The Predicter Protocol First Flights No. 20 Audit Report

Version 1.0

GuireWire

July 25, 2024

Protocol Audit Report

GuireWire

July 25, 2024

Prepared by: GuireWire Lead Auditors: - GuireWire

Table of Contents

Protocol Summary

The protocol can be summarized as follows: - Betting On Matches (Max 30 people). - Ivan and 15 friends. 14 random users = 30 people. - Ivan is the organizer and a player. - EntranceFees are distributed at the end of the tournament between players. - Tournament start date = 15 Aug 2024 20:00:00 UTC. - 9 matches in total. - Until 16:00:00 UTC on the day of the tournament, users can register by paying entrance fees - Ivan will give priority approval to him and his 14 friends, and then will approve the remaining users. - Betting works like this: First = First Team Wins, Second = Second Team Wins, Draw = match ends in a tie. - Every day at 20:00:00 UTC, one match is played. - Up till 19:00:00 UTC, predictions can be made by approved players <- Player pays prediction fee for this. - Organizer enters the final result of the match. - Player receives 2 points if they guess right, get deducted 1 point if they guess wrong, and get nothing if they don't predict a match or don't pay the prediction fee. - After the organizer enters the reesult from the final match (Match 9), players can take their rewards from the pool. - Players can get rewards only if their points are positive and if they paid out at least one prediction fee. - If all players are negative, they receive back their initial entrance fee.

The protocol consists the following contracts.

ScoreBoard.sol

The `ScoreBoard` contract is responsible for keeping the predictions of the Players and the final results of all the matches. This contract provides the calculation of the final score (number of points) of all the Players.

- `setResult` allows the Organizer to set the final result of any match.
- `confirmPredictionPayment` is executed by `ThePredicter` contract in order to mark the corresponding prediction of the Player as paid.
- `setPrediction` sets the prediction of the Player for the given match. This function is called when the Player pays the prediction fee. It can be called again by the Players to alter their predictions without a second payment of the prediction fee which is according to the rules.
- `clearPredictionsCount` is used to make the Player ineligible for a second reward after reward collection.
- `getPlayerScore` is used to calculate the score of any Player.
- `isEligibleForReward` returns whether the Player is compliant with the rules for getting a reward.

ThePredicter.sol

The `ThePredicter` contract is supposed to manage the registration of the players, to collect the entrance and prediction fees and to distribute the rewards.

- `register` allows Users to pay the entrance fee and become cadidates for Players.
- `cancelRegistration` allows the Users which are still not approved for Players to cancel their registration and to withdraw the paid entrance fee.
- `approvePlayer` allows the Organizer to approve any user to be a Player.
- `makePrediction` allows the Players to pay the prediction fee and in the same time to set their prediction for the corresponding match.
- `withdrawPredictionFees` allows the Organizer to withdraw the current amount of the prediction fees.
- `withdraw` allows the Players to withdraw their rewards after the end of the tournament.

Disclaimer

The GuireWire 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

Commit Hash:

```
main
```

Scope

```
./src
#-- ScoreBoard.sol
#-- ThePredicter.sol
```

Roles

- Organizer = approves users to be players
- User = anyone who can register and pay entrance fee
- Player = User becomes a player after approval. Once approved, the player can now make predictions etc.

Executive Summary

Audit deemed to be successful as we identified 2 high severity and 1 medium severity vulnerabilities.

These bugs ranged from Reentrancy attacks to game mechanic violations

We spent 4 hours on the audit.

Issues found

Severity	Number of Issues Found
High	2
Medium	1
Low	2
Informational	5
Gas	1
Total	11

High

[H-1] Reentrancy Attack From Use of External Call (Not Following CEI Pattern) in `ThePredicter::cancelRegistration()` allows multiple withdrawals, draining contract funds

Summary:

The `cancelRegistration()` function in `ThePredicter.sol` is vulnerable to reentrancy attacks, allowing malicious actors to withdraw funds multiple times before their status is updated.

Vulnerability Details:

The `cancelRegistration()` function performs an external call to transfer funds before updating the player's status - does not follow CEI pattern:

```
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;
```

```
6             return;
7         }
8         revert ThePredicter__NotEligibleForWithdraw();
9     }
```

This violates the Checks-Effects-Interactions pattern, allowing for reentrancy attacks.

Impact:

An attacker can exploit this vulnerability to withdraw funds multiple times, potentially draining the contract's entire balance. This denies legitimate users their funds and disrupts the intended functionality of the contract.

Proof of Concept:

Create `MaliciousContract.sol`:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  import "../src/ThePredicter.sol";
5
6  contract MaliciousContract {
7      ThePredicter public thePredicter;
8      uint256 public constant TARGET_WITHDRAWALS = 5;
9      uint256 public withdrawalCount;
10     uint256 public totalWithdrawn;
11
12     constructor(address _thePredicter) {
13         thePredicter = ThePredicter(_thePredicter);
14     }
15
16     function attack() external {
17         withdrawalCount = 0;
18         totalWithdrawn = 0;
19         thePredicter.cancelRegistration();
20     }
21
22     receive() external payable {
23         withdrawalCount++;
24         totalWithdrawn += msg.value;
25         if (withdrawalCount < TARGET_WITHDRAWALS) {
26             thePredicter.cancelRegistration();
27         }
28     }
29 }
```

Add this test to `ThePredicter.t.sol` and run `forge test --mt testReentrancyAttack -vvvv` in terminal.

Note: Make sure to import `MaliciousContract.sol` into `ThePredicter.t.sol`.

```
function testReentrancyAttack() public {
    // Fund the ThePredicter contract
    vm.deal(address(thePredicter), 1 ether);

    // Fund and register the malicious contract
    vm.deal(address(maliciousContract), ENTRANCE_FEE);
    vm.prank(address(maliciousContract));
    thePredicter.register{value: ENTRANCE_FEE}();

    // Perform the attack
    maliciousContract.attack();

    // Assert multiple withdrawals
    assertEq(maliciousContract.withdrawalCount(), 5, "Malicious
        contract should have withdrawn exactly 5 times");
}
```

This test demonstrates that an attacker can withdraw funds 5 times instead of the intended single withdrawal. So based off this test, withdrawing multiple times, a malicious user can drain the funds from the contract.

Tools Used:

Manual review, Unit test, AI (Claude 3.5) for troubleshooting

Recommended Mitigation:

Implement the Checks-Effects-Interactions pattern in the `cancelRegistration()` function:

```
function cancelRegistration() public {
// Check
if (playersStatus[msg.sender] != Status.Pending) {
     revert ThePredicter__NotEligibleForWithdraw();
}
// Effect
uint256 refundAmount = entranceFee;
playersStatus[msg.sender] = Status.Canceled;
// Interaction
(bool success,) = msg.sender.call{value: refundAmount}("");
require(success, "Failed to withdraw");
}
```

- Additionally, consider implementing a reentrancy guard using OpenZeppelin's ReentrancyGuard contract for added security.

To further improve this function, it would be a great idea to add a check for zero address on `msg.sender`, in addition to emitting an event when a registration is cancelled.

Further Proposed Updates to Function After Implementing Correct CEI Pattern:

```
+ event RegistrationCancelled(address indexed player, uint256
    refundAmount);

function cancelRegistration() public {
+   // Check: Ensure msg.sender is not the zero address
+   require(msg.sender != address(0), "ThePredicter: Invalid address");

    // Check
    if (playersStatus[msg.sender] != Status.Pending) {
        revert ThePredicter__NotEligibleForWithdraw();
    }

    // Effect
    uint256 refundAmount = entranceFee;
    playersStatus[msg.sender] = Status.Canceled;

+   // Emit event for registration cancellation
+   emit RegistrationCancelled(msg.sender, refundAmount);

    // Interaction
    (bool success,) = msg.sender.call{value: refundAmount}("");
    require(success, "ThePredicter: Refund transfer failed");
}
```

[H-2] Unapproved players can make predictions in `ThePredicter::makePrediction` function, which violates the intended game mechanics

Summary:

The `makePrediction` function in `ThePredicter.sol` allows unapproved players to make `predictions`, violating the intended game mechanics and potentially compromising the integrity of the prediction system.

Vulnerability Details:

The `makePrediction` function lacks a check to ensure that only approved players can make predictions. This allows any registered player, regardless of their approval status, to participate in the prediction game.

Proof of Concept:

Here is a test proving how a registered user who is not an approvedPlayer can make a prediction. Add this test to `ThePredicter.test.sol` and run `forge test --mt testUnapprovedPlayerCanMakePredic -vvvv` in terminal.

```
function testUnapprovedPlayerCanMakePrediction() public {
    address unapprovedPlayer = makeAddr("unapprovedPlayer");
```

```
3       vm.deal(unapprovedPlayer, 1 ether);
4
5       // Register the player, but don't approve them
6       vm.prank(unapprovedPlayer);
7       thePredicter.register{value: ENTRANCE_FEE}();
8
9       uint256 initialBalance = address(thePredicter).balance;
10
11      // Try to make a prediction with an unapproved player
12      vm.prank(unapprovedPlayer);
13      thePredicter.makePrediction{value: PREDICTION_FEE}(0, ScoreBoard.
            Result.First);
14
15      // Check if the prediction fee was transferred
16      assertEq(address(thePredicter).balance, initialBalance +
            PREDICTION_FEE, "Prediction fee should be transferred to the
            contract");
17
18      console.log("Unapproved player successfully made a prediction");
19  }
```

Test output:

```
1  [PASS] testUnapprovedPlayerCanMakePrediction() (gas: 137928)
2  Logs: Unapproved player successfully made a prediction
```

Impact:

This vulnerability allows unauthorized participants in the prediction game, which could lead to:

- Violation of game rules and mechanics.
- Potential financial losses if unapproved players can claim rewards.
- Undermining of the system's integrity and fairness.
- Possible exploitation through creation of multiple unapproved accounts.

Tools Used:

Forge testing framework, Manual review, AI for troubleshooting

Recommended Mitigation:

Add a check in the `makePrediction` function to ensure only approved players can make predictions:

```
1  function makePrediction(uint256 _matchNumber, ScoreBoard.Result _result
       ) public payable {
2  +    require(playersStatus[msg.sender] == Status.Approved, "ThePredicter
       : Player not approved");
3
4       if (_matchNumber >= 9) {
```

```
5             revert ThePredicter__InvalidMatchNumber();
6         }
7
8         if (block.timestamp > scoreBoard.getMatchDate(_matchNumber) -
              68400) {
9             revert ThePredicter__PredictionsAreClosed();
10        }
11
12        scoreBoard.confirmPredictionPayment(msg.sender, _matchNumber);
13        // ... rest of the function
14    }
```

Medium

[M-1] Reentrancy vulnerability in `ThePredicter::withdraw` function that could lead to loss of funds

Summary: The `withdraw` function in `ThePredicter.sol` is vulnerable to reentrancy attacks due to violating the Checks-Effects-Interactions pattern, potentially allowing malicious users to withdraw multiple times and unfairly distribute rewards.

Vulnerability Details: The `withdraw` function performs state changes before making external calls:

```
1  function withdraw() public {
2      if (!scoreBoard.isEligibleForReward(msg.sender)) {
3          revert ThePredicter__NotEligibleForWithdraw();
4      }
5      // ... (score calculation and eligibility checks)
6      if (reward > 0) {
7          scoreBoard.clearPredictionsCount(msg.sender);
8          (bool success,) = msg.sender.call{value: reward}("");
9          require(success, "Failed to withdraw");
10     }
11 }
```

The vulnerability arises because `clearPredictionsCount` is called before the Ether transfer. A malicious contract could potentially re-enter withdraw during the Ether transfer.

The state update occurs before the external call, violating the Checks-Effects-Interactions pattern.

Impact:

Medium. While not directly exposing all funds to risk, this vulnerability could allow multiple withdrawals by a single user, leading to unfair reward distribution and disruption of the protocol's intended functionality.

Tools Used:

Manual Review, Forge testing framework, AI for troubleshooting

Recommended Mitigation:

Implement the CEI pattern:

```
function withdraw() public {
+   address player = msg.sender;

+   // Checks
-   if (!scoreBoard.isEligibleForReward(msg.sender)) {
-       revert ThePredicter__NotEligibleForWithdraw();
-   }
+   require(scoreBoard.isEligibleForReward(player), "Not eligible for withdrawal");
+   int8 score = scoreBoard.getPlayerScore(player);
+   require(score > 0, "Score must be positive");
+   (uint256 reward, bool isEligible) = calculateReward(score);
+   require(isEligible, "Not eligible for reward");

-   // ... (score calculation and eligibility checks)

+   // Effects
+   scoreBoard.clearPredictionsCount(player);
+   playerWithdrawals[player] = true; // New state variable to track withdrawals

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

+   emit Withdrawal(player, reward);
}
```

This re-arranges the function to the correct CEI pattern. Also, there is a few other changes to aim to mitigate the reentrancy vulnerability:

- The addition of a new state variable `playerWithdrawals` to track withdrawals.
- The removal of the if (reward > 0) check, as it's now handled by the isEligible requirement.
- The addition of an event emission for transparency.

Additionally, it would be advised to implement a reentrancy guard using OpenZeppelin's `ReentrancyGuard`. To do this;

- First, import the ReentrancyGuard contract from OpenZeppelin.

- Then, make your contract inherit from ReentrancyGuard.
- Finally, add the nonReentrant modifier to the withdraw function.

```
+ import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

- contract ThePredicter {
+ contract ThePredicter is ReentrancyGuard {
    // ... existing contract code ...

-   function withdraw() public {
+   function withdraw() public nonReentrant {
        address player = msg.sender;

        // Checks
        require(scoreBoard.isEligibleForReward(player), "Not eligible
            for withdrawal");
        int8 score = scoreBoard.getPlayerScore(player);
        require(score > 0, "Score must be positive");
        (uint256 reward, bool isEligible) = calculateReward(score);
        require(isEligible, "Not eligible for reward");

        // Effects
        scoreBoard.clearPredictionsCount(player);
        playerWithdrawals[player] = true;

        // Interactions
        (bool success,) = player.call{value: reward}("");
        require(success, "Failed to withdraw");

        emit Withdrawal(player, reward);
    }
}
```

By adding the `nonReentrant` modifier, the `ReentrancyGuard` will prevent any reentrant calls to this function, providing an additional layer of protection against reentrancy attacks. This works in conjunction with the Checks-Effects-Interactions pattern to make the function even more secure.

Low

[L-1] Centralized time management in `ThePredicter.sol` may affect tournament fairness

Summary:

`register` and `makePrediction` functions in `ThePredicter.sol` uses hardcoded timestamps and `block.timestamp` for time-based operations, which could potentially be manipulated, affecting the fairness of the tournament. Additionally the use of magic numbers for time calculations would be bad industry practice.

Vulnerability Details:

A hardcoded `START_TIME` constant determines the tournament's start:

```
uint256 private constant START_TIME = 1723752000; // Thu Aug 15 2024
    20:00:00 GMT+0000
```

The `register()` function uses `block.timestamp` for registration deadline:

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

The `makePrediction()` function also does this similarly:

```
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);
    }
```

Impact:

While funds are not directly at risk, the use of `block.timestamp` and hardcoded times could potentially be manipulated by miners to a small degree. This might allow for slightly unfair advantages in registration timing or prediction submissions. The impact is considered low as it doesn't put funds at risk and the manipulation potential is limited.

Tools Used:

Manual Review

Recommended Mitigation:

Consider implementing Chainlink's Time Oracle for more reliable and decentralized time management:

Import Chainlink's Time Oracle.

```
import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.
    sol";
```

Update `ThePredicter.sol` to use Chainlink's Time Oracle (also remove the use of Magic Numbers to follow best practices).

Note: `REGISTRATION_BUFFER` replaces the use of 14400 seconds with a constant*

```
 contract ThePredicter {
     using Address for address payable;

-    uint256 private constant START_TIME = 1723752000; // Thu Aug 15 2024 20:00:00 GMT+0000
+    AggregatorV3Interface private timeOracle;
+    uint256 public immutable START_TIME;
+    uint256 private constant REGISTRATION_BUFFER = 14400; //4Hrs in seconds

     // ... other contract variables ...

-    constructor(address _scoreBoard, uint256 _entranceFee, uint256 _predictionFee) {
+    constructor(address _scoreBoard, uint256 _entranceFee, uint256 _predictionFee, address _timeOracleAddress, uint256 _startTime) {
         organizer = msg.sender;
         scoreBoard = ScoreBoard(_scoreBoard);
         entranceFee = _entranceFee;
         predictionFee = _predictionFee;
+        timeOracle = AggregatorV3Interface(_timeOracleAddress);
+        START_TIME = _startTime;
     }

+    function getLatestTimestamp() public view returns (uint256) {
+        (, int256 answer, , ,) = timeOracle.latestRoundData();
+        return uint256(answer);
+    }
+
     function register() public payable {
         if (msg.value != entranceFee) {
             revert ThePredicter__IncorrectEntranceFee();
         }

-        if (block.timestamp > START_TIME - 14400) {
+        if (getLatestTimestamp() > START_TIME - REGISTRATION_BUFFER) {
             revert ThePredicter__RegistrationIsOver();
         }

         // ... rest of the function
     }
     }
 function makePrediction(uint256 _matchNumber, ScoreBoard.Result _result) public payable {
-       if (block.timestamp > scoreBoard.getMatchDate(_matchNumber) - 68400) {
```

```
41 +         if (getLatestTimestamp() > scoreBoard.getMatchDate(_matchNumber
       ) - PREDICTION_CUTOFF) {
42               revert ThePredicter__PredictionsAreClosed();
43           }
44           // ... rest of the function
45       }
46
47       // ... other functions ...
48   }
49   }
50    }
```

(Magic Number `14400` was replaced with `REGISTRATION_BUFFER` constant to denote 4 hours (in seconds). Using named constants is better for readability and maintainability).

These changes would improve the contract's resistance to minor time manipulations and enhance its overall reliability and fairness.

[L-2] `ThePredicter::approvePlayer` Players can be approved after tournament has started, potentially creating unfair conditions for late entrants

Summary: The `approvePlayer` function lacks a check to ensure players are not approved after the tournament has started.

Vulnerability Details:

```
1  function approvePlayer(address _player) public onlyOrganizer {
2      if (playersStatus[_player] != Status.Pending) {
3          revert ThePredicter__InvalidStatus();
4      }
5      playersStatus[_player] = Status.Approved;
6  }
```

There is no check against the tournament start time.

Impact:

Players could be approved after the tournament has started, which could lead to unfair conditions such as:

- Late entrants might be at a disadvantage as they would have missed early prediction opportunities.
- It could create confusion about the official participant list.
- It might affect the fairness of reward distribution if late entrants are included after initial calculations.
- Early participants might feel the competition terms have changed unfairly.

Tools Used:

Manual Review

Recommended Mitigation:

Add a check to ensure the tournament hasn't started:

```
function approvePlayer(address _player) public onlyOrganizer {
+    require(block.timestamp < tournamentStartTime, "ThePredicter: Tournament has already started");
     if (playersStatus[_player] != Status.Pending) {
         revert ThePredicter__InvalidStatus();
     }
     playersStatus[_player] = Status.Approved;
}
```

Note: This finding doesn't take into the consideration of using Chainlink's Time Oracle which was suggested in another finding to eliminate the centralized time management issue in [L-1].

Informational

[I-1] `ScoreBoard::setThePredicter` and `ThePredicter::withdrawPredictionFees` no event emitted, making it harder for someone to track changes

Summary:

`ScoreBoard::setThePredicter` function is designed to set the role of the Predicter. This function should emit an event when a new predicter is set.

Additionally, `ThePredicter::withdrawPredictionFees` function does not emit an event when fees are withdrawn, reducing off-chain traceability.

Vulnerability Details:

It is important to emit an event for a parameter change. In this case, the role of `thePredicter` is important as the purpose is to ensure users of this betting system know who exactly is carrying out the role of `thePredicter`.

Similarly to `ThePredicter::withdrawPredictionFees`, an event should be emitted after the transfer of fees

Impact:

The only was this harms the protocol is by not providing enough transparency for users of the betting system to know who is the `thePredicter`. Transparency is key for the security of users of the betting system. Poor transparency will discourage users from using the betting system.

Similar to `ThePredicter.sol` contract, emitting events is good practice for tracking fee withdrawals.

Tools Used:

Static Analysis (Slither), Manual Review, AI (Phind, ChatGPT)

Recommended Mitigation:

It is recommended to add an event to the `setThePredicter` function.

Updated `ScoreBoard::setThePredicter` to Add Event Emission;

Note: Remember to add the `event` to the `ScoreBoard` contract

```
event PredicterSet(address indexed newPredicter);
```

And then emit the event in the `setThePredicter` function.

```
function setThePredicter(address _thePredicter) public onlyOwner {
        thePredicter = _thePredicter;
+       emit PredicterSet(_thePredicter);
    }
```

Similarly, add the event and emit the event for `ThePredicter::withdrawPredictionFees`:

```
+ event PredictionFeesWithdrawn(address organizer, uint256 amount);

function withdrawPredictionFees() public onlyOrganizer {
    uint256 fees = address(this).balance - (entranceFee * playersCount)
        ;
    payable(msg.sender).transfer(fees);
+   emit PredictionFeesWithdrawn(msg.sender, fees);
}
```

[I-2] No Param Definitions of `enum Result` in `ScoreBoard.sol`

Summary:

`enum Result` has 4 values which are `Pending`, `First`, `Draw` and `Second`. It is best practice to define what each of these exactly do.

Vulnerability Details:

N/A - Informational finding.

Impact:

N/A - Informational finding.

Tools Used:

Manual Review

Recommended Mitigation:

Here is an example of what can be done to the `ScoreBoard::enum Result` to make it more readable.

```
+   // @param Pending: match in progress
+   // @param First: first team win
+   // @param Second: second team win
+   // @param Draw: match was tied, ie draw
    enum Result {
        Pending,
        First,
        Draw,
        Second
    }
```

[I-3] No Natspec for multiple functions in `ThePredicter.sol`, which slows down the initial process of getting context of the protocol

Summary:

No Natspec is seen for multiple functions across `ThePredicter.sol`. There are a few examples within this finding that provide a roadmap for how the developer team can implement Natspecs throughout their contracts.

Vulnerability Details:

N/A - Informational finding. No vulnerability as this is more for readability purposes.

Impact:

No Natspecs slow down the initial review process for users, developers and auditors to get a grasp on the purpose of what a function is supposed to do etc.

Tools Used:

Manual Review

Recommended Mitigation:

Here are a number of examples to mitigate this problem moving forward.

`ThePredicter::enum Status` Example

```
+ /// @notice Represents the current status of a player's registration
```

```
+ /// @dev Used to track the registration state of players in the
    prediction game
+ /// @param Unknown The initial state, indicating the player has not
    interacted  with the contract
+ /// @param Pending The player has registered but is awaiting approval
+ /// @param Approved The player's registration has been approved and
    they can participate
+ /// @param Canceled The player has canceled their registration
enum Status {
    Unknown,
    Pending,
    Approved,
    Canceled
}
```

[I-4] Magic Numbers - Define and use `constant` variables instead of using literals

Summary:

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

Vulnerability Details/Impact:

4 Found Instances

- Found in src/ScoreBoard.sol Line: 65

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

- Found in src/ThePredicter.sol Line: 93

```
        if (block.timestamp > START_TIME + matchNumber * 68400 -
            68400) {
```

Tools Used:

Static Analysis - Aderyn

Recommended Mitigation:

Use constant variables instead of literals.

[I-5] Single point of control in protocol, Example seen in `withdrawPredictionFees` function

Summary:

The `withdrawPredictionFees` function is designed with a single point of control, allowing only the organizer to withdraw fees.

Vulnerability Details:

```
function withdrawPredictionFees() public onlyOrganizer {
    // ... function implementation
}
```

The function is restricted to be called only by the organizer address.

Impact:

Informational. While this design creates a centralization point, it appears to be an intentional choice by the developers to give Ivan sole control over a number of functions.

Recommended Mitigation: No immediate action required if this is the intended design. However, for completeness, we suggest documenting this design choice and its implications:

Clearly document in the contract comments or external documentation that fee withdrawal is intentionally restricted to a single address.

Consider implementing a way to transfer this responsibility to another address in case of emergency:

```
address public organizer;

function transferOrganizer(address newOrganizer) public onlyOrganizer {
    require(newOrganizer != address(0), "Invalid new organizer address");
    organizer = newOrganizer;
    emit OrganizerTransferred(msg.sender, newOrganizer);
}
```

*Note: This example is pretty loose. Since this is not important as the intent of the protocol is supposed to be somewhat centralized, GuireWire created a short example of what a user could do in a possible worst case scenario.

Ensure there's a robust key management strategy for the organizer's address.

These suggestions aim to maintain the intended centralized control while providing flexibility and transparency in the contract's management

Gas

[G-1] Gas optimization opportunity in `ThePredicter::makePrediction` by early input validation of `_matchNumber`

Summary:

The `makePrediction` function in `ThePredicter.sol` could be optimized for gas efficiency by validating the `_matchNumber` parameter before making an external call to the `ScoreBoard.sol` contract.

Vulnerability Details:

Currently, the `makePrediction` function passes the `_matchNumber` directly to the `ScoreBoard.sol` contract without prior validation:

```
function makePrediction(uint256 _matchNumber, ScoreBoard.Result _result) public payable {
    if (block.timestamp > scoreBoard.getMatchDate(_matchNumber) - 68400) {
        revert ThePredicter__PredictionsAreClosed();
    }
    scoreBoard.confirmPredictionPayment(msg.sender, _matchNumber);
    // ... rest of the function
}
```

While the `ScoreBoard` contract appears to validate the `_matchNumber`, performing this check in ThePredicter before making the external call could save gas in cases where an invalid match number is provided.

Impact:

Gas optimization - Users may pay slightly more gas than necessary when submitting invalid match numbers.

Tools Used:

Manual review, Static Analysis - Slither

Recommended Mitigation:

Add a check for valid `_matchNumber` at the beginning of the makePrediction function:

```
function makePrediction(uint256 _matchNumber, ScoreBoard.Result _result) public payable {
+   if (_matchNumber >= 9) {
+       revert ThePredicter__InvalidMatchNumber();
+   }

    if (block.timestamp > scoreBoard.getMatchDate(_matchNumber) - 68400) {
        revert ThePredicter__PredictionsAreClosed();
    }

    scoreBoard.confirmPredictionPayment(msg.sender, _matchNumber);
    // ... rest of the function
}
```

This avoids the gas cost of an external call to ScoreBoard when the match number is invalid.