**Q. contract DWG {**

**mapping(address => uint256) private balances;**

**function deposit() public payable {**

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

**}**

**function withdraw(uint256 amount) public {**

**uint256 balance = balances[msg.sender];**

**balances[msg.sender] -= amount;**

**payable(msg.sender).transfer(amount);**

**}**

**function getBalance() public view returns (uint256) {**

**return balances[msg.sender];**

**}**

**}**

**Identify the issues in the smart contract and fix the issue that you identify. Explain the core reason why the issue happens.**

**Ans.)**

**1. Vulnerability Description: Reentrancy Vulnerability**

This vulnerability is located in the withdraw function. The primary concern here is that balances[msg.sender] -= amount; is executed before transferring amount to msg.sender. Now an attacker can exploit this by entering the withdraw function once again before updates occur in balances[msg.sender]; hence they will be able to withdraw more funds than they deposited.

Example of Attack

The attacker can make a contract call withdraw and reenter withdraw several times before the balance is updated, which will cause them to withdraw more than they initially put into the contract.

How the Bug Occurs

The bug occurs since the external call payable(msg.sender).transfer(amount); occurs before the contract has a chance to update the user's balance. This means an external contract can make a reentrant call and end by draining funds.

How to Fix the Bug

Hence, to fix this, we would need to do the following:

• We need to update the balance of the user before actually transferring funds so that there is no chance of reentrancy.

• Optionally, we use the Checks-Effects-Interactions pattern, which is also known as a best practice in Solidity to avoid this kind of vulnerability.

**pragma solidity ^0.8.0;**

**contract DWG {**

**mapping(address => uint256) private balances;**

**function deposit() public payable {**

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

**}**

**function withdraw(uint256 amount) public {**

**uint256 balance = balances[msg.sender];**

**require(balance >= amount, "Insufficient balance");**

**balances[msg.sender] -= amount;**

**payable(msg.sender).transfer(amount);**

**}**

**function getBalance() public view returns (uint256) {**

**return balances[msg.sender];**

**}**

**}**

**2. Changes Explanation**

1. require statement: Insert a require statement: require balance >= amount, "Insufficient balance"; That statement checks whether the user has sufficient balance in his account to withdraw money.

2. Order of Operations: Update balances[msg.sender] -= amount; before calling the external call with payable(msg.sender).transfer(amount); Thus, even in case of reentrant call, balance is already deducted not to withdraw repeated times.

**3. Reason at the Core for the Problem**

The root cause of the vulnerability is making an external call-for example, transferring Ether-before updating the internal state. A malicious user may trigger the function to be called repeatedly; therefore, they can spend funds from the contract. It is something that the checks-effects-interactions pattern keeps away from you.