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, storage, randomness pattern and the differences between `msg.sender` and `tx.origin`.

Project Setup

You have two options for working with this exercise. Using docker container or local installation. Choose the one that best fits your preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click ”Reopen in Container” or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally.

Prerequisites

- **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.

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `npm install`.

1.3 Testing Your Solution

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
```

If you are using Docker, set them in your current container by running:

```
$ npx hardhat vars set INFURA_API_KEY
Enter value: ********************************

$ npx hardhat vars set SEPOLIA_PRIVATE_KEY
Enter value: ********************************
```

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`.

```solidity
// 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 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`.

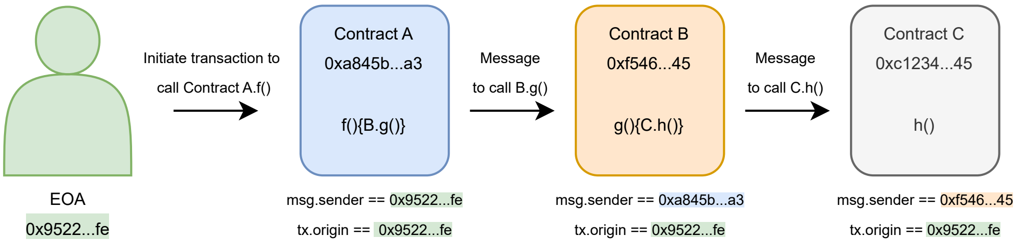

Figure 1: msg.sender vs tx.origin

```
// 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 and Smart Contract Accounts. 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: Pseudo-Random Trap

Generating a random number in Ethereum can be tricky. The Ethereum Virtual Machine executes smart contracts in a deterministic environment, ensuring consistent outputs for identical inputs. Solidity does not provide anu function that generate random numbers. To work around this, some developers use pseudo-random number generators (PRNGs) based on block properties like:

- `blockhash(uint blockNumber)`: Hash of a recent block.
- `block.number`: Current block number.
- `block.timestamp`: Block timestamp (Unix epoch).

The problem is, that block proposers might attempt to delay transactions that do not yield the desired outcome. This issue is particularly relevant for transactions involving bigger amounts of money (more than the staking reward). It is potentially predictable and generally not recommended.

Solution with Oracles

Oracles are typically used to access external data from outside the blockchain. Additionally, oracles can provide verifiable random numbers. However, they come with limitations such as high gas costs and dependency on third parties (the oracle owner can manipulate the data). We will learn more about oracles in future exercises.

Commit and Reveal Scheme

Another solution might be commit and reveal scheme. The proccess involves two steps:

1. **Commit**: Users (more than one) hash their random number concatenated with a secret value. They commit to this number by publishing the hash to the smart contract.

2. **Reveal**: Users reveal their random number and the secret value. The smart contract verifies the hash and calculates the random number from these commits:

$$\left(\sum_{i=1}^{n} r_i\right) \bmod N$$

   where r_i is the random number from user i and N is the number of users.

A potential limitation is that it requires user interaction, and users can withhold their reveals.

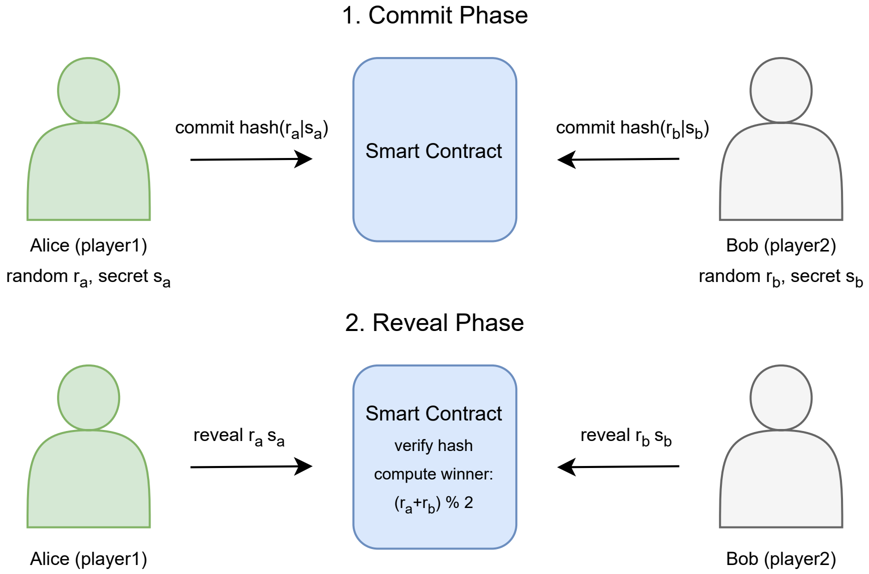

Figure 2: Commit and reveal scheme

For this vault, you will need a proxy contract as well. Implement your solution in the `contracts/AttackVault04.sol` file.

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

contract Vault04 {
    address public lastSolver;

    function breachVault(uint256 guess) public returns (bool) {
        require(
            guess ==
                uint256(
                    keccak256(
                        abi.encodePacked(
                            blockhash(block.number - 1),
                            block.timestamp
                        )
                    )
                ) % 100,
            "Incorrect guess"
        );
        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;
    uint256 private guess;
    uint256 private lockInBlockNumber;

    function lockInGuess(uint256 _guess) public {
        guess = _guess;
        lockInBlockNumber = block.number;
    }

    function breachVault() public returns (bool) {
        require(block.number > lockInBlockNumber, "Wait for the next block");
        require(
            guess == uint256(blockhash(lockInBlockNumber)) % 100,
            "Incorrect guess"
        );
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault05 in Remix IDE

Vault06: Explorer

EVM can store data in different areas: storage, transient storage, memory, and stack.

- **Storage** is persistent between function calls and transactions. It is the most expensive type of memory in terms of gas cost. We can think of storage as a hard drive. State variables are saved in storage by default.

- **Transient Storage** is similar to storage, but the main difference is that it resets at the end of each transaction. The value stored in this data persists only across function calls originating from the first call of the transaction. The cost is significantly lower than storage.
- **Memory** is a temporary storage location for data. A contract obtains a freshly cleaned instance for each message call. Once execution is completed, memory is cleared for the next execution. It is comparable to RAM.
- **Stack** EVM is a stack machine rather than a register machine; all computations are done in the data region called the stack. 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

Vault 08: Tokens for Free

In older versions of Solidity (prior to 0.8.0), there was no built-in protection against integer overflow and underflow. This led to many security vulnerabilities in smart contracts, where numeric calculations could wrap around in unexpected ways:

- **Integer Overflow**: Occurs when a number exceeds its maximum value and wraps around to its minimum value. For example, incrementing a `uint8` with value 255 by 1 would result in 0, not 256.
- **Integer Underflow**: Occurs when a number goes below its minimum value and wraps around to its maximum value. For example, decrementing a `uint8` with value 0 by 1 would result in 255, not -1.

For a `uint256` type:

- Maximum value: $2^{256} - 1$ (a number with 78 digits)
- If it exceeds this maximum → wraps around to 0
- If decremented below 0 → wraps around to $2^{256} - 1$

Starting with Solidity 0.8.0, arithmetic operations automatically include overflow and underflow checks, which will cause transactions to revert if they would result in overflow or underflow. However, if you're working with contracts using older Solidity versions, or if you specifically need to bypass these checks in newer versions, you can use the `unchecked` keyword:

```
    // In Solidity 0.8.0+
    function unsafeAdd(uint256 a, uint256 b) public pure returns (uint256) {
        // This block disables overflow/underflow checks
        unchecked {
            return a + b; // Can overflow without reverting
        }
    }
```

The `unchecked` keyword tells the compiler to skip the overflow/underflow checks for the code inside the block, which can save gas but potentially introduces vulnerabilities if not used carefully.

This challenge involves a token vault running on Solidity 0.7.6. Your goal is to obtain 1,000,000 tokens without paying the required amount of ETH. The price of each token is 1 ETH and you (player) start only with 1 ETH. Implement your solution in `test/Vault08.js`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

contract Vault08 {
    address public lastSolver;
    mapping(address => uint256) public tokenBalances;
    uint256 public constant TOKEN_PRICE = 1 ether;

    function buyTokens(uint256 numTokens) public payable {
        require(msg.value == numTokens * TOKEN_PRICE, "Incorrect payment amount");
        tokenBalances[msg.sender] += numTokens;
    }

    function breachVault() public returns (bool) {
        require(tokenBalances[msg.sender] >= 1_000_000,
        "You don't have enough tokens");
        lastSolver = tx.origin;
        return true;
    }

    // Allow the contract to receive ether
    receive() external payable {}
}
```

Vault08 in Remix IDE

Verify your solution with:

```
$ npm run vault08
```

Vault09: Less Is More

For this challenge, you'll need to gain more than 1,000,000 tokens. The vault is initialized with 1,000,000 tokens given to the deployer, and you start with just 1 token. You'll need to implement your attack in contracts/Vault09Attack.sol and then use the attack contract from test/Vault09.js. This time you are allowed to code your solution also in the designated part of the test file.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

contract Vault09 {
    address public lastSolver;
    mapping(address => uint256) public tokenBalances;
    mapping(address => mapping(address => uint256)) public allowances;

    constructor() {
        tokenBalances[msg.sender] = 1_000_000;
    }

    function approve(address spender, uint256 amount) external {
        allowances[msg.sender][spender] = amount;
    }

    function transferFrom(address from, address to, uint256 amount) external {
        require(tokenBalances[msg.sender] >= amount, "Not enough tokens");
        require(allowances[from][to] >= amount, "Not approved");
        allowances[from][to] -= amount;
        tokenBalances[from] -= amount;
        tokenBalances[to] += amount;
    }

    function breachVault() external returns (bool) {
        require(tokenBalances[msg.sender] >= 1_000_000, "You need at least 1,000,000
    tokens");
        lastSolver = tx.origin;
        return true;
    }
}
```

Vault09 in Remix IDE

Verify your solution with:

```
$ npm run vault09
```