# Assignment 6 -

# Multi-Signature Smart Contract Wallet using Ethereum

## Security Tools Lab 2

Hand out : 19-Mar-2019

Hand in: 27-Mar-2019

| | |
|---|---|
| 1. | Objectives<br><br>By the end of this lab, you should be able to:<br>• Understand blockchain & smart contracts<br>• Understand Ethereum, Solidity & Truffle<br>• Implement a multi-signature smart contract wallet<br><br>System<br>• Ethereum VM<br><br>Notes<br>All needed files are on the VM and not Edimension this time.<br>Edimension only has this instruction and the slides. |
| 2. | 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.<br><br>Download and install the VM on VirtualBox. Solidity & Truffle have already been installed. |
| 3. | Download and unzip truffle project. Observe the following files:<br>• *./contracts/MultisigWallet.sol* - represents the smart contract that implements 2-of-2 multi signature wallet.<br>• ./config/wallet_config.js - the file contains configuration of the wallet, which i represented by the number of owners and minimum required signatures to execute any transaction.<br>• *./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.<br>• *./tests/TestWallet.js* - represents tests that prove the correct functionality of the smart contract wallet. In other words, this file simulates the 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 task is to make them work properly by instruction given below. |

| | |
|---|---|
| 4. | In a terminal navigate to '/Lab 6/multiSigWallet' directory on the Desktop<br><br>Compile the contracts. Note that for the 1st time it is necessary to compile with sudo, and afterwards no sudo is required (I've also already compiled the contract):<br>$ sudo truffle compile |
| 5. | Run blockchain and tests<br><br>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 run blockchain and its log in a different terminal.<br><br>Thus in 1 terminal :<br><br>$ truffle develop –log<br><br>which, after issuing will display you all 10 default accounts of your local blockchain collaborating with truffle. 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.<br><br>In the 2nd terminal run tests by:<br>$ truffle test<br><br>which, after issuing will display information about<br>1) successful contract compilation,<br>2) deployment of the contract by script,<br>3) results and printings of tests themselves.<br><br>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.<br><br>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.<br><br>You will observe that three test fails. Your task is to make them work properly by resolving 4 tasks below. At all times you are only required to modify only two files (Don't touch other files):<br><br>1) smart contract in MultisigWallet.sol and<br>2) wallet_config.js that adjust parameters of the wallet, |

| | |
|---|---|
| 6. | Task 1 - Protect against Replay Attack<br><br>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.<br>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. |
| 7. | Task 2 - Transform the 2-of-2 Wallet to n-of-m Multisig<br><br>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. |
| 8. | Task 3 - Check Enough Balance of Contract Wallet<br><br>Before execution of transaction that transfer Ether to some other address, you must 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. |
| 9. | Task 4 - Write a Function for Retrieval of Owners that Sign a Transaction<br><br>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, the functions that only read data from blockchain can be implemented. Note that such function can be executed within the DAPP of the client for free, or optionally they 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<br>option. After implementing this function, even the last unit test should pass. |
| 10. | Task 5 - Investigate the Receipt of a Transaction<br><br>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).<br><br>What is the block number and how much you would pay for it in USD?<br><br>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.<br><br>What events do you see there? |
| 11. | 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 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://solidity.readthedocs.io/en/v0.5.3/solidity-indepth.html

## 2) Truffle

The documentation of Truffle can be found at
https://truffleframework.com/docs/truffle/overview
Truffle contain 10 default accounts that we are using in this lab. Also, for simplicity, the order of signers corresponds to the order of these testing accounts.

- Since results of calls to smart contract function may cause some time to be evaluated (i.e., interaction with the blockchain), the javascript tests are using await statements to synchronously wait until the result of asynchronous promise objects.

| | |
|---|---|
| | • It may happen that you deplete all balance of some account after some time. Therefore, from time to time terminate the blockchain (1st terminal) and run it again.<br>• If you wish to debug any transaction seen in the 1st terminal, take its hash and pass it to truffle command:<br><br>$ truffle debug <TX_HASH><br><br>where, you can use enter key to move on the next instruction and 'v' command to display values of local and state variables.<br><br>• Tests are executed sequentially and their order matters.<br>• In the tests you may see transaction-based and call-based (containing call() ) invocation of functions of smart contract. The first one cost gas and the second type is for free and it is executed by the client. |