Smart Contracts Exercise 04: Unbreakable Vault

1 Introduction

In this exercise, you will be tasked with breach several vaults, one by one. You will gain familiarity with the JavaScript library Ethers.js, which is designed to facilitate interaction with the Ethereum blockchain and its ecosystem. We will also demonstrate how to work in Remix IDE, an open-source development environment accessible through a web browser. Additionally, you will learn about blockchain data transparency, the differences between `msg.sender` and `tx.origin`, and how to predict `blockhash` or `block.timestamp` in certain scenarios.

Prerequisites

Ensure that you have already installed the following on your system:

- **Node.js** - `https://nodejs.org/en/` An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development. It is recommended that you use NPM 7 or higher.

For the purposes of this exercise, you will need an Infura API key and a configured wallet. If you do not have this set up yet, we recommend going through the Smart Contracts Exercise 01: Hello, Blockchain World! where everything is explained. Ensure that configuration variables are set for your Hardhat projects. You can verify this by running:

```
$ npx hardhat vars get INFURA_API_KEY
$ npx hardhat vars get SEPOLIA_PRIVATE_KEY
```

Project Set Up

To get started, visit the following GitLab repository and clone it to your local machine. This repository contains a template in which you will complete this exercise. After you clone the repository, start with the following command within your project folder:

```
$ npm install
```

This will install all the necessary dependencies for the project. Your implementation will be in the `contracts` and `test` folders. There will be multiple vaults in this exercise that you need to breach, each one having a separate test. To see if you have completed the task successfully, run `npm run vaultXX` where `XX` is the number of the vault you are trying to breach. For example, to test the first vault, run:

```
$ npm run vault01
```

To run all tests at once, run:

```
$ npx hardhat test
```

2 Task: Breach the Vaults

Vault01: A Password Password

The first vault is quite straightforward. To complete this challenge, you need to call the `breachVault` function with the correct password and become the `lastSolver`. Implement your solution in `test/Vault01.js`. Do not alter the contract code. Use only the `player` account to breach the vault.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault01 {
    address public lastSolver;

    function breachVault(uint256 _password) public returns (bool) {
        require(
            _password == uint256(keccak256("password")),
            "Incorrect password"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Sources you might want to use:

- https://docs.ethers.org/v6/api/hashing/
- https://docs.soliditylang.org/en/latest/

Verify your solution with:

```
$ npm run vault01
```

Remix IDE

In this exercise, each individual task is also available in Remix IDE. Remix is a versatile tool that requires no installation, promotes rapid development, and offers a wide range of plugins with intuitive GUIs created by Ethereum foundation. It is available as both a web application and a desktop application. The purpose of this is to familiarize you with the basic operations in this program and to facilitate your interaction with smart contracts.

Vault01 in Remix IDE

How to get started with Remix:

- Getting Started With Remix (Solidity) in 2 mins
- https://remix-ide.readthedocs.io/en/latest/

Vault02: Packet Sender

There is nothing new here, the previous hints are enough for you to break into this vault! Solve the challenge in `test/Vault02.js`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault02 {
    address public lastSolver;

    function breachVault(uint256 _password) public returns (bool) {
        require(
            _password == uint256(keccak256(abi.encodePacked(msg.sender))),
            "Incorrect password"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault02 in Remix IDE

Vault03: Origins

In the Ethereum network, there are two main types of accounts:

- Externally Owned Accounts (EOAs)
- Smart Contract Accounts (SCAs)

EOAs are managed by private keys, SCAs are governed by smart contract code.

To breach the third vault, you need to understand the difference between `msg.sender` and `tx.origin`. The key distinction is that `tx.origin` always refers to the original external account that initiated the transaction, while `msg.sender` can be any contract or account that called the current function. As illustrated in the graph below (see Figure 1), smart contracts can call other smart contracts, but only an externally owned account (EOA) can initiate a transaction and forward the gas. It is important to never use `tx.origin` for authentication. For this challenge, you cannot implement the solution directly in `test/Vault03`. Instead, you need to use a proxy contract. Implement your solution in `contracts/AttackVault03.sol`.

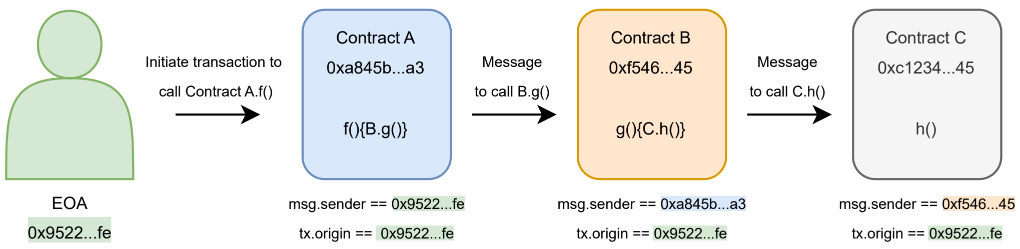

Figure 1: msg.sender vs tx.origin

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault03 {
    // Address of the last person who solved the challenge
    address public lastSolver;

    function breachVault() public returns (bool) {
        require(msg.sender != tx.origin,
            "Caller must not be the transaction origin"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault03 in Remix IDE

Optional deep dive: EIP 4337 is a proposal that aims to enable smart contract-like functionality for all accounts, effectively eliminating the distinction between Externally Owned Accounts (EOAs) and Smart Contract Accounts (SCAs). This would allow for more advanced and flexible control over account operations, including features like gas sponsorship, multi-signature authentication, and custom transaction logic.

Vault04: On the Same Block

For this vault, you will need a proxy contract as well. Implement your solution in the `contracts/AttackVault04.sol` file. The hint is hidden in the name of the challenge.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault04 {
    address public lastSolver;

    function breachVault(bytes32 _password) public returns (bool) {
        require(
            _password ==
```

```
                keccak256(
                    abi.encodePacked(
                        blockhash(block.number - 1),
                        block.timestamp
                    )
                ),
            "Incorrect password"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault04 in Remix IDE

Vault05: Fortune Teller

This vault cannot be opened without a crystal ball. Or can it? Implement your solution in the `test/Vault05.sol` file. Look for hints here: Units and global variables in Solidity.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault05 {
    address public lastSolver;
    bytes32 private password;
    uint256 private lockInBlockNumber;

    function lockInPassword(bytes32 _password) public {
        password = _password;
        lockInBlockNumber = block.number;
    }

    function breachVault() public returns (bool) {
        require(block.number > lockInBlockNumber, "Wait for the next block");
        require(password == blockhash(lockInBlockNumber), "Incorrect password");
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault05 in Remix IDE

Vault06: Explorer

Ethereum has three main types of memory when executing smart contracts:

- **Storage**
  - Data stored in **storage** is written to the blockchain and persists between transactions.
  - It is the most expensive type of memory to modify because it requires writing to disk (state updates).
  - Defined with the `storage` keyword or used implicitly for state variables.

- **Memory**
  - Memory is temporary and only lasts during the execution of a transaction.
  - It is used for function execution and is cleared at the end of each transaction.
  - Declaring variables with `memory` ensures they are not stored on-chain, reducing gas costs.
- **Stack**
  - The stack is a small, ultra-fast memory section used for local variable storage and function execution.
  - It has a strict size limit (1024 slots), meaning complex operations often require `memory` or `storage`.

It's important to note that marking a variable as private only restricts access from other contracts. Private state variables and local variables remain publicly accessible. For this task, there is already a deployed contract on the Sepolia testnet. You can find the contract address and the source code below. Implement your solution in the `test/Vault06.js` file. The address of the deployed contract is `0xA3a763bF62550511A0E485d6EB16c98937609A32`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault06 {
    address public lastSolver;
    string private password;

    constructor(string memory _password) {
        password = _password;
    }

    function breachVault(string memory _password) public returns (bool) {
        require(
            keccak256(abi.encodePacked(password)) ==
                keccak256(abi.encodePacked(_password)),
            "Incorrect password"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Hint: For this challenge, you can find the solution just by closely inspecting the contract on: Vault06 on Sepolia Etherscan

You can also interact with the contract directly from the Remix IDE if you connect it to your MetaMask wallet, change the environment to "Injected Provider - MetaMask" and use the "Load contract from address" function.

Vault06 in Remix IDE

Vault07: You Shall Not Pass!

In this exercise, you will not be able to find the solution just by inspecting Etherscan as you did in the previous exercise. Instead, you will need to decode the storage yourself.

You can find the necessary functions here: https://docs.ethers.org/v6/api/providers/. Implement your solution in the `test/Vault07.js` file. The address of the deployed contract is `0xa81C96B2216eDFfF8945e371dd581D13f8ECfbAD`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract Vault07 {
    address public lastSolver;
    uint8 private small1 = 42;
    uint16 private small2 = 999;
    bool private isActive = true;
    uint256 private big1 = 1337;
    bytes32 private hashData = keccak256(abi.encode("You Shall Not Pass"));
    uint256 private big2 = 0xDEADBEEF;
    string private password;

    constructor(string memory _password) {
        password = _password;
    }

    function breachVault(bytes32 _password) public returns (bool) {
        require(
            keccak256(abi.encodePacked(password, msg.sender)) == _password,
            "Incorrect password"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault07 in Remix IDE

Hint: Exploring the Storage Layout in Solidity and How to Access State Variables