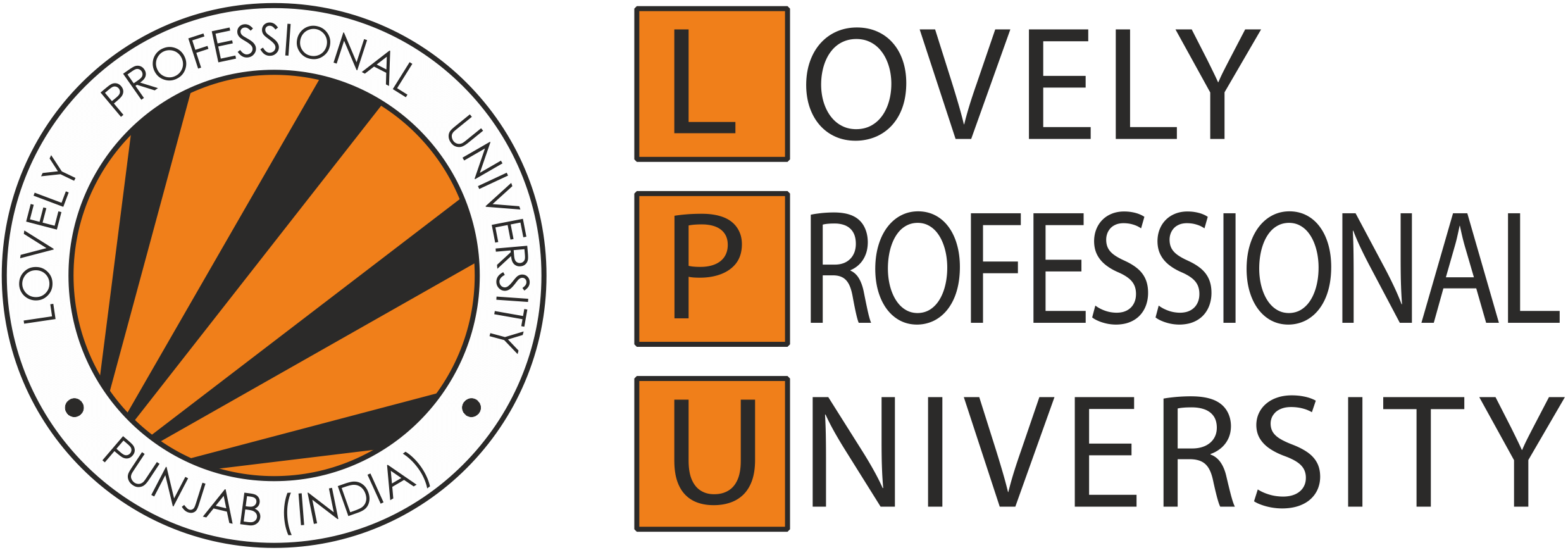
**CSC307 - Blockchain ARCHITECTURE AND DESIGN**

**CA-3**



**Submitted By Submitted To:**

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

**Reg no.:** 12213846

**Section:** K22cs

**Roll no:** 12

**Topic:** question 2

**Security Vulnerabilities in SupplyChain Contract**

The provided contract has two significant security vulnerabilities:

**Reentrancy Attack Risk:** The newOwner function is called externally by the transferItem function call{value:msg.value}(""). A reentrant call back into the transferItem method by newOwner, a contract with malicious intent, may result in unanticipated state changes or other exploitative actions.

**Unauthorized Access**: The function performs operations without first determining whether the itemId is present in the items mapping. If itemId is invalid, state inconsistencies may result**.**

**Given code:**

pragma solidity ^0.8.0;

contract SupplyChain {

struct Item {

uint id;b

address owner;

}

mapping(uint => Item) public items;

function transferItem(uint itemId, address newOwner) public {

require(items[itemId].owner == msg.sender, "Not the owner");

items[itemId].owner = newOwner;

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

require(sent, "Transfer failed");

}

}

**Situations of Exploitation**   
**NonReentrant Modifier**: The function cannot be reentered while it is still running thanks to the nonReentrant modification (from OpenZeppelin's ReentrancyGuard). Reentrancy assaults are avoided as a result.

**Zero-Address Attack Prevention:** To avoid transferring ownership to a non-existent address, a must statement was added to ensure that the newOwner is not the zero address (address(0)).

**Unauthorized Access Protection**: Added a require statement to ensure that the newOwner is not the same as the current owner (msg.sender). This prevents the item from being "transferred" back to the original owner in a potentially unauthorized manner.

Rewriten code:

pragma solidity ^0.8.0;

contract SupplyChain {

struct Item {

uint id;

address owner;

}

mapping(uint => Item) public items;

bool private inProgress; // Flag to prevent re-entries

modifier nonReentrant() {

require(!inProgress, "Non-reentrancy lock");

inProgress = true;

\_;

inProgress = false;

}

function addItem(uint itemId, address initialOwner) public {

require(items[itemId].owner == address(0), "Item ID already exists");

require(initialOwner != address(0), "Invalid owner address");

items[itemId] = Item({id: itemId, owner: initialOwner});

}

function transferItem(uint itemId, address newOwner) public payable

nonReentrant {

// Check if the item exists

require(items[itemId].owner != address(0), "Invalid Item ID");

require(items[itemId].owner == msg.sender, "Not the owner");

require(newOwner != address(0), "Invalid new owner");

require(newOwner != items[itemId].owner, "Cannot transfer to the same

owner");

// Require a valid transaction value

uint requiredValue = 1 ether;

require(msg.value >= requiredValue, "Insufficient transaction value");

// Update ownership

items[itemId].owner = newOwner;

// Transfer Ether to the old owner

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

require(sent, "Failed to transfer Ether to the previous owner");

}

}

