user's one, the first user is the winner. In any other case, the second user will be the winner. This results on the second user, the opponent, always having a higher probability of winning as if both price are equally close to the final price, the funds are not refunded but given to the second user as winning.

Users can not vote for the same price, meaning that both users can not set X as final price but they still can be equally close to the final price. Consider the following scenario:

1. The final price is Y.
2. UserA voted for Y - P.
3. UserB voted for Y + P.
4. diff1 will be P and diff2 will also be P.
5. As a result, UserB will be the winner.

Recommendation:
Add another case where if diff1 == diff2 both users get refunded and the game is closed/ finished.

MEDIUM-1 | RESOLVED

Attacker Can Steal Initiator Fees

Let's consider the following scenario →
1.) Victim creates a setup game(createSetup()) , the gameId is keccak256(

```
            abi.encodePacked(
                block.timestamp,
                endTime,
                takeProfitPrice,
                stopLossPrice
            )
        );
```

2.) The attacker sees this tx in the mempool and backruns this (in the same block so that block.timestamp is same) providing the same params , this results in the same gameID and would overwrite the section →

```
newGame.isLong = isLong;
        newGame.initiator = msg.sender;
        newGame.endTime = endTime;
        newGame.stopLossPrice = stopLossPrice;
        newGame.takeProfitPrice = takeProfitPrice;
        newGame.gameStatus = Status.Created;
        newGame.feedId = feedId;
        games[gameId] = newGame;
```

And the new initiator would be the attacker.

3.) Now during finalization of the game the initiator fee would go to the attacker instead of the original game initiator.

Recommendation:
When calculating the hash for gameId, include msg.sender too.

MEDIUM-2 | RESOLVED

Reward Distribution Or Refunds Can Be Griefed If One Of The Address Gets Blacklisted

Assuming the approved token is USDC which has a concept of blacklisting addresses , there can be a scenario where if let's say 10 people participated and one of the 3 winners (Bullseye example) gets blacklisted before the finalization/distribution of rewards then the whole function reverts. This would lead to the game never getting finished since the game won't be finalized or closed and a new game can not be started unless the previous game ends.

Recommendation:
Use a pull method to distribute rewards/refunds instead of pushing them to the players.

MEDIUM-3 | RESOLVED

If Approved Token Is A Fee-On-Transfer Token Then The Refund/Distribution Would Be Broken

Let's consider the example for Bullseye →

1. User makes a deposit using the play function and with the depositAmount , if this amount is 100 the treasury would receive something less than 100 if the token is fee on transfer , let's say treasury receives 98.
2. Assuming there were only two players the finalization of the game would result in refunds and both players would get a refund of depositAmount which is 100 , but only 98 got deposited in the treasury . This would be problematic because more is getting transferred out than received . Same problem would arise for distribution of prizes logic.

Recommendation:
Make sure to account for the correct amount received or blacklist fee on transfer tokens.

`DISTRIBUTOR` AND `DEFAULT_ADMIN` ROLES CAN DRAIN THE WHOLE TREASURY

The `Treasury.sol` smart contract implements a `refund()` and a `withdrawFees()` functions:

```
/**
    * Refunds tokens
    * @param amount token amount
    * @param to reciever address
    */
   function refund(uint256 amount, address to) onlyRole(DISTRIBUTOR_ROLE) public {
IERC20(approvedToken).approve(to, amount);
SafeERC20.safeTransfer(IERC20(approvedToken), to, amount);
   }

   /**
    * Withrad earned fees
    * @param amount amount to withdraw
    */
   function withdrawFees(uint256 amount, address to) onlyRole(DEFAULT_ADMIN_ROLE) public {
IERC20(approvedToken).approve(to, amount);
SafeERC20.safeTransfer(IERC20(approvedToken), to, amount);
   }
```

As it can be observed, both functions get an `amount` and a `to` parameters indicating the amount to be withdrawn and the receiver address, which could lead on an scenario where the trusted roles withdraw more funds than only fees or the right refund amount.

Recommendation:

As this issue is present in two different functions, it should be treated separately:

1. For the `withdrawFees()` function: implement a mapping for tracking the accumulated fees and only allow withdrawing these fees.
2. For the `refund()` function: implement a mapping with the user's deposits and only allow withdrawing the funds he has previously deposited.

`collectedFee` can be easily inflated.

The `calculateSetupRate()` and `calculateUpDownRate()` functions within the `Treasury.sol` smart contract calculates the amount of fees and updates the global variable `collectedFee` among other operations. However, this `collectedFee` is easily manipulable and get inflated by just setting X and Y values to `lostTeamTotal` and `wonTeamTotal` parameters. As the function is callable by anybody, any user can inflate the variable.
If the protocol grows and new smart contracts relies on the `collectedFee` amount, it can lead to a critical error as the value would be inflated.

Recommendation:
Add an access restriction modifier to only allow certain game's contracts to call this function.

Playing is restricted for an unlimited amount of users

In order to take part in an `UpDown` game, a player must execute the `play()` or `playWithPermit()` functions within the `UpDown.sol` smart contract. These 2 functions implements an `isParticipating` modifier:

```
/**
    * Checks if player is participating in the game
    * @param player player address
    */
   modifier isParticipating(address player) {
       for (uint i = 0; i < UpPlayers.length; i++) {
           require(UpPlayers[i] != player, "Already participating");
       }
       for (uint i = 0; i < DownPlayers.length; i++) {
           require(DownPlayers[i] != player, "Already participating");
       }
       _;
   }
```

As it can be observed, the `isParticipating()` function implements 2 'for loops' that traverse the `UpPlayers` and `DownPlayers` array, checking if the user is participating or not. This implementation leads to a problem when the arrays grow enough to run out of gas when executing the for loop.

If users continue participating in the game, the array will continue growing until certain points where it runs out of gas, leading to a scenario where no more users can take part in the game.

Recommendation:

Using an array for storing is a player has participated in the game and then implement a 'for loop' to check it is not a good design decision. Instead, use mappings to track if a user is participating or not: mapping(address ⇒ bool).

`endTime` can be set higher than `stopPredictAt` leading to a wrong behavior of the protocol.

When a new `UpDown` game is started, the admin must execute the `startGame` function within the `UpDown.sol` smart contract. This function receives several parameters, `endTime` and `stopPredictAt` among others.

```
function startGame(
      uint48 endTime,
      uint48 stopPredictAt,
      bytes32 feedId
  ) public onlyRole(DEFAULT_ADMIN_ROLE) {
```

- `stopPredictAt`: represents the deadline for users to participate in the game.
- `endTime`: represents the deadline for the asset's value to be triggered and the game finished.

It is assumed that `stopPreditAt` is lower than `endTime`, however, the `startGame` function allows setting them in the reserve assumed order or even to the same value, leading to wrong behavior of the protocol where users are allowed to participate on ended games.

Recommendation:
Add a require statement that checks if `stopPredictAt` is lower than `endTime`.

Calculated commission cut does not align with the expected percentages

The `getComissionCut()` function within the `Treasury.sol` smart contract is used to calculate the commission that must be taken from the players. After some calculations for the 'tier', the commission is calculated. However, the calculation for the commission is not following the expected design:

```
if (tier == 4) {
            //10-20%
            comissionCut = 3000;
        } else if (tier > 0) {
            //30%
            comissionCut = 1000 + 500 * tier - 1;
        }
```

As it can be observed, when the tier is equal to 4 the commission is expected to be between 10% and 20% according to the comment. However, it is set to 30%.
If the tier not 4 but greater than 0 the commission is expected to be 30%. However, it is set to another value calculated.

Recommendation:
Review the function logic and adjust the implementation to follow the designated comments if the same behavior is desired.

Games can get closed without a reason, leading to players not being paid

The `closeGame()` function within the `UpDown.sol` smart contract allows the administrator to close the game at any point in time without any reason, refunding users but not paying winners.

The same issue is present within the `Setup.sol` smart contract. The admin can close a game even after `endTime`

```
require(
        ((data.startTime + (data.endTime - data.startTime) / 3 <
            block.timestamp &&
            (games[gameId].teamSL.length == 0 ||
                games[gameId].teamTP.length == 0)) ||
            block.timestamp > data.endTime),
        "Wrong status!"
    );
```

The conditions that allows this behavior are:
1. `block.timestamp > data.endTime`.
2. `data.startTime + (data.endTime - data.startTime) / 3 < block.timestamp`.

Recommendation:
Add a condition that should be fulfilled for closing a game, for example, the `stopPredictAt` not being reached yet.
Comment: The closeGame function is needed for cases where we cannot provide a closing price for technical reasons. In this case we need to return money to players and open a new game. Therefore, we do not know how to formulate this condition.

`startTime` is not checked, allowing games with past prices to be started.

The function `createSetup()` within the `Setup.sol` smart contract is used for starting a new set up game. This function receives an `unverifiedReport` parameter which is used to determine the startingPrice of the game. The problem is that the `startTime`, which is the time where the startingPrice is considered is not checked. This makes it possible to create Setup games with startingPrices from the past, not representing the real and actual price of the assets.

```
(newGame.startringPrice, newGame.startTime) = IDataStreamsVerifier(
ITreasury(treasury).upkeep()
      ).verifyReportWithTimestamp(unverifiedReport, feedId);
```

A similar issue is present within the `setStartingPrice()` in the `UpDown.sol` smart contract. The `priceTimestamp` is checked but allows setting price from 10 minutes in the past:

```
(int192 startingPrice, uint32 priceTimestamp) = IDataStreamsVerifier(
          upkeep
      ).verifyReportWithTimestamp(unverifiedReport, game.feedId);
      require(
          block.timestamp - priceTimestamp <= 10 minutes,
          "Old chainlink report"
      );
```

The same issue is present within the `createSetUp()` function in the `Setup.sol` smart contract:

```
(int192 startingPrice, uint32 startTime) = IDataStreamsVerifier(
          ITreasury(treasury).upkeep()
      ).verifyReportWithTimestamp(unverifiedReport, feedNumber);
      if (isLong) {
          require(
              uint192(startingPrice) / 1e14 > stopLossPrice ||
                  uint192(startingPrice) / 1e14 < takeProfitPrice,
              "Wrong tp or sl price"
          );
      } else {
          require(
              uint192(startingPrice) / 1e14 < stopLossPrice ||
                  uint192(startingPrice) / 1e14 > takeProfitPrice,
              "Wrong tp or sl price"
          );
      }
```

Recommendation:

In order to implement a correct representation of the actual price of the asset, the priceTimestamp and startTime parameters should be checked to be as close to the current timestamp as possible.

There may exist a discordance between price units and a precision loss.

The whole protocol performs price down-escalations by dividing prices by `1e14`. These price variables are compared with prices directly from function parameters, which may come in terms of 18 decimals. This may lead to wrong comparisons, check the following example:

```
uint256 diff1 = game.initiatorPrice > uint192(finalPrice) / 1e14
          ? game.initiatorPrice - uint192(finalPrice) / 1e14
          : uint192(finalPrice) / 1e14 - game.initiatorPrice;
```

`game.initiatorPrice` is compared with `finalPrice / 1e14`, but `game.initiatorPrice` may come in terms of 18 decimals as it is a function parameter variable.

There is another problem related with these down escalations as precision it being lost. Solidity do not hold decimals and rounds down.
E.g: 134453782839232255 / 1e14 = 1344, losing a lot of precision.

Another example is the creation of SetUps:

```
if (isLong) {
          require(
              uint192(startingPrice) / 1e14 > stopLossPrice ||
                  uint192(startingPrice) / 1e14 < takeProfitPrice,
              "Wrong tp or sl price"
          );
      } else {
```

```
          require(
              uint192(startingPrice) / 1e14 < stopLossPrice ||
                  uint192(startingPrice) / 1e14 > takeProfitPrice,
              "Wrong tp or sl price"
          );
      }
```

The division by `1e14` affect in such a way where certain `stopLossPrice`s or` takeProfitPrices` are not accepted, but they should.

Recommendation:
Consider not escalating critical values where precision is important.

LOW-1 | RESOLVED

Reentrance To Drain Funds If Approved Token Is A Callback Token (ERC777)

When the game is being finalized by the admin (finalizeGame() in Bullseye.sol) there's a possibility that only 2 players participated in the game. A malicious admin can leverage this by being one of the 2 participant and when the admin triggers finalizeGame() it first sends the refund and only after that updates the state to `0`, if the token is a callback token with transfer hooks, the malicious admin can reenter and call finalizeGame() again to drain most of the tokens in the treasury. The same is also present in the closeGame() function.

Recommendation:
Follow Checks Effects Interaction pattern

LOW-2 | RESOLVED

CollectedFees Is Not Paid Out Anywhere

In the treasury contract collectedFees is being accounted for every winning or loss , it is updated in the distribute() , calculateUpDownRate() and calculateSetupRate() function .Admins can claim the fee using the withdrawFees() function but it does not modify the collectedFees , this would mean admin can drain any amount from the treasury as fees whereas it should be restricted to the collected fees.

Recommendation:
collectedFee should be modified in the withdrawFees() function.

Users Can Make A Safe Bet In UpDown Game And Increase Their Chances Of Winning

The users in the up down game need to bet if the price would go up or down which is calculated with respect to the starting price which is set after stopPredictAt time . This means that after every user has predicted the price movement only after that set the start price which ensures no user can game the system. But a user can still increase their chances of winning by doing the following →

1. In the play function the check to ensure if the user is not predicting after the stopPredictAt time is →

```
require(
            game.stopPredictAt >= block.timestamp,
            "Game is closed for new players"
        );
```

2. The check in the setStartingPrice is →

```
require(block.timestamp >= game.stopPredictAt, "Too early");
```

3. Therefore , it is possible that the starting price is set at the same block as the stopPredictAt and a user can see this and make a prediction at the same block since the check in the play function is >= meaning the user can make a prediction at the stopPredictAt . This can result in a safer bet for the user.

Recommendation:

Change the require statement in the play function to `require(`

```
            game.stopPredictAt > block.timestamp,
            "Game is closed for new players"
```

`depositAmounts` marks an incorrect amount of funds after closing a game which may lead to wrong implications in future updates or external parts relying on this information.

If it is decided for a game to get closed by executing `closeGame()` within the `UpDown.sol` smart contract, the `depositAmounts` variable will show an incorrect amount of funds.

The `closeGame()` function is used to refund players with their deposited funds. This function also deletes the `UpPlayers`, `DownPlayers` and `game` variables. However, the `depositAmounts` mappings are not deleted.

```
function closeGame() public onlyRole(DEFAULT_ADMIN_ROLE) {
        for (uint i; i < UpPlayers.length; i++) {
            ITreasury(treasury).refund(
                depositAmounts[UpPlayers[i]],
                UpPlayers[i]
            );
        }
        delete UpPlayers;
        for (uint i; i < DownPlayers.length; i++) {
            ITreasury(treasury).refund(
                depositAmounts[DownPlayers[i]],
                DownPlayers[i]
            );
        }
        delete DownPlayers;
        emit UpDownCancelled(currentGameId);
        currentGameId = bytes32(0);
        packedData = 0;
    }
```

Consider the following scenario:

1. Alice deposits 1 ETH to the game.
2. The game is closed, so Alice gets refunded with her 1 ETH.
3. The `depositAmounts` mapping for Alice's address is still 1 ETH, however, 1 ETH has already been distributed.

Recommendation:
Delete the `depositAmounts` mapping for the users when the `closeGame` function is executed.

LOW-5 | RESOLVED

The `calculateUpDownRate` function does not distribute fees.

The NatSpec comments from the `calculateUpDownRate()` within the `Treasury.sol` smart contract state that it should 'Calculates updown reward rate and distributes fee for setup creator'. However, no fees are distributed within the function.

Recommendation:
Add a fee distribution functionality or update the NatSpec comment.

LOW-6 | RESOLVED

The `withdrawFees` and `withdrawRakeback` functions are not following the decimal notation from the rest of functions, which may lead to an incorrect usage.

The `withdrawFees()` and `withdrawRakeback()` functions within the `Treasury.sol` smart contract are used for withdrawing fees and rakeBack. This function receives an `amount` parameter which indicates the number of fee tokens to be withdrawn. However, the `amount` is not escalated by `* 10 ** IERC20Mint(approvedToken).decimals()`, as the rest of functions from `Treasury.sol`. If in future updates `withdrawFees()` is wanted to be called from another protocol smart contract following the same logic as the rest of functions it will not behave as expected.

Recommendation:
Consider escalating amount by `* 10 ** IERC20Mint(approvedToken).decimals()` as it has been done with the rest of functions.

LOW-7 | RESOLVED

Game initiator is allowed to set his own address as opponent, leading to creating a non-payable game.

The `createGame()` and `createGameWithPermit()` functions within the `OneVsOneExactPrice.sol` smart contract allows the initiator to create a game setting his own address as `opponent` address. Then, he is not allowed to accept the game via `acceptGame()` function, therefore he would need to execute a `closeGame()` function loosing the funds needed to pay the gas.

Recommendation:
Add a require statement which does not allow setting his own address as opponent.

LOW-8 | RESOLVED

Missing zero address check

The following functions are missing zero address checks and this may result in making unexpected behaviors to zero address.

Bullseye.sol : setTreasury() function
Setup.sol : setTreasury() function
Treasury.sol : setToken() and setUpkeep() function

Recommendation:
Add zero address checks.

LOW-9 | RESOLVED

Overflow on updating the deposited amount

In the Setup contract, totalDepositsSL and totalDepositsTP are stored in packedData2 with the 32 bits size.

```
require(
    data.startTime + (data.endTime - data.startTime) / 3 >
        block.timestamp &&
        (data.totalDepositsSL + depositAmount‹ <= type(uint32).max ||
            data.totalDepositsTP + depositAmount‹ <= type(uint32).max),
    "Game is closed for new players"
);
```

However, this requirement can be bypassed if one of totalDepositsSL and totalDepositsTP doesn't exceed type(uint32).max even though another exceeds.
And this can result in overflow on updating totalDepositsSL or totalDepositsTP.

Recommendation:
If the user wants to play the game with long position, then check if totalDepositsTP updated exceeds type(uint32).max, and if short, then check if totalDepositsSL exceeds type(uint32).max.

INFORMATIONAL-1 | ACKNOWLEDGED

Created Check Can Be Bypassed

The Status enum only has 3 values i.e. Created , Canceled and Finished . Since the first value of the enum is Created that would be the default value everytime . In the play function it is checked that →

```
require(games[gameId].gameStatus == Status.Created, "Wrong status!");
```

But this would be true for non existing games too since the default value is "Created" . This check can be dropped since it offers no value or restriction.

Recommendation:
The check can be removed or update the enum to have a 4th value i.e. Default which would be the first value in the enum.

INFORMATIONAL-2 | RESOLVED

Add A Sanity Check For End Time

When starting a game (startGame() Bullseye.sol) a sanity check should be added which ensures end time > block.timestamp or some minimum amount , this might lead to the game never getting finalized.

Recommendation:
Add the recommended sanity check.

INFORMATIONAL-3 | RESOLVED

User can play knowing the starting price

Users are able to play `UpDown` knowing the selected starting point, which gives them an advantage compared to the rest of players.

The `UpDown.sol` smart contract and game should work in the following way:
1. A game is started.
2. Players place bets for up/down.
3. After every bet is placed, a starting price is set by `setStartingPrice`.

However, the `play()` and `playWithPermit()` functions implement the following time condition:

```
require(
            game.stopPredictAt >= block.timestamp,
            "Game is closed for new players"
        );
```

At the same time, the `setStartingPrice()` implements the following time condition:

```
equire(block.timestamp >= game.stopPredictAt, "Too early");
```

This allows a user to place a bet after knowing the starting price, below, in the same block.

Recommendation:
Change the time condition for `setStartingPrice()` to only allow executing the function after all bets has been placed:

`require(block.timestamp > game.stopPredictAt, "Too early");`

INFORMATIONAL-4 | RESOLVED

`endTime` can be set to a lower value than `startTime`.

The functions for creating a new game within the `UpDown.sol`` smart contract do not check if `startTime` is lower than `endTime` allowing creating game with unexpected times.

Recommendation:
Add a require statement to ensure that `endTime` is greater than `starTime`.

INFORMATIONAL-5 | ACKNOWLEDGED

A user can initiate a game with a non-reliable price.

A user can create a new 'OneVsOneExactPrice' by executing the `createGame()` function within the ``'OneVsOneExactPrice'.sol` smart contract. This function receives an `initiatorPrice` parameter, among others.

```
function createGame(
      uint8 feedNumber,
      address opponent,
      uint32 endTime,
      uint32 initiatorPrice,
      uint16 depositAmount
   ) public {
```

This parameter represents the initial price of the asset that will be considered for comparison at the end of the game. However, its value is not checked to be real, thus it can be any value used as input. As a result, a game with a non-reliable price is created.

Recommendation:
Check that the value used for creating a new game is a reliable price to avoid users accept participating in non-reliable games.

INFORMATIONAL-6 | ACKNOWLEDGED

Should use local variables instead of storage variables

In the finalizeGame() function of the Bullseye contract, if the players.length is less than 2, then only considers palyers[0].
But players[0] is read from storage multiple times after it's cached with a local variable (player). It will cost more gas.

Recommendation:
Cache the storage with a local variable.
Comment: We have thought about this problem, and have not been able to formulate when the requirements for a reliable price, so as not to hurt the capabilities of the users. And also it will lead to the fact that it is necessary to verify the starting price, which will also lead to the reflection of the game creation transaction

INFORMATIONAL-7 | RESOLVED

Lack of events

The following functions are missing events when key storages are updated.
Bullseye.sol : setTreasury() and setExactRange()
Treasury.sol : setUpkeep()
UpDown.sol : setTreasury()

Recommendation:
Add relevant events based on the variables to be updated.

The `refuseGame` functionality is useless

The `OneVsOneExactPrice.sol` smart contract implements a `refuseGame` function used for an opponent to refuse a game. However, the function is useless as if a user does not want to participate in a game, he will take no actions as executing the `refuseGame()` function will cost gas for the opponent.

Recommendation:
Consider removing the refuseGame() functionality.

	Bullseye.sol OneVsOneExactPrice.sol Setup.sol Treasury.sol UpDown.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Xyro in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the Xyro contract/s requirements for details about issuance amounts and how the system handles these.

OneVsOneExactPrice

Create game
- ✓ should create exact price game (43ms)
- ✓ should fail - wrong min bet duration
- ✓ should fail - wrong max bet duration
- ✓ should fail - Wrong deposit amount

Accept game
- ✓ should accept exact price bet
- ✓ should create and accept exact price open bet with non zero address (48ms)
- ✓ should create and accept exact price open bet with zero address (44ms)
- ✓ should fail - acceptGame wrong status (44ms)
- ✓ should fail - acceptGame game closed after 1/3 of duration (38ms)
- ✓ should fail - acceptGame same asset price
- ✓ should fail - acceptGame only opponent can accept

Close game
- ✓ should create and close game (46ms)
- ✓ should fail - closeGame wrong status (67ms)
- ✓ should fail - closeGame wrong sender

Refuse game
- ✓ should refuse game
- ✓ should fail - refuseGame only opponent can refuse bet
- ✓ should fail - refuseGame wrong status (48ms)

Finalize game
- ✓ should end the game (89ms)
- ✓ initiator should win (93ms)
- ✓ initiator should loose (95ms)
- ✓ should fail - finalizeGame wrong status (52ms)
- ✓ should fail - finalizeGame ealy finalization (64ms)

Permit

✓ should create game with permit (60ms)
✓ should fail - wrong min bet duration
✓ should fail - wrong max bet duration
✓ should fail - Wrong deposit amount
✓ should accept game with permit with non zero oponent (48ms)
✓ should accept game with permit with zero address oponent (84ms)
✓ should fail - acceptGame wrong status (48ms)
✓ should fail - acceptGame game closed after 1/3 of duration (40ms)
✓ should fail - acceptGame same asset price (41ms)
✓ should fail - acceptGame only opponent can accept (40ms)

AdminFunctions

✓ should change min and max game duration
✓ should change treasury address

UpDown

Create game

✓ should create updown game
✓ should fail - start new game without finishing previous
✓ should fail - no admin call

Play game

✓ should play down
✓ should play up
✓ should fail - already participating (up)
✓ should fail - already participating (down)
✓ should fail - Game closed

Set starting price

✓ should fail - old chainlink report (46ms)
✓ should set starting price (40ms)
✓ only owner should set starting price
✓ should fail - too early
✓ should fail - not enough players

Close game

✓ should close game and refund (71ms)
✓ only owner should close game and refund (52ms)

Finalize game

✓ should fail - game not started
✓ should fail - too early to finish (44ms)
✓ should fail - startring price should be set (50ms)
✓ only owner should finalize (66ms)
✓ should end updown game (up wins) (100ms)
✓ should end updown game (down wins) (102ms)
✓ should end updown game (no players)
✓ should end updown game (down wins, priceTimestamp > game.endTime (109ms)

✓ should fail - invalid pricetime (81ms)
✓ should refund if players only in up team (100ms)
✓ should refund if players only in down team (102ms)

Permit

✓ should play down with permit (41ms)
✓ should play up with permit (42ms)
✓ should fail - Game closed
✓ should fail - already participating (47ms)

Set

✓ should return amount of players
✓ should set a new treasury
✓ only owner should set a new treasury

Treasury

Deployment

✓ should deploy with correct values

Set a token

✓ should set a token
✓ only admin should set a token

Set a fee

✓ admin should set a fee
✓ dao should set a fee
✓ only admin or dao should set a fee

Set a setupFee

✓ admin should set a setupFee
✓ dao should set a setupFee
✓ only admin or dao should set a setupFee

Set a upkeep

✓ admin should set a upkeep
✓ only admin or dao should set a upkeep

Withdraw fee

✓ admin should withdraw fee
✓ only admin should withdraw fee

Refund

✓ admin should refund
✓ only distributor role should refund

Distribute

✓ only distributor role should distribute

Distribute without fee

✓ only distributor role should distribute

WithdrawRakeback

✓ withdraw rake back through distribute (52ms)
✓ withdraw rake back through distributeWithoutFee (41ms)
✓ withdraw rake back through distribute (47ms)
✓ should not withdraw greater than earned (44ms)

Setup Game

Create game

✓ should create SL setup game (asset price < stop loss price)
✓ should create SL setup game (asset price > take profit price)
✓ should create TP setup game
✓ should fail - wrong sl price (short)
✓ should fail - high game duration
✓ should fail - low game duration

Play game

✓ should play SL game (41ms)
✓ should play TP game (42ms)
✓ should fail - totalDepositTP > max uint32
✓ BUG (overflow) : should fail - totalDepositTP > max uint32 (56ms)
✓ should play and rewrite totalDepositsTP (57ms)
✓ should fail - can't enter twice (41ms)
✓ should fail - play wrong status
✓ should fail - enter time is up

Close game

✓ should close setup game
✓ should close game and refund for TP team (93ms)
✓ should close game and refund for TP team (90ms)
✓ should fail - close not exitsting game
✓ should fail - close under wrong status (time not passed)
✓ should fail - close under wrong status (not ended) (57ms)

Finalize game

✓ should end setup game (long) tp wins (121ms)
✓ should end setup game (long) sl wins (124ms)
✓ should fail - can't end (long) (73ms)
✓ should end setup game (short) tp wins (122ms)
✓ should end setup game (short) sl wins (119ms)
✓ should fail - can't end (short) (75ms)
✓ should fail - wrong status
✓ should fail - wrong caller (56ms)
✓ should refund if only tp team count = 0 (91ms)
✓ should refund if only sl team count = 0 (97ms)

Permit

✓ should play with permit (77ms)
✓ should revert with permit under wrong status (81ms)
✓ should fail - totalDepositTP > max uint32
✓ should fail - can not enter twice with permit (60ms)

Other

✓ should change treasury
✓ should return playes amount

✓ should change min and max game duration
✓ only admin should change min and max game duration
✓ only admin should change treasury

Bullseye

Create game

✓ should create bullseye game

Miscellaneous Cases Of Finalize

✓ Case 1: One player participated

Case 2 :

✓ Case 2: Two players participated

Case 3 :

Opponent Balance Before: 999994950000000000000000n
Opponent Balance After: 999989900000000000000000n
Alice Balance Before: 100000485000000000000000n
Alice Balance After: 100003197500000000000000n
User1 Balance Before: 100000000000000000000000n
User1 Balance After: 99999000000000000000000n
User2 Balance Before: 100000000000000000000000n
User2 Balance After: 99999742500000000000000n
User3 Balance Before: 100000000000000000000000n
User3 Balance After: 99999000000000000000000n

✓ Case 2: 5 players participated

Permit

✓ should play with permit

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	%UNCOVERED LINES
OneVsOneExactPrice.sol	100	87.88	100	100	
Treasury.sol	100	100	100	100	
Setup.sol	100	90.79	100	100	
UpDown.sol	100	100	100	100	
Bullseye.sol	96	90.21	100	100	
All Files	**99,2**	**93.77**	**100**	**100**	

We are grateful for the opportunity to work with the Xyro team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Xyro team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo