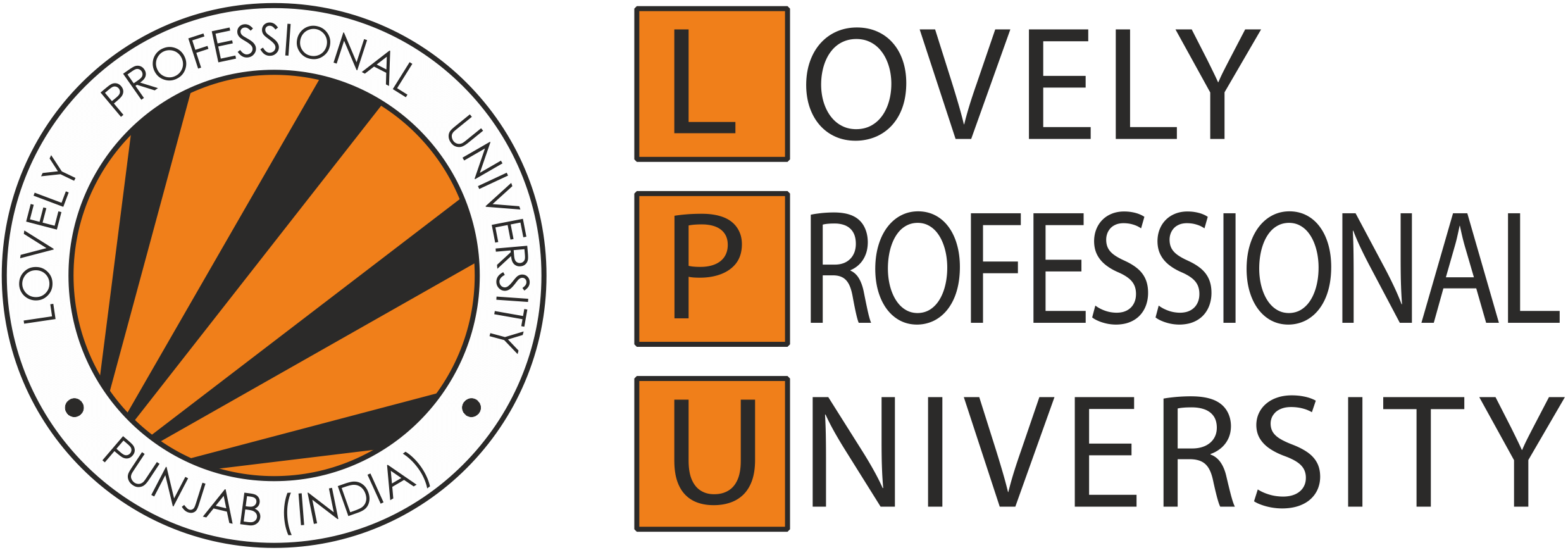
**CSC307 - Blockchain ARCHITECTURE AND DESIGN**

**CA-II**



**Submitted By Submitted To:**

**Name:** Nasolla soumyasiri **Piyush Gururani**

**Reg no.:** 12213846

**Section:** K22cs

**Roll no:** 12

**Topic:** question 2

1. **Introduction**   
   A programming language called Solidity is intended for use in the development of Ethereum blockchain smart contracts. These contracts frequently carry out crucial corporate functions and manage priceless assets. In order to reduce the risk, we will examine a particular Solidity contract, pinpoint a significant security flaw, describe how it could be used, and offer a safe fix.
2. **Overview**

The Solidity contract that is offered for payment is made to manage users' ether deposits and withdrawals.

1. **Contract:**

pragma solidity ^0.8.0;

contract Payment {

mapping(address => uint) public balances;

function deposit() public payable {

balances[msg.sender] += msg.value;

}

function withdraw(uint amount) public {

require(balances[msg.sender] >= amount, "Insufficient balance");

(bool sent, ) = msg.sender.call{value: amount}("");

require(sent, "Failed to send Ether");

balances[msg.sender] -= amount;

}

}

1. **Identification of Vulnerabilities**

There is a serious re-entrancy vulnerability in the contract's withdraw function.   
The problem is with the order of operations:

* + - 1. The user receives ether before their balance is updated;
      2. A malicious actor could take advantage of this order to spend the contract's funds.

1. **Situations of Exploitation**   
   An attacker can construct a hostile contract that communicates with the Payment contract in order to take advantage of this vulnerability.
2. **Attack Execution Steps**
   * + 1. Deploying: The attacker or any user deploys the vulnerable Payment contract, which contains a reentrancy flaw in its withdraw function.
       2. Deploying a Malicious Contract: To take advantage of the Payment contract's reentrancy flaw, the attacker crafts and installs a malicious contract. A fallback function to rejoin the withdraw function is included in this contract.
       3. Funding the Payment Contract: To make sure there is sufficient money in the Payment contract for the attack, the attacker uses the malicious contract to pay Ether to the susceptible Payment contract.
       4. Starting the Withdrawal: Using the malicious contract, the attacker starts the Payment contract's withdraw feature. This initiates the process of withdrawal.
       5. Execute the Reentrancy Attack: The malicious contract's fallback function is activated while the Payment contract is moving Ether to the malicious contract. Before the susceptible contract has updated its balance, the fallback function makes another call to the withdraw method. The money from the Payment contract is depleted as a result of the repeated use of this procedure.
3. **Secure Solution:** We may use the checks-effects-interactions pattern to stop reentrancy attacks. This pattern makes sure that before any external calls are made, the contract's state is changed.

Revised withdraw function:

pragma solidity ^0.8.0;

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

function deposit() public payable {

balances[msg.sender] += msg.value;

}

function withdraw(uint amount) public {

require(balances[msg.sender] >= amount, "Insufficient balance")

balances[msg.sender] -= amount;

(bool sent, ) = msg.sender.call{value: amount}("");

require(sent, "Failed to send Ether");

}

}

Verify that the user has enough money in their account to withdraw the designated sum.   
• **Impact:** Prior to delivering Ethereum, update the status by lowering the user's balance.   
• **Interaction**: After the state has been modified, give the user the ether.   
Reentrancy attacks are avoided by updating the balance before to transferring Ether since the external call takes place after the state change.

**conclusion**   
Due to the order of activities, the Payment contract's original withdraw function was susceptible to reentrancy attacks.   
We secured the function by adhering to the checks-effects-interactions design, which makes sure that the state of the contract is modified prior to any ether being transmitted.   
Maintaining security and avoiding vulnerabilities requires adhering to standard practices while developing smart contracts.