Multi-Signature Smart Contract Wallet (Solidity & Ethereum)

Goal of this lab is to implement a multi-signature smart contract wallet, running in Ethereum. Multi-signature wallets require arbitrary selection of **n** owners out of **m** owners to co-sign a transaction containing a transfer of crypto-tokens, where n < m. A popular example is 2-of-3 multisig wallet. In the lab, you obtained a 2-of-2 multisig wallet, while you have to extend its functionality to support n-of-m.

A) Prerequisites & Environment:

We recommend you to use Linux Ubuntu/Debian environment with Solidity and truffle framework of 0.8.x version. See appendix for short intro to Solidity and Truffle.

1) Download and unzip truffle project. Observe the following files:

- ***./contracts/MultisigWallet.sol*** - represents the smart contract that implements 2-of-2 multi signature wallet.
- ***./config/wallet_config.js*** - the file contains configuration of the wallet, which is represented by the number of owners and minimum required signatures to execute any transaction.
- ***./migrations/2_deploy_wallet.js*** - the script is responsible for deployment of the wallet smart contract (i.e., correct calling of its constructor), while the parameters are obtained from the config file.
- ***./tests/TestWallet.js*** - represents tests that prove the correct functionality of the smart contract wallet. In other words, this file simulates the DAPP client application by interacting with the smart contract. There are several tests that demonstrate the correct functionality of the smart contract, while 3 of them are failing. Your tasks is to make them working properly by instruction given below.

2) Install Truffle suite*. Start by navigating to root directory of the truffle project and issue*

```
$ sudo apt-get install -y nodejs build-essential

$ sudo npm install -g truffle

$ npm install truffle-wallet-provider @truffle/hdwallet-provider @truffle/contract bip39
web3@v1.0.0-beta.34
```

3) Compile the contracts. *Note that for the 1st time it is necessary to compile with sudo, and afterwards no sudo is required:*

```
$ sudo truffle compile
```

4) Run blockchain and tests. Running tests causes compilation of smart contract automatically, so you do not need to compile them before running test. Observe what is going on in the tests and how they demonstrate a functionality of the smart contract wallet. We recommend you to run local blockchain and its log in one terminal:

```
$ sudo npm install ganache-cli
$ npx ganache-cli
```

which, after issuing will activate 10 default accounts of your local blockchain collaborating with truffle (including their private keys). Note that all of these accounts are accessible from the *TestWallet.js* file, and signing by them is done by adding *{from: accounts[0]}* field to the invocation of any transaction-based function (i.e., functions that modifies the state of the blockchain). Contrary, call-based invocations (i.e., not modifying the state of blockchain) can be usually called without this field by anybody.

In the 2nd terminal run tests by:

```
$ truffle test
```

which, after issuing will display information about: 1) successful contract compilation, 2) deployment of the contract by script, 3) results and printings of tests themselves. During the execution of tests, you may observe what transactions are appended to the blockchain in the 1st terminal and how much of gas their cost.

Note that the output of the deployment script shows who are the owners of the contract (compare the addresses with 10 truffle accounts), gas limit of the block, and the address of the smart contract deployed.

You will observe that three test fails. Your tasks is to make them working properly by resolving 4 tasks below. You are required to modify only two files: 1) smart contract in *MultisigWallet.sol* and 2) wallet_config.js that adjusts parameters of the wallet,

B) Task 1 - Protect against Replay Attack

Modify smart contract file to fix the replay attack and thus make the appropriate test working. By fixing it, you will not allow already confirmed and executed transaction to be executed multiple times. You might create the proper modifier and put it to the correct place (see placeholders in Solidity file). After fixing this issue, the replay test should pass.

C) Task 2 - Transform the 2-of-2 Wallet to n-of-m Multisig

So far, the wallet allows execution of transactions after signing by 2 owners from 2 owners. In this task, you have to transform the wallet in order to support arbitrary number of owners (m) with minimal required number of signers (n). In other words, you will implement an n-of-m multi-signature wallet, while the maximum number of owners is 10. To test the correct functionality, you might try different configuration options in config file, while the number of tests passing should be the same as before.

D) Task 3 - Check Enough Balance of Contract Wallet

Before execution of transaction that transfer Ether to some other address, you have to perform check whether the current contract has enough balance. If not, emit the event NotEnoughBalance with indicated amounts. After resolving this task, the next unit test should be passed

E) Task 4 - Write a Function for Retrieval of Owners that Sign a Transaction

DAPPs that interact with smart contract may require some convenient functions that retrieve required data from the smart contract / blockchain, while not spending any Ether for their invocation. For this purpose a function that only reads data from blockchain can be implemented. Note that such a function can be executed within the DAPP of the client for free, or optionally it might be executed by EVM if some state-modifying function is using them (in this case it costs gas). In your case, it is the 1st option. After implementing this function, even the last unit test should pass.

F) Task 5 - Investigate the Receipt of a Transaction

First of all, observe in the first terminal how much gas do you pay for the deployment of your contract (ignore a deployment of migration contract that costs around 300K of gas). What is the block number and how much you would pay for it in USD?

Some of tests prints the gas consumption of a few function calls of smart contract. Try to print the full receipt of confirmation and submission calls and comment on the meaning of the most important fields seen there. What events do you see there?

G) Task 6 - Deploy Your Contract on Ethereum Testnet Sepolia + Verify Its Source Code

1) Install Metamask wallet into your browser from https://metamask.io/

2) Select S*epolia* network in Metamask

3) Get some ETH on your created accounts by faucets. Try https://www.infura.io/faucet or see the list of faucets at official Ethereum web - https://ethereum.org/en/developers/docs/networks/.

4) Add your login to the dummy constant called "login" to make your smart contract code unique and to enable later contract code verification. Deploy the MultisigWallet contract to Sepolia testnet by truffle: First, export the private key from Metamask and put it into *.secret* file of the truffle project. Second run the following:

```
$ truffle deploy --network sepolia --reset
```

Note that you can alternatively deploy your contract by https://www.myetherwallet.com/ or https://remix.ethereum.org.

5) Copy the address of the contract from the output and paste it into Ethereum Sepolia Explorer https://sepolia.etherscan.io/

6) Click on the contract tab and start verification of the source code. You will need to set the proper compiler version 0.8.13 and then copy&paste the source code of your contract.

What to Hand In?

1) The modified file MultisigWallet.sol

2) The address of you verified contract on Sepolia network

Appendix

1) Solidity

Smart contracts are pieces of codes and data that are stored on the blockchain after their deployment - calling a constructor. If the contract is deployed on the blockchain, then its data (i.e., storage variables) can be modified by sending messages (a.k.a., transactions) that contains execution orders -- i.e., functions of the smart contract. Therefore, these functions must exactly define rules how storage variables of contracts are modified, and who can call these functions.

To specify visibility of functions, Solidity introduces several visibility modifiers:

- **public** - a function can be called by anybody, including the contract itself.
- **internal** - a function can be called only internally from the contract itself.
- **view** - a function can only read the storage variables, not modify them.
- **pure -** a function cannot read and modify the storage variables.
- **external** - a function can be called by other contracts, but cannot be called internally.
- **payable -** a function can accept Ether sent to the contract.

To specify a location of a variables (or their references) and parameters of functions during function execution by Ethereum Virtual Machine (EVM), Solidity introduces 3 variable specifiers:

- **storage -** the variable is placed in a persistent storage of the blockchain (i.e., state). They are usually used as "references" for variables in the storage, and they are used within the body of functions. Each update of storage variable is expensive, as it needs to be persisted on the blockchain.
- **memory** - the variable is placed in the memory of the EVM. It is default specifier for local variables of the functions and parameters of the functions. Variables of memory type are not persisted on the blockchain, therefore their modification is cheaper, in contrast to storage variables.
- **calldata** - this specifier is used for the parameters of external functions (i.e., functions called by other smart contracts) and reside in memory of EVM. The difference against the memory specifier is that calldata variables cannot be modified within the body of the functions, and thus they are constant.

The full documentation of Solidity can be found at https://docs.soliditylang.org/en/v0.8.13/