MultisigWallet

purpose:

This contract implements a Multi-signature Wallet that allows a group of owners to collectively manage and authorize transactions. A transaction can only be executed when a specified number of owners approve it.

Functionalities of MultisigContract:

1.Owners: Only designated owners can submit, confirm, execute, or revoke transactions.

2.Submit Transaction: An owner proposes a transaction.

3.Confirm Transaction: Other owners must confirm the transaction.

4.Execute Transaction: Once the required number of confirmations is met, the transaction can be executed.

5.Revoke Confirmation: An owner can revoke their confirmation before execution.

6.Deposit: Ether can be deposited into the wallet.

About the Contract::

This is a simple Multi-signature contract that allows a set of designated owners to approve transactions before they can be executed.

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

contract MultisigWallet {
    address[] public owners; // Owners of the multisig wallet who can approve transactions

    mapping(address => bool) public isOwner;
    uint public required;

    struct Transaction {
        address to;
        uint value;
        bytes data;
        bool executed;
        uint numConfirmations;
    }

    Transaction[] public transactions;

    mapping(uint => mapping(address => bool)) public isConfirmed;
```

```
// --- Events ---
event Deposit(address indexed sender, uint amount);
event Submit(
    uint indexed txIndex,
    address indexed to,
    uint value,
    bytes data
);
event Confirm(uint indexed txIndex, address indexed owner);
event Execute(uint indexed txIndex);
event Revoke(uint indexed txIndex, address indexed owner);

// --- Modifiers ---
modifier onlyOwner() {
    require(isOwner[msg.sender], "Not owner");
    _;
}

modifier txExists(uint _txIndex) {
    require(_txIndex < transactions.length, "Tx doesn't exist");
    _;
}

modifier notExecuted(uint _txIndex) {
    require(!transactions[_txIndex].executed, "Already executed");
    _;
}

modifier notConfirmed(uint _txIndex) {
    require(!isConfirmed[_txIndex][msg.sender], "Already confirmed");
    _;
}

// --- Constructor ---
constructor(address[] memory _owners, uint _required) {
    require(_owners.length > 0, "Owners required");
    require(
        _required > 0 && _required <= _owners.length,
        "Invalid required"
    ); // Sets the number of required confirmations for executing a transaction


    for (uint i; i < _owners.length; i++) {
        address owner = _owners[i];
        //@nkdoubt://what is the use of zero address in ethereum and significance of zer
        require(owner != address(0), "Invalid owner");
```

```
        require(!isOwner[owner], "Owner not unique");
        isOwner[owner] = true;
        owners.push(owner);
    }

    required = _required;
}

// --- Functions ---

receive() external payable {
    emit Deposit(msg.sender, msg.value); // When anyone deposits Ether into the contract

}

function submitTransaction(
    address _to,
    uint _value,
    bytes memory _data
) public onlyOwner {
    uint txIndex = transactions.length;

    transactions.push(
        Transaction({
            to: _to,
            value: _value,
            data: _data,
            executed: false,
            numConfirmations: 0
        })
    );

    emit Submit(txIndex, _to, _value, _data);
}

function confirmTransaction(
    uint _txIndex
)
    public
    onlyOwner
    txExists(_txIndex)
    notExecuted(_txIndex)
    notConfirmed(_txIndex)
{
    Transaction storage transaction = transactions[_txIndex];
    transaction.numConfirmations += 1;
```

```
    isConfirmed[_txIndex][msg.sender] = true;

    emit Confirm(_txIndex, msg.sender);
}

function executeTransaction(
    uint _txIndex
) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) {
    Transaction storage transaction = transactions[_txIndex];
    require(
        transaction.numConfirmations >= required,
        "Not enough confirmations"
    );

    transaction.executed = true;

    (bool success, ) = transaction.to.call{value: transaction.value}(
        transaction.data
    );
    require(success, "Tx failed");

    emit Execute(_txIndex);
}

function revokeConfirmation(
    uint _txIndex
) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) {
    require(isConfirmed[_txIndex][msg.sender], "Tx not confirmed");

    Transaction storage transaction = transactions[_txIndex];
    transaction.numConfirmations -= 1;
    isConfirmed[_txIndex][msg.sender] = false;

    emit Revoke(_txIndex, msg.sender);
}

// Utility functions
function getOwners() public view returns (address[] memory) {
    return owners;
}

function getTransactionCount() public view returns (uint) {
    return transactions.length;
}

function getTransaction(
```

```
        uint _txIndex
    )
        public
        view
        returns (
            address to,
            uint value,
            bytes memory data,
            bool executed,
            uint numConfirmations
        )
    {
        Transaction storage transaction = transactions[_txIndex];
        return (
            transaction.to,
            transaction.value,
            transaction.data,
            transaction.executed,
            transaction.numConfirmations
        );
    }
}
```

Vulnerability Scenario: If an attacker is one of the owners, they may try to:

1.Submit malicious transactions (e.g., to their own wallet).

2.Revoke confirmations repeatedly to prevent quorum.

3.Spam the contract with transactions to exhaust gas or memory.

4.Trigger reentrancy attacks if executeTransaction() is not protected.

5.Abuse revokeConfirmation() to manipulate the confirmation state rapidly.

Fix the contract:

nonReentrant Modifier: Prevents reentrancy attacks during transaction execution.

require(_to != address(this)): Prevents self-calls that could manipulate internal state or reenter functions.

revokeCooldown using lastRevokeTime : Imposes a slowdown (e.g., 30 seconds) between consecutive revokes by the same owner.

max pending transactions : Prevents spamming of the contract by limiting the total number of unexecuted transactions.

require(unique owners) : Enforces that all owners are valid and unique, rejecting zero-address or duplicates.

Updated contract:

After all these udpates ,the contract looks like:

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

contract MultisigWallet {
    address[] public owners;
    mapping(address => bool) public isOwner;
    uint public required;

    struct Transaction {
        address to;
        uint value;
        bytes data;
        bool executed;
        uint numConfirmations;
        uint timestamp;
    }

    Transaction[] public transactions;
    mapping(uint => mapping(address => bool)) public isConfirmed;
    mapping(address => uint) public lastRevokeTime;

    bool private locked;// Used to prevent reentrancy attacks


    // --- Events ---
    event Deposit(address indexed sender, uint amount);
    event Submit(
        uint indexed txIndex,
        address indexed to,
        uint value,
        bytes data
    );
    event Confirm(uint indexed txIndex, address indexed owner);
    event Execute(uint indexed txIndex);
    event Revoke(uint indexed txIndex, address indexed owner);

    // --- Modifiers ---
    modifier onlyOwner() {
        require(isOwner[msg.sender], "Not owner");
        _;
    }
```

```
    modifier txExists(uint _txIndex) {
        require(_txIndex < transactions.length, "Tx doesn't exist");
        _;
    }

    modifier notExecuted(uint _txIndex) {
        require(!transactions[_txIndex].executed, "Already executed");
        _;
    }

    modifier notConfirmed(uint _txIndex) {
        require(!isConfirmed[_txIndex][msg.sender], "Already confirmed");
        _;
    }
//added for reentrancy
    modifier nonReentrant() {
        require(!locked, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }

    // --- Constructor ---
    constructor(address[] memory _owners, uint _required) {
        require(_owners.length > 0, "Owners required");
        require(
            _required > 0 && _required <= _owners.length,
            "Invalid required"
        );

        for (uint i; i < _owners.length; i++) {
            address owner = _owners[i];
            require(owner != address(0), "Invalid owner");// Prevents zero addresses or dupl
            require(!isOwner[owner], "Owner not unique");
            isOwner[owner] = true;
            owners.push(owner);
        }

        required = _required;
    }

    // --- Receive Ether ---
    receive() external payable {
        emit Deposit(msg.sender, msg.value);
    }
```

```
// --- Core Functions ---
function submitTransaction(
    address _to,
    uint _value,
    bytes memory _data
) public onlyOwner {
    require(_to != address(this), "Can't call self");// Prevents internal calls to the c
    require(transactions.length < 1000, "Too many pending transactions");// Limit the nu

    uint txIndex = transactions.length;

    transactions.push(
        Transaction({
            to: _to,
            value: _value,
            data: _data,
            executed: false,
            numConfirmations: 0,
            timestamp: block.timestamp
        })
    );

    emit Submit(txIndex, _to, _value, _data);
}

function confirmTransaction(
    uint _txIndex
)
    public
    onlyOwner
    txExists(_txIndex)
    notExecuted(_txIndex)
    notConfirmed(_txIndex)
{
    Transaction storage transaction = transactions[_txIndex];
    transaction.numConfirmations += 1;
    isConfirmed[_txIndex][msg.sender] = true;

    emit Confirm(_txIndex, msg.sender);
}

function executeTransaction(
    uint _txIndex
) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) nonReentrant {
    Transaction storage transaction = transactions[_txIndex];
```

```
        require(
            transaction.numConfirmations >= required,
            "Not enough confirmations"
        );

        transaction.executed = true;

        (bool success, ) = transaction.to.call{value: transaction.value}(
            transaction.data
        );
        require(success, "Tx failed");

        emit Execute(_txIndex);
    }

    function revokeConfirmation(
        uint _txIndex
    ) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) {
        require(isConfirmed[_txIndex][msg.sender], "Tx not confirmed");

        //Adds a 30-second delay before the same owner can revoke
        require(
            block.timestamp > lastRevokeTime[msg.sender] + 30,
            "Revoke cooldown"
        );
        lastRevokeTime[msg.sender] = block.timestamp;

        Transaction storage transaction = transactions[_txIndex];
        transaction.numConfirmations -= 1;
        isConfirmed[_txIndex][msg.sender] = false;

        emit Revoke(_txIndex, msg.sender);
    }

    // --- View Functions ---
    function getOwners() public view returns (address[] memory) {
        return owners;
    }

    function getTransactionCount() public view returns (uint) {
        return transactions.length;
    }

    function getTransaction(
        uint _txIndex
    )
```

```
        public
        view
        returns (
            address to,
            uint value,
            bytes memory data,
            bool executed,
            uint numConfirmations
        )
    {
        Transaction storage transaction = transactions[_txIndex];
        return (
            transaction.to,
            transaction.value,
            transaction.data,
            transaction.executed,
            transaction.numConfirmations
        );
    }
}
```

How it works:

Let's say the company wants to send 10 ETH to a freelancer for development work. But to ensure security and agreement, at least 2 directors must approve before the money is sent.

steps:

1.Deployment: Alice deploys the MultisigWallet contract with:

Owners: [Nithin, uday, rocky]

Required confirmations: 2 `solidity MultisigWallet([ Nithin, uday, rocky], 2)`

2.Fund the Wallet: Anyone (e.g., the company) sends ETH to the wallet.

`It emits a Deposit event.`

```solidity
send 20 ETH to MultisigWallet address
```

3. Submit a Transaction: Nithin wants to pay a freelancer.

he submits a transaction to send 10 ETH to Dev.

`submitTransaction(to: Dev, value: 10 ETH, data: "")`

This is recorded in the contract as a Transaction object:

Not yet executed.

Has 0 confirmations.

Submit event is emitted with txIndex = 0.

4.Confirm the Transaction: Uday agrees with the transaction and confirms it.

```
confirmTransaction(0)
```

The number of confirmations increases to 1.

Confirm event is emitted.

Rocky also confirms:

```
confirmTransaction(0)
```

Now it has 2 confirmations — which meets the required threshold.

Confirm event is emitted again.

5.Execute the Transaction: Anyone of the owners (say NITHIN again) can now call:

```
executeTransaction(0)
```

Contract checks:

Is confirmed by 2 owners?

Is it not already executed?

If checks pass, the 10 ETH is sent to the freelancer.

Execute event is emitted.

Summary:

Why use a Multisig Wallet? >Trustless collaboration: No single person has full control.

Security: Reduces the risk of fund theft even if one owner is compromised.

Transparency: On-chain confirmations and log