Experiment No-7

Title: Security Analysis of Smart Contracts (Detecting and Preventing Re-entrancy Attacks)

Aim: To study, detect, and prevent re-entrancy attacks in Ethereum smart contracts using Solidity.

Theory:

- **Smart Contracts** are self-executing programs stored on the blockchain.
- **Re-entrancy Attack** happens when a contract calls an external contract before updating its own state.
    - An attacker repeatedly calls the function before the state is updated, draining funds.
- **Example:** A withdrawal function sends ETH before updating the balance → attacker re-calls `withdraw()` multiple times.

Famous Example: The DAO hack (2016) where $60M was stolen due to a re-entrancy bug.

Prevention Techniques:

1. **Checks-Effects-Interactions Pattern** – Update state first, then send funds.
2. **re-entrancy Guard** – Use `nonpenetrant` modifier (Open Zeppelin).
3. **Limit external calls** – Avoid calling untrusted contracts directly.

Key Characteristics of dApps :

- **Occurs during external calls** (e.g., `call()`, `send()`, `transfer()`).
- **Attacker drains funds** by recursive function calls.
- **State inconsistency** – Balance not updated before transfer.
- **High severity** – Can cause total loss of funds.

Steps of Execution:

1. Write a **vulnerable smart contract**.
2. Write an **attacker contract** to exploit it.
3. Deploy both contracts on Ethereum Testnet (or Remix VM).
4. Execute attack → observe funds drained.
5. Apply **fixes** (Checks-Effects-Interactions or ReentrancyGuard).
6. Re-test to confirm attack prevention.

Stepwise Procedure :

Part A – Vulnerable Contract

1. Open **Remix IDE**.
2. Create `VulnerableBank.sol`.
3. Write withdrawal function that sends ETH before updating balance.
4. Deploy contract and deposit some ETH.

Part B – Attacker Contract

1. Create `Attacker.sol`.
2. Write a fallback function to repeatedly call `withdraw()`.
3. Deploy attacker contract and attack the bank.

Part C – Fix

1. Modify withdrawal function to update balance **before** sending ETH.
2. Or use OpenZeppelin's `ReentrancyGuard`.
3. Redeploy and test again → attack should fail.

Program Vulnerable Bank Contract :

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract VulnerableBank {

    mapping(address => uint) public balances;

    function deposit() public payable {

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

    }

    function withdraw(uint _amount) public {

        require(balances[msg.sender] >= _amount, "Not enough balance");

        // ❌ Vulnerable: Sending ETH before updating balance

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

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

        balances[msg.sender] -= _amount;

    }

    function getBalance() public view returns (uint) {

        return balances[msg.sender];

    }

}
```

Fixed Bank Contract (Safe):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SafeBank is ReentrancyGuard {
    mapping(address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint _amount) public nonReentrant {
        require(balances[msg.sender] >= _amount, "Not enough balance");

        // ✅ Effect before interaction
        balances[msg.sender] -= _amount;

        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }
    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }
}
```

Key Points :

- Reentrancy occurs when contracts make **external calls before updating state**.
- The DAO hack is the biggest example of this vulnerability.
- Always use **Checks-Effects-Interactions** or **ReentrancyGuard**.
- Never trust external contract calls.

Conclusion :

In this experiment, we studied how a re-entrancy attack works in Ethereum smart contracts. We implemented a vulnerable contract, exploited it using an attacker contract, and then applied prevention techniques. This experiment highlights the importance of secure coding practices in blockchain.

Viva Questions:

1. What is a reentrancy attack in Ethereum?
2. How did the DAO hack exploit reentrancy?
3. Why is call() risky compared to transfer()?
4. Explain the **Checks-Effects-Interactions** pattern.
5. What is the role of OpenZeppelin's ReentrancyGuard?
6. How can you detect a reentrancy bug before deployment?
7. Why is security more critical in smart contracts than in normal applications?