# Implementation of shared wallet

Artemis prework assignment 1: implement some form of on-chain quorum

Artemis Prework @ Git Hist.

Contract address 0x934c31492AEDd0d74B22344617DA607b8e66Fcf2.

Discord **guyon#4286**

August 10, 2022

### Introduction

A Multi-Sig wallet is implemented in which a submitted transactions needs to be signed by multiple authorized members before it is executed. In this document the approach, implementation and challenges are discussed. The word members and owners are used interchangeably throughout this document.

### Approach and setup

From software engineering best practices one should opt for test-driven development: first design and write tests, then the actual contract. However, given the zero-to-none JavaScript experience and the fact that the focus is more on learning Solidity and problem solving, the go-to approach is to list functional requirements, implement the contract, develop bare minimum unit tests, and deploy it to the Goerli network.

The aim is to leverage preliminary experience in software development, thus instead of using the in-browser Remix IDE a local node is run with Hardhat. This makes sure one immerse him- or herselves in the Solidity ecosystem from the start and allows for tracking of work with git and the ability to use developers' tools of preference (e.g. Visual Studio Code and shortcut commands with `make`). OpenZeppelin was looked into for upgrading and updating deployed contracts, however given the sheer number of possible contract addresses, for the current scope solely using Hardhat and a few contract deployments suffices.

### Contract design and implementation challenges

Only authorized members should be able to submit a transaction and take part in the process of submitting, approving, and executing transactions. This is implemented by setting the owners and minimum number of required approvals at the time of deployment, i.e. in the `constructor()` or `initialize()` (in case of OpenZeppelin).

By keeping track of all the submitted transactions in a integer indexed public array the contract keeps track of all transactions. Per transaction the recipient, the amount, the number of votes, whether it is executed or not, and a mapping with the approvals is stored. This way by design it is only possible for members to interact with own approvals. A challenge that might arise is that there is a limitation in the number of transactions the contract can store, given that each `Transaction` instance continues to exist throughout the existence of the contract, and thus the array can become quite large.

On each public function call an `event` should be emitted using `emit` such that the action is logged to the sidechain. Together with the public array of transaction this should allow for the decentralized app to check with the contract what the outstanding transactions are that still require more approval from owners. For the MVP of this contract deployment only few emits are implemented to explore its functionality, i.e. for `Deposit` and for the internal `_executeTransaction`, which gives insights on Etherscan whether a transaction is correctly executed.

### Next steps

The current contract contains basic functionality but provides room for quite some improvements and further implementations. First of all, the current unit tests are not exhaustive for all test cases, and thus should be thoroughly designed and developed. A second addition would be a concise application to host locally to test interaction with the contract with e.g. MetaMask. Thirdly, the repository contains (artifacts from) a set of frameworks (Truffle, Yarn, OpenZeppelin, HardHat, Ganache-CLI) that were explored during implentation, but for simplicity it would make sense to opt for only a minimal set of necessary frameworks. Finally, a (non-complete) attempt was done to containerize the contract. For shipping and reproduce-ability it is desired to develop and deploy the contract from within a container.

---

### Epilogue

The focus for this case was to dive into the different frameworks that the Solidity and JavaScript ecosystem provide and try to deliver a functional MVP, while showcasing my current understanding and skills of contract engineering and software development. With zero-to-none experience in both Solidity and JavaScript (my native language is Python in a machine learning engineering context), but with preference of working from the command line, it was a fun case that I thoroughly enjoyed. For all components that I think are relevant the code base contains at least some implementation, functional (contract and unit tests) or not (`Dockerfile`). Obviously I could spend tens, if not hundreds of hours more to make this a robust contract framework and making all open ends work, but the point was rather to come up with a basic functional contract and outline the ideas for future steps.