CS577: Introduction to Blockchain and Cryptocurrency

End-Term Examination

Name: M Maheeth Reddy	**Roll No**: 1801CS31	**Date**: 26-Nov-2021

Q1. Explain re-entrancy attack in smart contract using a suitable solidity code example. [4 points]

Ans 1:

Re-Entrancy Attack

Smart Contracts are essentially computer programs, but they differ from conventional programs because they are stored on a blockchain and are automatically executed when certain conditions are met. Therefore, it should not be a surprise that Smart Contracts too, suffer from vulnerabilities.

One of the vulnerabilities that Smart Contracts suffer from is the Re-entrancy problem. This vulnerability was taken advantage in the infamous DAO Attack on the Ethereum Blockchain. The attack happened on 17th June 2016 and around 3.6 million Ether had been stolen by the attackers (equivalent to $50 million at the time) through recursive calls to the DAO smart contract.

Meaning of Re-entrant

A procedure is said to be **re-entrant** if it can be re-initiated during the middle of its execution and both runs can be allowed complete without any errors in execution. Such procedures can lead to serious vulnerability attacks in a Smart Contract ecosystem.

Explanation of Re-entrant Attack using the DAO Example

The DAO had a crowdfunding campaign and when they received sufficient funding, the DAO Smart Contract had a provision for the community members to withdraw ether from the contract via its Ethereum address. The vulnerability in the contract that made the attack possible was in the ether transfer mechanism.

The protocol was, the ether would be sent to the address of the receiver before the balance was updated in the contract's internal state. So, the attackers used re-entrancy to withdraw more ether than they were eligible.

The attackers mounted the attack on the DAO through the **Fallback Function**. Every Ethereum Smart Contract consists of this function and it is called when a non-existent function is called on the contract. This function can be overridden as per the choice of the developer.

```
// DAO.sol
contract DAO {
   // assign Ethers to an address
   mapping(address => uint256) public deposit;

   // credit an amount to sender's account
   function credit(address to) payable {
      deposit[msg.sender] += msg.value;
   }

   // get credited amount
   function getCreditedAmount(address)
       returns (uint) {
      return deposit[msg.sender];
   }

   // withdraw fund from contract
   function withdraw(uint amount) {
      if (deposit[msg.sender] >= amount) {
      msg.sender.call.value(amount)();
      deposit[msg.sender] -= amount; }
   }
}
```

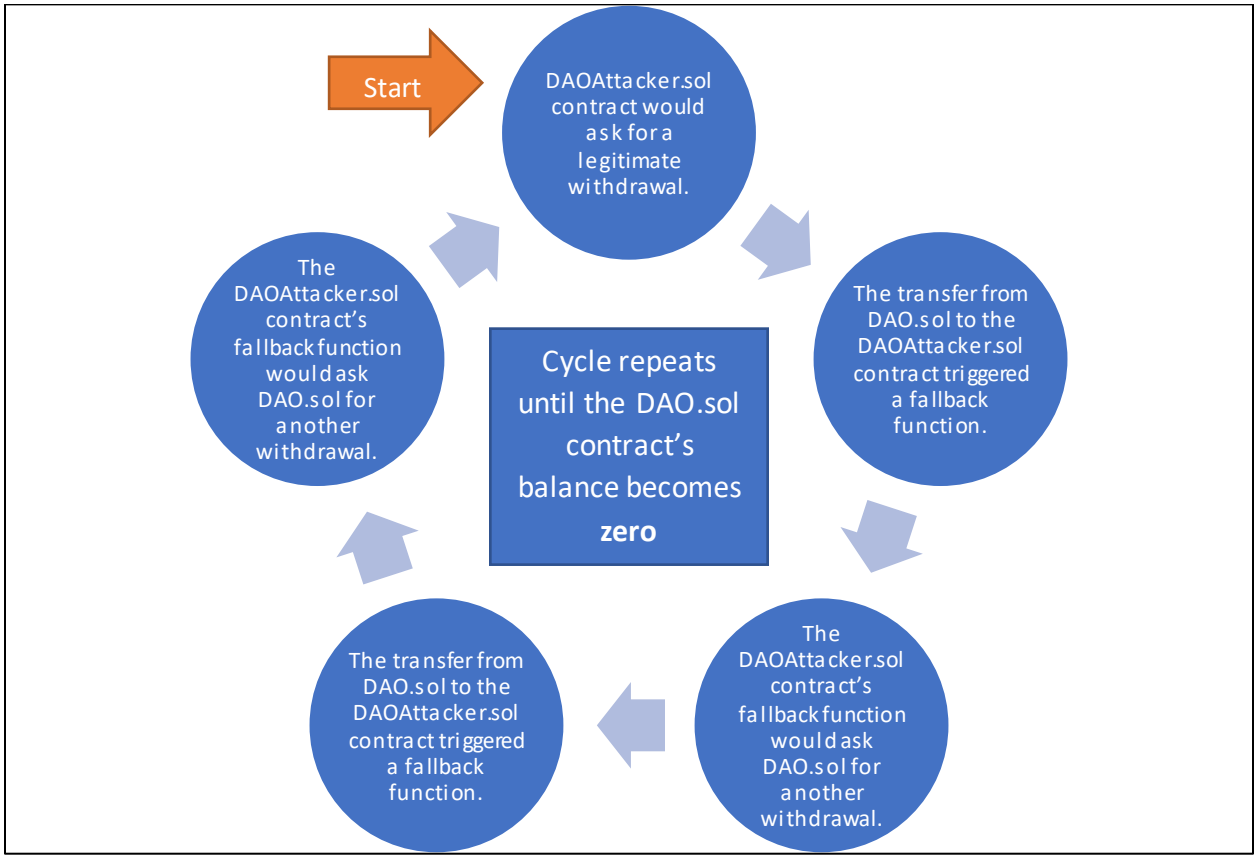
```
// DAOAttacker.sol
import 'DAO.sol';
contract DAOAttacker {

   // initialize DAO contract instance
   DAO public dao = DAO(0xDa32C9e....);
   address owner;

   //set contract creator as owner
   constructor(DaoAttacker) public {
      owner = msg.sender;
   }
   //fallback function calls widthraw function
   function() public {
      dao.withdraw(dao.getCreditedAmount(this));
   }

   /*send stolen funds to attacker's address*/
   function stealFunds() payable public{
      owner.transfer(address(this).balance);
   }
}
```

In the case of the DAO smart contract (a basic version is given above) the ether was transferred using the call.value() method (see DAO.sol, **line 19**). This allowed the transfer to use the maximum possible gas limit and also prevented reverting the state upon possible exceptions. Thus, the attackers were able to create a sequence of recursive calls which helped them steal funds from the DAO using a smart contract similar to the one shown above **DAOAttacker.sol**. The control flow of smart contract execution has been mentioned below.



The balance of the DAOAttacker.sol contract is never updated because that is supposed to happen after the transfer. Unless the transfer to the proxy contract fails, an exception is never thrown and the state never gets reverted.

Ways of Preventing Re-Entrancy Attack

1. The functions send() or transfer() can be used for ether transfer instead of call.value(). **Reason**: These functions do not allow for recursive withdrawal calls because they allow lower amount gas to be spent.
2. If the amount of gas passed to call.value() can be limited manually, attack can be prevented.
3. If the DAO contract updates the user balance before the ether transfer, any recursive calls will try to transfer a balance of 0 ether. A method is safe from re-entrancy if no internal state updates happen after an ether transfer or an external function call inside a method.

Q2. Given a smart contract that is vulnerable to re-entrancy attack. Suggest a possible solution to detect its presence in the smart contract. [6 marks]

Ans 2:

Possible solutions to detecting the presence of re-entrancy attack vulnerability in smart contracts

Monitoring only 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 deposit will act as a state variable which controls the decision to make a transaction or not. So, after the call we can lock the deposit variable to avoid the misuse of re-entrancy vulnerability. Thus, the attacker can be prevented from sending more ether to themselves.

Preventing the attack if the contract is yet to be deployed

All state changes must be ensured to happen before calling external contracts. Contract programmers should change the storage variable used as condition, before making any send() or call() to the attacker contract.

```
contract ReEntrancyGuard {
  bool internal locked;

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

One more way is to apply modifiers like the one mentioned below to any function that can be called externally from other contracts and affect the storage variables, to prevent re-entrancy. In this way, we can limit the extent of the attack to only occur once.

Preventing the attack if the contract is deployed

The code of contract deployed in the blockchain cannot be changed because of the blockchain's immutability. So, the course of action will be scenario specific i.e., it will largely depend on the effect of vulnerability. **Eg**: To counter the infamous DAO Attack, a hard fork was done due to the scale of possible misuse.

Q3. Propose a solution sketch (using schematic diagram and its brief description) to demonstrate the use of blockchain technology to establish a secure & trustful mechanism for storing Aadhaar-details and Aadhaar-based user-identity validation. [10 marks]

Ans 3:

I have proposed a solution to use blockchain technology to establish a secure and trustful mechanism for storing Aadhaar details and Aadhaar based user identity validation. I am calling it as BIAS: **Blockchain** based **Identity** validation for **Aadhaar Scheme**.

The BIAS Blockchain will be a **public blockchain**. It will comprise of **hashed user data** only, **accessible only to the corresponding users, the government agencies and verified third parties who require Aadhaar data**.

Initially, a smart contract must be mined in the BIAS Blockchain to store the details of various users. The contract would consist of Hashed User Data stored as mapping with Aadhaar Number as key. User Data could be added, modified or fetched via necessary getter and setter functions.

Step 1: **User Registration** (Schematic on the right)

First of all, for a person's to be identity to be verified, their details must be stored on the Aadhaar blockchain network. So, they need to go through a registration process first.

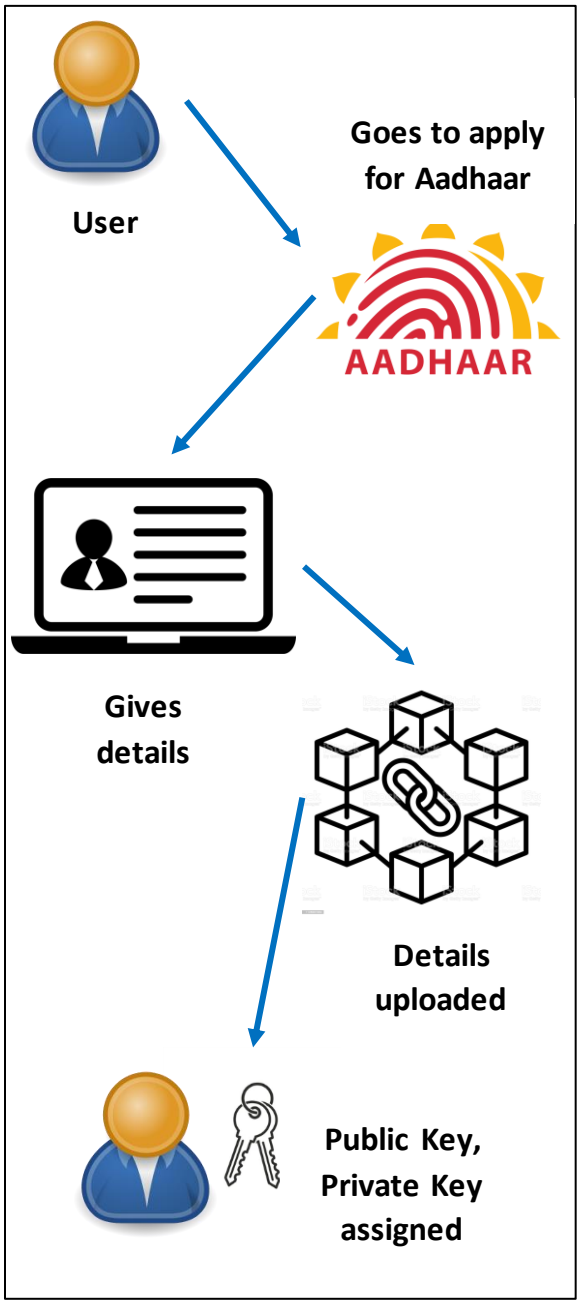
In general, the process begins with a person going to an Aadhaar Enrolment Centre.

At the center important personal information like Name, Date of Birth, Sex, Contact Details, Address etc., and biological identification information like his Fingerprint and Iris Scan will be taken.

Next, this **user information will be hashed** and uploaded to the BIAS blockchain system from the computer at the enrollment center. This uploading of user details could be considered a **miner node** in the network, because it is responsible for adding blocks of user data into the BIAS blockchain.

The **person will be added to the blockchain** as a light node. They will be assigned an **Aadhaar Identification Number, which acts as a public key** on the BIAS blockchain **and a private key**. **The two keys will help them access their information**.

The user info would have been signed with their Aadhaar number and the signature generated is mapped to that number.



Step 2: User Verification

Consider the following scenario:

The government has released subsidies to the bank against a person's Aadhaar Number. Now the person has to be verified to get that money.

This is how Blockchain will be involved in the verification process:

- The person will be updated about the release.
- The person will place a request in the bank to release the money given to the bank by the Government.
- **The bank will ask the person for his Aadhaar Number, along with some personal details he filled in the Aadhaar Enrolment Center.**
- The person will give the user info, as per his/her Aadhaar to the bank.
- **The bank will now encrypt the data given by the person with his Aadhaar Number i.e., the public key of the BIAS blockchain.**
- The bank will verify the person's details by **fetching the hashes of relevant data from BIAS blockchain** using the person's Aadhaar ID.
- Once the **person's details are verified by the bank**, they can release the subsidy to the person.