Homework 2: Introduction to Solidity and Ethereum

February 14, 2024

Introduction

This homework is designed to introduce you to the fundamentals of Solidity and the Ethereum blockchain. You will learn how to interact with Ethereum testnets, deploy smart contracts, and understand the basics of blockchain-based applications.

Part 1: Setting up MetaMask and Connecting to a Testnet

1. **Install MetaMask**
   - Use Chrome or Chromium.
   - Navigate to the MetaMask website: `https://metamask.io`.
   - Select the version suitable for your browser and click on "Download" then "Install MetaMask for [Your Browser]".
   - Follow the installation instructions to add the MetaMask extension to your browser.
   - For convenience, pin the MetaMask plugin to your toolbar: click the plugins (jigsaw) icon on the top right and click the pin icon next to "MetaMask".
2. **Create a New Wallet**

- Open the MetaMask extension and click on "Get Started".
- Choose "Create a Wallet" and agree to the terms.
- Set a strong password, then write down and safely store your secret recovery phrase.
- Confirm your secret seed (recovery phrase) to complete the setup.

3. **Connect to the Sepolia Testnet**

    - In MetaMask, click on the "Ethereum Mainnet" button at the top to open the network selection dropdown.
    - Turn on "Show test networks".
    - Click "Sepolia".

4. **Obtain Test SepoliaETH from a Faucet**

    - Visit the faucet URL: `https://sepolia-faucet.pk910.de/`.
    - Follow the site's instructions to receive test ETH by entering your MetaMask wallet address.
    - Wait until the website indicates you're eligible to withdraw. This may take about an hour and do a lot of computation (so don't run on battery power).
    - Check your MetaMask wallet for the updated balance with the received test ETH.

Part 2: Interacting with a Deployed Contract

We have deployed the following smart contract on the Sepolia Testnet at the address `0x1e33DaE11dcd6197259673C286C1F56e75A46A18`. This smart contract exemplifies basic blockchain data storage and retrieval interactions, providing a foundation for understanding how smart contracts can manage and provide access to on-chain data.

```
pragma solidity ^0.8.0;

contract StringStorage {
    // Array to store strings
    string[] public strings;

    // Event declaration for logging string submission
    event StringSubmitted(string submittedString, uint256 index);

    // Function to submit a string. Any address can call this function.
    function submitString(string memory newString) public {
        strings.push(newString); // Add the new string to the array
        // Emit an event with the submitted string and its index
        emit StringSubmitted(newString, strings.length - 1);
    }

    // Function to retrieve a string by index
    function getString(uint256 index) public view returns (string memory) {
        // Check that the index is valid
        require(index < strings.length, "Index out of bounds");
        return strings[index]; // Return the string at the specified index
    }

    // Function to get the total number of strings stored
    function getTotalStrings() public view returns (uint256) {
        return strings.length; // Return the length of the strings array
    }

    // Function to retrieve a the last string if it exists
    function getLastString() public view returns (string memory) {
        require(strings.length > 0, "No strings submitted yet");
        return strings[strings.length - 1]; // Check that the index is valid
    }

}
```
Listing 1: StringStorage Smart Contract

- **Storing Strings:** Users can submit strings to the contract, which are then stored in an array. Each time a string is submitted, the contract emits an `StringSubmitted` event, logging the string and its index within the array.

- **Retrieving Strings:** The contract offers several functions for data retrieval:
    1. `getString(uint256 index)` allows fetching a string by its index in the storage array.
    2. `getTotalStrings()` returns the total number of strings stored.
    3. `getLastString()` retrieves the most recently submitted string.
- **Events for Tracking:** The `StringSubmitted` event facilitates tracking new string submissions in real-time or analyzing the history of stored data by external applications or observers.

1. Visit Etherscan (`https://sepolia.etherscan.io`) and locate our deployed toy contract.
2. Use MetaMask to interact with the contract. You are provided with three files to help you do so. These run in-browser so they can talk to the MetaMask browser extension, and are therefore written in JavaScript.
    (a) `app.html`: This contains a simple UI in HTML that allows you to interact with the deployed smart contract using metamask.
    (b) `js/app.js`: This file contains all the scripts needed behind the scenes to make the interaction happen.
    (c) `js/abi.js`: This file contains the ABI (Application Binary Interface) of our deployed smart contract. The ABI of a smart contract is a JSON-formatted interface descriptor that outlines how to interact with the contract, detailing its functions, arguments, and return types. It acts as an interface, enabling applications to call functions and receive outputs from contracts on the Ethereum blockchain. In general, the ABI is automatically generated by the compiler of your smart contract (for example, Remix).
3. Open `app.html` and write a message to get submitted to the smart contract. Document the transaction ID and your message.

    Note that for security reasons, browsers restrict running files directly from your directory (using `file://...` URLs). Thus, to run your `app.html`, you need to run an HTTP server that serves that file. A simple way to do that is to run `python3 -m http.server 8000` within the directory where `app.html` is placed, and then open the URL `http://localhost:8000/app.html` in your browser.

4. Bonus: Modify `app.js` and contract.html to add a button. When this button is pressed you should fetch all submitted strings so far from the smart contract and display them.

Part 3: Writing and Deploying a Smart Contract

Now that you completed Part 2 and have interacted with an already deployed smart contract, your next job is to write a smart contract yourself!

The smart contract will be conceptually similar to the setup of Homework 1. However, this time *you* will we be one handing out the secrets and tokens. And you will be handing out your own custom token, instead of testnet bitcoins.

1. Using Remix IDE (`https://remix.ethereum.org`), write a smart contract in Solidity to create a new fungible token.

2. Implement a function for an "airdrop" that gets a set of secret strings.. After that function is called, any party who knows one of these strings can use it to claim 1 token. You should do this by hard coding a list of hash digest, each digest correspond to a different secret. A secret can only be used once, to claim 1 token.

3. Deploy your contract to a testnet and verify its functionality by claiming two tokens using two different secrets.

4. Submit your contract's source code, the deployed address, and transaction IDs for the claimed tokens.

Part 4: Theoretical Optimization

The implementation in part 3 had a list of hashes hard-coded in the contract, and this cause the contract to be very large if there are many secrets. This would cause a high gas cost for the issuer (i.e., you).

Discuss how you could reduce the cost for the airdrop issuer, making it grow much slower (as a function of the number of secrets). What new code would be needed to be written and who would run it?

Submission Guidelines

Please submit all parts of your homework as a zip file. Include any relevant code snippets, transaction IDs, contract addresses, and your discussions. Ensure your submissions are clear, concise, and well-documented.