**Building a Multi-Signature Wallet Contract in Solidity**

[[](https://medium.com/@christianinyekaka?source=post_page-----63641eaacd2b--------------------------------)](https://medium.com/@christianinyekaka?source=post_page-----63641eaacd2b--------------------------------)

[[CoinsBench](https://coinsbench.com/?source=post_page-----63641eaacd2b--------------------------------)](https://coinsbench.com/?source=post_page-----63641eaacd2b--------------------------------)

[Christianinyekaka](https://medium.com/@christianinyekaka?source=post_page-----63641eaacd2b--------------------------------)

·

Follow

Published in

[CoinsBench](https://coinsbench.com/?source=post_page-----63641eaacd2b--------------------------------)

·

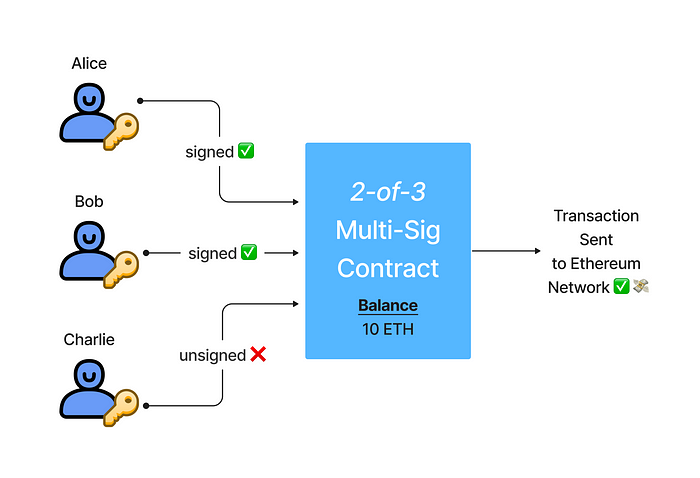
6 min read

·

Dec 16, 2023

22

2



**Introduction**

Blockchain transactions often involve risks associated with single-authority control. Multi-Signature Wallets offer a solution by requiring multiple approvals to execute a transaction, enhancing security and trust. In this guide, we’ll walk through creating a Multi-Signature Wallet contract using Solidity, Ethereum’s smart contract language.

**Understanding Multi-Signature Wallets**

Multi-Signature Wallets, or multi-sigs, necessitate several signatures to validate and execute a transaction. This setup mitigates the risk of single points of failure, making it ideal for joint accounts, company funds, or any scenario demanding heightened security.

**Knowledge and Skills**

To proceed with building this contract, the following skills and knowledge are necessary:

1. **Solidity Programming**: Understanding of Solidity, Ethereum’s smart contract language, is crucial.
2. **Blockchain Basics**: Familiarity with blockchain concepts, Ethereum, and how transactions work on the Ethereum network.
3. **Smart Contract Development Tools**: Proficiency in using development tools like Remix IDE, Truffle, or Hardhat for coding, testing, and deploying contracts.

**Contract Structure Overview**

// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.9;  
  
contract MultiSig {  
 address[] public owners;  
 uint public transactionCount;  
 uint public required;  
  
 struct Transaction {  
 address payable destination;  
 uint value;  
 bool executed;  
 bytes data;  
 }  
  
 mapping(uint => Transaction) public transactions;  
 mapping(uint => mapping(address => bool)) public confirmations;  
  
 receive() payable external {  
   
 }  
  
 function executeTransaction(uint transactionId) public {  
 require(isConfirmed(transactionId));  
 Transaction storage \_tx = transactions[transactionId];  
 (bool success, ) = \_tx.destination.call{ value: \_tx.value }(\_tx.data);  
 require(success);  
 \_tx.executed = true;  
 }  
  
 function isConfirmed(uint transactionId) public view returns(bool) {  
 return getConfirmationsCount(transactionId) >= required;  
 }  
  
 function getConfirmationsCount(uint transactionId) public view returns(uint) {  
 uint count;  
 for(uint i = 0; i < owners.length; i++) {  
 if(confirmations[transactionId][owners[i]]) {  
 count++;  
 }  
 }  
 return count;  
 }  
  
 function isOwner(address addr) private view returns(bool) {  
 for(uint i = 0; i < owners.length; i++) {  
 if(owners[i] == addr) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 function submitTransaction(address payable dest, uint value,bytes calldata data) external {  
 uint id = addTransaction(dest, value,data);  
 confirmTransaction(id);  
 }  
  
 function confirmTransaction(uint transactionId) public {  
 require(isOwner(msg.sender));  
 confirmations[transactionId][msg.sender] = true;  
 if(isConfirmed(transactionId)) {  
 executeTransaction(transactionId);  
 }  
 }  
  
 function addTransaction(address payable destination, uint value,bytes calldata data) public returns(uint) {  
 transactions[transactionCount] = Transaction(destination, value, false,data);  
 transactionCount += 1;  
 return transactionCount - 1;  
 }  
  
 constructor(address[] memory \_owners, uint \_confirmations) {  
 require(\_owners.length > 0);  
 require(\_confirmations > 0);  
 require(\_confirmations <= \_owners.length);  
 owners = \_owners;  
 required = \_confirmations;  
 }  
}

**Step-by-Step Guide**

**1. Contract Initialization**

The contract initializes the owners array and defines the required number of confirmations in the constructor. It ensures the deployment transaction reverts if:

* No owner addresses are sent.
* The number of required confirmations is zero.
* The number of required confirmations is more than the total number of owner addresses.

constructor(address[] memory \_owners, uint \_confirmations) {  
 require(\_owners.length > 0);  
 require(\_confirmations > 0);  
 require(\_confirmations <= \_owners.length);  
 owners = \_owners;  
 required = \_confirmations;  
 }

**2. Transaction Setup**

The Transaction struct includes member variables:

* Address for the destination of the transaction’s value.
* Uint256 value of the transaction in wei.
* Bool named executed indicating if the transaction has been executed.
* Bytes data the call data.

struct Transaction {  
 address payable destination;  
 uint value;  
 bool executed;  
 bytes data;  
 }

**4. Transaction Storage**

We’ll need to store the transactions while they are being confirmed by the other owners. we have *two options* for our implementation here:

1. Create a public mapping from a uint id to a Transaction
2. Create a public array of Transaction

Then, create a public transactionCount view function which returns the total number of transactions stored. In this tutorial, we shall use mapping

uint public transactionCount;  
mapping(uint => Transaction) public transactions;

*Note: we said that we were going to create a function transactionCount but what we did was just create a state variable and set its visibility to public.in solidity, the public view on a state variable will automatically create a getter function. this is what we are utilizing here.*

**4. Receiving Ether**

A multi-sig can receive funds, essential for organizational wallets. The receive() function ensures the contract can always receive funds.

receive() payable external {  
   
 }

**5. Adding Transactions**

A function is defined to add transactions to the storage, returning a unique transaction ID.

function addTransaction(address payable destination, uint value,bytes calldata data) public returns(uint) {  
 transactions[transactionCount] = Transaction(destination, value, false,data);  
 transactionCount += 1;  
 return transactionCount - 1;  
 }

*Note: the reason why we are subtracting 1 from the transactionCount when returned is that The transaction IDs should be zero-based. The first one being 0, then 1, 2, and so on.*

**6. Storage Variable for Confirmations**

The contract maintains a mapping for confirmations to track the number of confirmations received for a transaction.

mapping(uint => mapping(address => bool)) public confirmations;

**7. Confirmed Count**

let’s create A function to count the confirmations for a particular transaction.

function getConfirmationsCount(uint transactionId) public view returns(uint) {  
 uint count;  
 for(uint i = 0; i < owners.length; i++) {  
 if(confirmations[transactionId][owners[i]]) {  
 count++;  
 }  
 }  
 return count;  
 }

**8. Check if Confirmed**

Let’s create a function that will help us determine if a transaction is confirmed or not! This function should return true if the transaction is confirmed and false if it is not.

function isConfirmed(uint transactionId) public view returns(bool) {  
 return getConfirmationsCount(transactionId) >= required;  
 }

**9. Execute Transaction**

let’s create a function to execute a confirmed transaction by sending the transaction value to the specified address

*Note: The transaction should only execute if it is confirmed. If not, revert the transaction.*

function executeTransaction(uint transactionId) public {  
 require(isConfirmed(transactionId));  
 Transaction storage \_tx = transactions[transactionId];  
 (bool success, ) = \_tx.destination.call{ value: \_tx.value }(\_tx.data);  
 require(success);  
 \_tx.executed = true;  
 }

**10. Confirm Transaction**

let’s make a function to confirm a transaction by an owner. If confirmed, it triggers immediate execution if required.

function confirmTransaction(uint transactionId) public {  
 require(isOwner(msg.sender));  
 confirmations[transactionId][msg.sender] = true;  
 if(isConfirmed(transactionId)) {  
 executeTransaction(transactionId);  
 }  
 }

**11. Submit Transactions**

Finally, let’s create A function allowing a user to create and immediately confirm a transaction.

function submitTransaction(address payable dest, uint value,bytes calldata data) external {  
 uint id = addTransaction(dest, value,data);  
 confirmTransaction(id);  
 }

**Testing**

const { assert } = require('chai');  
describe('MultiSig', function () {  
 let contract;  
 let accounts;  
 beforeEach(async () => {  
 accounts = await ethers.provider.listAccounts();  
 const MultiSig = await ethers.getContractFactory("MultiSig");  
 contract = await MultiSig.deploy(accounts.slice(0, 3), 1);  
 await contract.deployed();  
 });  
  
 describe('storing ERC20 tokens', function () {  
 const initialBalance = 10000;  
 let token;  
  
 beforeEach(async () => {  
 const EIP20 = await ethers.getContractFactory("EIP20");  
 token = await EIP20.deploy(initialBalance, 'My Token', 1, 'MT');  
 await token.deployed();  
 await token.transfer(contract.address, initialBalance);  
 });  
  
 it('should store the balance', async () => {  
 const balance = await token.balanceOf(contract.address);  
 assert.equal(balance.toNumber(), initialBalance);  
 });  
  
 describe('executing an ERC20 transaction', function () {  
 beforeEach(async () => {  
 const data = token.interface.encodeFunctionData("transfer", [accounts[2], initialBalance]);  
 await contract.submitTransaction(token.address, 0, data);  
 });  
  
 it('should have removed the contract balance', async () => {  
 const balance = await token.balanceOf(contract.address);  
 assert.equal(balance.toNumber(), 0);  
 });  
  
 it('should have moved the balance to the destination', async () => {  
 const balance = await token.balanceOf(accounts[2]);  
 assert.equal(balance.toNumber(), initialBalance);  
 });  
 });  
 });  
  
 describe('storing ether', function () {  
 const oneEther = ethers.utils.parseEther("1");  
 beforeEach(async () => {  
 await ethers.provider.getSigner(0).sendTransaction({ to: contract.address, value: oneEther });  
 });  
  
 it('should store the balance', async () => {  
 const balance = await ethers.provider.getBalance(contract.address);  
 assert.equal(balance.toString(), oneEther.toString());  
 });  
  
 describe('executing the ether transaction', function () {  
 let balanceBefore;  
  
 beforeEach(async () => {  
 balanceBefore = await ethers.provider.getBalance(accounts[1]);  
 await contract.submitTransaction(accounts[1], oneEther, "0x");  
 });  
  
 it('should have removed the contract balance', async () => {  
 const balance = await ethers.provider.getBalance(contract.address);  
 assert.equal(balance, 0);  
 });  
  
 it('should have moved the balance to the destination', async () => {  
 const balance = await ethers.provider.getBalance(accounts[1]);  
 assert.equal(balance.sub(balanceBefore).toString(), oneEther.toString());  
 });  
 });  
 });  
});

**Conclusion**

In conclusion, a Multi-Signature Wallet in Solidity offers enhanced security by requiring multiple approvals for transactions. Understanding its structure and functionality is crucial for secure and robust smart contract development.

**Final Thoughts**

Solidity enables developers to create intricate smart contracts like the Multi-Signature Wallet, fostering trust and security in decentralized applications. Further exploration and experimentation are encouraged to deepen understanding in this domain.

**Community and Resources**

1. Online Communities: Engage with blockchain developer communities, forums, and platforms like Ethereum Stack Exchange, Reddit’s r/ethereum, or Discord/Telegram groups.
2. Documentation and Tutorials: Refer to official Ethereum documentation, Solidity documentation, Alchemy University courses, and various tutorials available online.

*This marks the end of our tutorial if you found it useful feel free to clap or leave a comment*. *Any suggestions or feedback will be welcomed. You can also connect with me on* [***Twitter***](https://twitter.com/InyekakaB)*,*[***LinkedIn***](http://www.linkedin.com/in/christianinyekaka)*,*[***GitHub***](https://github.com/chrisBokotaII)**.**

[Solidity](https://medium.com/tag/solidity?source=post_page-----63641eaacd2b---------------solidity-----------------)

[Blockchain](https://medium.com/tag/blockchain?source=post_page-----63641eaacd2b---------------blockchain-----------------)

[Ethereum](https://medium.com/tag/ethereum?source=post_page-----63641eaacd2b---------------ethereum-----------------)

[Smart Contracts](https://medium.com/tag/smart-contracts?source=post_page-----63641eaacd2b---------------smart_contracts-----------------)

[Multi Signature Wallet](https://medium.com/tag/multi-signature-wallet?source=post_page-----63641eaacd2b---------------multi_signature_wallet-----------------)