Theme: “Lotery”

Team: Pinchuk Yan and Igenbekov Dinmukhamed

Contract address: 0x5435B6233964B47D9532b715d12f6694D89D189D

Github link: <https://github.com/YPNM/Final>

We took the code of Wheel of fortune from the website: [www.dougtesting.net](http://www.dougtesting.net)

In the file Winwheel.js there are all information about author and license

**Smart Contract Report - WheelOfFortune**

**Summary:** The "WheelOfFortune" smart contract is a Solidity-based implementation designed to simulate a game of chance where players can place bets and spin a virtual wheel. The contract is structured with various functions to handle bets, calculate payouts, manage player balances, and facilitate fund withdrawals. This report provides an overview and analysis of the key components of the contract.

**Contract Structure:**

1. **State Variables:**
   * **owner**: Stores the address of the contract owner.
   * **lastResult**: Keeps track of the last spin result.
   * **contractBalance**: Represents the total balance held by the contract.
   * **playerBalances**: Mapping of player addresses to their respective balances.
2. **Constants:**
   * Constants define different payout coefficients for various wheel segments.
3. **Events:**
   * **BetPlaced**: Triggered when a player places a bet.
   * **SpinResult**: Indicates the result of a spin, including the player, result, and payout.
   * **FundsWithdrawn**: Triggered when funds are withdrawn by a player.
   * **ContractBalanceUpdated**: Notifies when the contract balance is updated.
4. **Modifiers:**
   * **onlyOwner**: Restricts access to functions only to the contract owner.
5. **Constructor:**
   * Sets the initial contract balance during deployment.
6. **Functions:**
   * **placeBetAndSpin**: Allows players to place bets and spin the wheel. Generates a pseudo-random result based on block data and calculates payouts.
   * **calculatePayout**: Internal function to determine the payout based on the wheel segment and bet amount.
   * **withdrawFunds**: Enables players to withdraw funds from their balances.
   * **withdrawAllFunds**: Allows the contract owner to withdraw all funds from the contract.
   * **depositFunds**: Enables players to deposit funds into their balances.

**Security Considerations:**

* The contract uses block data and timestamps for randomness, which may not be entirely secure. Consider using an external randomness oracle for improved security.
* The contract checks for valid inputs and balances to prevent erroneous transactions.

**Recommendations:**

1. Consider implementing additional security measures, such as integrating an external randomness source.
2. Ensure that the contract adheres to the latest Solidity version and best practices.
3. Perform thorough testing, including edge cases and boundary conditions, to ensure the contract's robustness.

Our JavaScript code interacts with a smart contract deployed on the Ethereum blockchain using the web3 library. Let's break down the main components and functionalities:

1. **Initialization:**

- Checks if the browser has the Ethereum provider (MetaMask).

- Initializes the `web3` object based on the provider (MetaMask or other).

- Attempts to enable the Ethereum provider (MetaMask) if available.

- Loads the contract ABI (Application Binary Interface) from a JSON file.

2. **Contract Initialization:**

- Attempts to create a contract instance using the contract's ABI and address.

- Logs success or failure in the console.

3. **Connect Metamask Button:**

- When the "Connect MetaMask" button is clicked:

- Retrieves the user's Ethereum accounts.

- Displays the user's Ethereum address.

- Calls the `updateBalance` function to display the user's balance.

4. **Deposit Funds Function:**

- Prompts the user to enter the deposit amount.

- Converts the deposit amount to wei (the smallest unit of Ether).

- Calls the `depositFunds` method of the smart contract to deposit funds.

- Logs success or failure in the console.

- Calls the `updateBalance` function to display the updated balance.

5. **Get Last Result Function:**

- Calls the `lastResult` method of the smart contract to retrieve the last spin result.

- Logs the result in the console.

- Returns the result.

6. **Place Bet Function:**

- Retrieves the bet amount from the input field.

- Converts the bet amount to wei.

- Calls the `placeBetAndSpin` method of the smart contract to place a bet and spin the wheel.

- Logs success or failure in the console.

7. **Withdraw Funds Function:**

- Retrieves the withdrawal amount from the input field.

- Converts the withdrawal amount to wei.

- Calls the `withdrawFunds` method of the smart contract to withdraw funds.

- Logs success or failure in the console.

- Calls the `updateBalance` function to display the updated balance.

8. **Update Balance Function:**

- Checks if the contract is loaded.

- Retrieves the user's Ethereum accounts.

- Calls the `playerBalances` method of the smart contract to get the user's balance.

- Converts the balance from wei to Ether and displays it in the UI.

9. **Second Withdraw Funds Function:**

- Prompts the user to enter the withdrawal amount.

- Converts the withdrawal amount to wei.

- Calls the `withdrawFunds` method of the smart contract to withdraw funds.

- Logs success or failure in the console.

- Calls the `updateBalance` function to display the updated balance.

Note: It's important to handle user inputs carefully, validating and sanitizing them to prevent vulnerabilities such as injection attacks. Also, ensure that the contract address and ABI are correctly configured, and consider implementing error handling for a better user experience.

In the front end we added some features such as:

1. Connect Metamask wallet

You can easily connect your wallet by clicking the button

1. Add Balance

To add balance on your fortunewhell account you need to enter the amount and click the button “Add Balance”

1. After adding a balance you can play this game. First of all you need to enter the amount of bet you want to place, then you need to choose the speed of wheel and after that you can just enter Spin button and try your luck.
2. You can win x0.25, x0.5, x1, x2, x3 and x4 from your bet amount

**Conclusion:** The "WheelOfFortune" smart contract provides a basic implementation of a game of chance. While functional, it could benefit from enhancements in terms of security and adherence to best practices. Further testing and consideration of additional security measures are recommended to improve the reliability of the contract.

File description:

/contracts/FortuneWheel.sol – The main logic Solidity file of the contract

/wheel\_of\_fortune – Folder with all of the Front end files

We deployed the contract from Remix IDE, because hardhat unfortunately doesn’t works with our contract

Solidity code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract WheelOfFortune {

    address public owner;

    uint256 public lastResult;

    uint256 public contractBalance;

    mapping(address => int256) public playerBalances;

    uint256 constant public COEFFICIENT\_0\_25 = 25;

    uint256 constant public COEFFICIENT\_0\_5 = 50;

    uint256 constant public COEFFICIENT\_1 = 100;

    uint256 constant public COEFFICIENT\_2 = 200;

    uint256 constant public COEFFICIENT\_3 = 300;

    uint256 constant public COEFFICIENT\_4 = 400;

    event BetPlaced(address indexed player, uint256 betAmount);

    event SpinResult(address indexed player, uint256 result, int256 payout);

    event FundsWithdrawn(address indexed player, uint256 amount);

    event ContractBalanceUpdated(uint256 newBalance);

    modifier onlyOwner() {

        require(msg.sender == owner, "Only the owner can call this function");

        \_;

    }

    constructor(uint256 \_initialContractBalance) payable {

        owner = msg.sender;

        contractBalance = \_initialContractBalance;

        require(owner != address(0), "Invalid owner address");

        require(msg.value == \_initialContractBalance, "Incorrect initial contract balance");

        contractBalance = \_initialContractBalance;

    }

    function placeBetAndSpin(uint256 amount) external payable {

        require(amount > 0, "Bet amount must be greater than zero");

        require(playerBalances[msg.sender] >= int256(amount), "Not enough funds in the player's balance");

        require(contractBalance >= amount, "Not enough funds in the contract");

        playerBalances[msg.sender] -= int256(amount);

        emit BetPlaced(msg.sender, amount);

        // Используем хэш текущего блока и времени для генерации случайного числа

        bytes32 randomSeed = keccak256(abi.encodePacked(block.number, block.timestamp, msg.sender));

        uint256 randomResult = uint256(randomSeed) % 10;

        lastResult = randomResult;

        int256 payout = calculatePayout(randomResult, amount);

        playerBalances[msg.sender] += payout;

        // Если игрок проиграл, добавляем проигрыш на баланс контракта

        if (payout <= 0) {

            contractBalance += uint256(-payout);

        }

        emit SpinResult(msg.sender, randomResult, payout);

        emit FundsWithdrawn(msg.sender, uint256(payout));

}

    function calculatePayout(uint256 segmentNumber, uint256 betAmount) internal pure returns (int256) {

        if (segmentNumber == 0) {

            return int256(betAmount \* COEFFICIENT\_0\_25 / 100);

        } else if (segmentNumber == 1) {

            return int256(betAmount \* COEFFICIENT\_0\_5 / 100);

        } else if (segmentNumber == 2 || segmentNumber == 10) {

            return int256(betAmount \* COEFFICIENT\_1 / 100);

        } else if (segmentNumber == 8) {

            return int256(betAmount \* COEFFICIENT\_2 / 100);

        } else if (segmentNumber == 3 || segmentNumber == 5) {

            return int256(betAmount \* COEFFICIENT\_3 / 100);

        }

        else if (segmentNumber == 4) {

            return int256(betAmount \* COEFFICIENT\_4 / 100);

        }

        else {

            return -int256(betAmount);

        }

    }

    function withdrawFunds(uint256 withdrawAmount) external {

        uint256 playerBalance = uint256(playerBalances[msg.sender]);

        require(playerBalance >= withdrawAmount, "Not enough funds to withdraw");

        playerBalances[msg.sender] -= int256(withdrawAmount);

        payable(msg.sender).transfer(withdrawAmount);

        emit FundsWithdrawn(msg.sender, withdrawAmount);

    }

    function withdrawAllFunds() external onlyOwner {

        require(contractBalance > 0, "No funds to withdraw");

        uint256 balanceToWithdraw = contractBalance;

        contractBalance = 0;

        payable(owner).transfer(balanceToWithdraw);

        emit ContractBalanceUpdated(contractBalance);

    }

    function depositFunds() external payable {

        require(msg.value > 0, "Deposit amount must be greater than zero");

        playerBalances[msg.sender] += int256(msg.value);

        contractBalance += msg.value;

        emit FundsWithdrawn(msg.sender, msg.value);

        emit ContractBalanceUpdated(contractBalance);

    }

}

Js Code:

let web3, contract, userAddress;

window.onload = async () => {

  if (window.ethereum) {

    web3 = new Web3(window.ethereum);

    try {

      await window.ethereum.enable();

    } catch (error) {

      alert('You need to allow access to your MetaMask account.');

      return;

    }

  } else if (window.web3) {

    web3 = new Web3(window.web3.currentProvider);

  } else {

    alert('Please install MetaMask!');

    return;

  }

  const abi = await fetch('js/contractABI.json').then(response => response.json());

  try {

    contract = new web3.eth.Contract(abi, '0x5435B6233964B47D9532b715d12f6694D89D189D');

    console.log("Contract loaded successfully");

  } catch (error) {

    console.error("Error loading contract:", error);

  }

  document.getElementById('connectMetamask').onclick = async () => {

    const accounts = await web3.eth.getAccounts();

    userAddress = accounts[0];

    document.getElementById('userAddress').innerText = "User address = " + userAddress;

    updateBalance();

  };

};

async function depositFunds() {

  const depositAmount = prompt("Enter the deposit amount:");

  if (isNaN(depositAmount) || depositAmount <= 0) {

      alert("Invalid deposit amount");

      return;

  }

  try {

      const accounts = await web3.eth.getAccounts();

      await contract.methods.depositFunds().send({

          from: accounts[0],

          value: web3.utils.toWei(depositAmount, 'ether') // Преобразуем в wei

      });

      console.log("Deposit successful");

      updateBalance();

  } catch (error) {

      console.error("Error depositing funds:", error);

  }

}

async function getLastResult() {

  try {

    const result = await contract.methods.lastResult().call();

    console.log('Last result:', result.toString());

    return result;

  } catch (error) {

    console.error('Error fetching last result:', error);

  }

}

async function placeBet() {

  const betAmount = document.getElementById('betAmount').value;

  try {

      const accounts = await web3.eth.getAccounts();

      await contract.methods.placeBetAndSpin(web3.utils.toWei(betAmount, 'ether')).send({ from: accounts[0] });

  } catch (error) {

      console.error("Error placing bet:", error);

  }

}

async function withdrawFunds() {

  const withdrawAmount = document.getElementById('withdrawAmount').value;

  try {

      const accounts = await web3.eth.getAccounts();

      await contract.methods.withdrawFunds(withdrawAmount).send({ from: accounts[0] });

  } catch (error) {

      console.error("Error withdrawing funds:", error);

  }

}

async function updateBalance() {

  try {

    // Проверяем, что контракт успешно загружен

    if (!contract) {

        console.error("Contract is not loaded");

        return;

    }

    const accounts = await web3.eth.getAccounts();

    const balance = await contract.methods.playerBalances(accounts[0]).call({ from: accounts[0] });

    document.getElementById('userBalance').innerText = "Your balance = " + web3.utils.fromWei(balance, 'ether');

  } catch (error) {

    console.error("Error updating balance:", error);

  }

}

async function withdrawFunds() {

  const withdrawAmount = prompt("Enter the withdrawal amount:");

  if (isNaN(withdrawAmount) || withdrawAmount <= 0) {

    alert("Invalid withdrawal amount");

    return;

  }

  try {

    const accounts = await web3.eth.getAccounts();

    await contract.methods.withdrawFunds(web3.utils.toWei(withdrawAmount, 'ether')).send({

      from: accounts[0],

    });

    console.log("Withdrawal successful");

  } catch (error) {

    console.error("Error withdrawing funds:", error);

  }

  updateBalance();

}