School of Informatics
Blockchains and Distributed Ledgers

Assignment 2

Erodotos Demetriou (s2187344)

Game High-Level Description

Matching pennies is a zero-sum game used in game theory. In particular, we have two players who want to bet against each other. In this game, each player will randomly pick a number 0 or 1 (we can think of it as true/false or head/tails). If both players place the same bet, *playerA* will get the reward of 1 ETH; else, if both players choose different numbers *playerB* wis. This is a zero-sum game since the reward of the winner is the loss of the other player. The diagram below illustrates the aforementioned winner/loser scheme we have described.

		Player A Bet	
		0	1
Player B Bet	0	*Player A*	*Player B*
	1	*Player B*	*Player A*

At any moment, only two players can participate in this game. They have a specific time limit to complete their round so other players can play the game later. Each player who wants to join the game has to bet 1 ETH and pay extra transaction gas fees.

It is essential to mention that when a player wants to play, he has to do this in a two-step process. Firstly, he has to obscure his bet. This is because any transaction data are visible on public-permissionless blockchains. For this reason, we use a hash function to locally hash a player's bet concatenated with a random salt that the player chooses.

$$Obscured\ Bet = H(bet||salt), \quad where\ H : Keccak256Hash$$

Using this technique, when *playerA* submits his bet on the Smart Contract, nobody can view his real bet value and thus not cheat on him. Afterward, *playerB* has to place his bet in the same manner.

When this phase completes, players' bets are locked, and they can not make any changes. Subsequently, we move to the second phase of the game, where each player has to provide his real bet and salt to validate their initial bet inputs.

We have to mention that whenever a player interacts with the Smart Contract, an event is emitted to notify the other player to progress. Each player has 10 minutes to respond, else the one who played last can request a refund and cancel the game. The player who did not responded in time, loses his ETH, which are kept by the contract.

Finally, any of the two players can call the Smart Contract evaluate function to calculate the winner. The winner can immediately withdraw his reward from the Smart Contract and reset the game settings allowing another pair of players to compete.

The above process describes the core concept behind the implementation of the game. Furthermore, during the realization of the program, several programming techniques have been used to facilitate the flawless and safe execution of the Smart Contract. We discuss such good programming practices and considerations in the "Potential Hazards and Vulnerabilities" section.

There follows a detailed description of each Smart Contract Variables, Functions, and Events:

Variables

_playDeadline:

_playedLast:

_adr_playerA:

_adr_playerB:

_bets:

Bet:

_locked:

_playersJoined:

_winner:

giveHiddenBet():

Events

event Play:

event Winner:

event NewGame:

Functions

giveHiddenBet():

giveRealBet():

eveluateWinner():

requestRefund():

withdraw():

gameReset:

Gas Costs Evaluation

This section measures and evaluates the gas cost we expect to have when deploying and interacting with the Smart Contract.

Gas costs appear in the following table, and they are unquestionably high. The Smart Contract owner has to pay 1,801,345 gas units to deploy the code, which is approximately $995 (price on 26/10/21). This is definitely a high amount of gas which makes the Smart Contract not so gas efficient.

Gas Costs

Contract Owner Fees	
Contract Deployment	1,798,945

Players Fees	***Player A***	***Player B***
giveHiddenBet()	135,643	84,620
giveRealBet()	81,007	83,807
evaluate()	40,423	
withdraw()	43,717	
requestRefund()	54,648	

In regards to the players interacting with the Smart Contract, the fees are much lower. Namely, during the initial phase of submitting a hidden bet, *playerA* has to pay 135,643 and *playerB* 84,620 gas units, translating to approximately $61 and $44, respectively. We can notice a considerable difference between the fees paying each player, making this a bit unfair.

For the next step, both players need to call the giveRealBet() function, which will cost them 81,007($42) and 83807 ($43) accordingly. Finally, the evaluation, withdrawal, and refund functions will cost 40,423($20), 43,717($20), 54,648($28) gas units.

Commenting on the above findings, it is apparent that *playerA* has a disadvantage against *playerB* in terms of gas fairness. In total, *playerA* has to pay 216,650 gas units while *playerB* 168,427 gas units, occurring a difference of 48,223. To mitigate this imbalance, we propose that *playerB* call the evaluateWinner() function narrowing the gap to 7800 gas units. Regarding refund requests and withdrawals, it is more than fair that the transaction invoker pays the proportional fee for himself.

- Techniques to make Smart Contract more fair and cost efficient.

Potential Hazards and Vulnerablities

Developing a Smart Contract for the Ethereum Network can always be challenging. This is because, on a public-permissionless blockchain, everything is observable by everyone. This fact makes it difficult when it comes to securing users' data. Besides that, a developer has to be alert to write code that is attack-resistant. Ethereum Blockchain and specifically Smart Contracts expose a broad spectrum of vulnerabilities, enabling an adversarial entity to exploit them for its interest.

When developing the Matching Pennies game, we took into consideration possible attacks. The following list presents vulnerabilities and mechanisms to moderate them.

- List of security mechanisms used to mitigate such hazards

Security vs Performance

- security vs performance trade-offs

Fellow Student Contract Analysis

- Vulnerabilities
- How the player can exploit these vulnerabilities and win the game ?
- Include code snippets

Smart Contract Execution History

Implementation Code

```
pragma solidity 0.8.0;

/// @title Matching pennies game
/// @author Erodotos Demetriou
contract Game {
    uint256 public _playDeadline;
    address public _playedLast;
    address public _adr_playerA;
    address public _adr_playerB;

    mapping(address => Bet) public _bets;

    struct Bet {
        string _realBet;
        bytes32 _hiddenBet;
        bool _isValid;
    }

    uint8 public _locked = 0;
    uint8 public _playersJoined = 0;
    address public _winner;

    event Play(address indexed _playerAddress, uint8 _playerNumber);
```

```
24      event WinnerAnnounced(address indexed _winner);
25      event NewGame(string _newGame);
26
27      /// @notice Takes 1 ETH as bet stake and set contract state accordingly
28      /// @param _bet This is an obscured 32-byte string produced after
29      /// hashing (real_bet || salt)
30      function giveHiddenBet(bytes32 _bet) public payable {
31          // Perform checks
32          require(
33              _locked == 0,
34              "There are already 2 players. Wait for the next game to start!"
35          );
36          require(msg.value == 1 ether, "You must bet 1 ETH");
37          require(
38              _bets[msg.sender]._hiddenBet == bytes32(0),
39              "You have already put your bet"
40          );
41
42          // Change the smart contract state
43          _playersJoined += 1;
44          _bets[msg.sender]._hiddenBet = _bet;
45          _playDeadline = block.timestamp + 10 minutes;
46          _playedLast = msg.sender;
47
48          // Lock the contract if both players beted
49          // and emmit events to announce their participation
50          if (_playersJoined == 2) {
51              _locked = 1;
52              _adr_playerB = msg.sender;
53              emit Play(msg.sender, 2);
54          } else {
55              _adr_playerA = msg.sender;
56              emit Play(msg.sender, 1);
57          }
58      }
59
60      /// @notice Receives the players real bets
61      /// and their salt and check the initial bet validity
62      /// @param _realBet A string representing the real bet
63      /// @param _salt The salt that the message sender used
64      /// to create his initial obscured bet
65      function giveRealBet(string memory _realBet, string memory _salt)
            external {
66          require(_playersJoined == 2, "Wait for player #2 to join the game");
67          require(
68              keccak256(abi.encodePacked(_realBet, _salt)) ==
69                  _bets[msg.sender]._hiddenBet,
70              "Error: Provided invalid input: Abort"
71          );
72
73          _bets[msg.sender]._realBet = _realBet;
74          _bets[msg.sender]._isValid = true;
75
76          _playedLast = msg.sender;
77          _playDeadline = block.timestamp + 10 minutes;
78      }
79
80      /// @notice Calculates the game winner
81      function evaluateWinner() external {
82          require(
```

```
            _bets[_adr_playerA]._isValid && _bets[_adr_playerB]._isValid,
            "Error: Players did not provide their real bet"
        );

        if (
            keccak256(abi.encode(_bets[_adr_playerA]._realBet)) ==
            keccak256(abi.encode(_bets[_adr_playerA]._realBet))
        ) {
            _winner = _adr_playerA;
        } else if (
            keccak256(abi.encode(_bets[_adr_playerA]._realBet)) !=
            keccak256(abi.encode(_bets[_adr_playerA]._realBet))
        ) {
            _winner = _adr_playerB;
        }

        // Emit event
        emit WinnerAnnounced(_winner);
    }

    /// @notice Let a player to stop the game and get
    /// refund in case his opponent griefs
    function requestRefund() external {
        // Checks
        require(
            block.timestamp > _playDeadline &&
                msg.sender == _playedLast &&
                _winner == address(0),
            "You are not allowed  to request a refund yet!"
        );

        gameReset();
    }

    /// @notice Allows the winner to withdraw his reward
    function withdraw() external {
        // Checks
        require(msg.sender == _winner, "You are not the winner!");

        gameReset();
    }

    /// @notice Send money to the winner or the
    /// refund requestor and reset game variables for a new round
    function gameReset() internal {
        _locked = 0;
        _winner = address(0);
        _playersJoined = 0;
        _bets[_adr_playerA] = Bet("", bytes32(0), false);
        _bets[_adr_playerB] = Bet("", bytes32(0), false);
        _adr_playerA = address(0);
        _adr_playerB = address(0);
        _playDeadline = 0;

        // Reward/Refund transfer
        (bool success, ) = msg.sender.call{value: 2 ether}("");
        require(success, "Error: Withdraw unsuccessful");

        // Emmit event
        emit NewGame("New game spots available");
```

```
143         }
144 }
```