目录

[1 Network Premissioning 2](#_Toc28677)

[1.1 Network 2](#_Toc18137)

[1.1.1 Write the network layer permission smart contract 3](#_Toc20262)

[1.1.2 How to obtain Bytecode? 5](#_Toc21649)

[1.1.3 Test if network authorization works 12](#_Toc23366)

[1.2 Transaction Layer Permission 15](#_Toc27770)

[1.2.1 code explanation 15](#_Toc19033)

[1.2.2 how to add the permission to the chain spec json? 15](#_Toc14329)

[1.2.3 verify if the transaction layer permission works 16](#_Toc26189)

[1.3 Validator Set Layer Permission 17](#_Toc3812)

[1.3.1 Interface Explanation 17](#_Toc28375)

[1.3.2 What is a upgradable contract? 18](#_Toc9348)

[1.3.3 Kovan Validator Set Explanation 18](#_Toc6878)

[1.3.4 How to implement the validator set? (no need to implement) 21](#_Toc27104)

[1.3.5 What is solidity assembly language 21](#_Toc15900)

[1.3.6 fallback proxy contract 23](#_Toc14160)

# 1 Network Premissioning

OpenEthereum provides a number of features which enable the network participants to permission different aspects of a blockchain. Often conflated as simply “permissioned blockchains” we introduce permissions on a number of different layers:

* [Network](https://openethereum.github.io/Permissioning" \l "network)
* [Transaction type](https://openethereum.github.io/Permissioning" \l "transaction-type)
* [Validator set](https://openethereum.github.io/Permissioning" \l "validator-set)

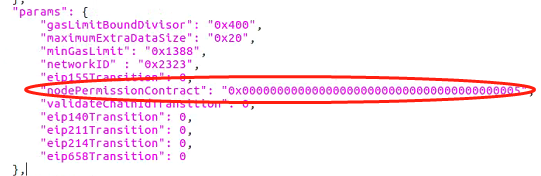
Each user can have different permissions on each layer. All permissioning is based on blockchain accounts, which means that permissions always correspond to an address.

## 1.1 Network

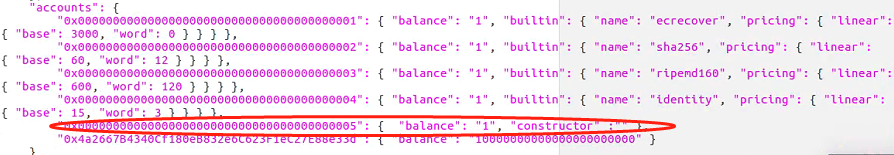
Permissions on this layer determines peer nodes connection. For instance, if we set the network connection permission for node 1 and node 2, then these 2 nodes are able to connect to each other, otherwise they will not be able to discover each other in the same blockchain network.

How to set the network layer permission?

1. Add a “nodePersissionContract” parameter in “params” section in chain spec json file. In our case, it is called demo-spec.json. <https://github.com/onebit256/poa> (see Picture 1)
2. Under “account”section in demo-spec.json, add a new contract address ‘0x00000...005’ and reserve this contract for the network layer permission contract.
3. Then write the network layer permission smart contract
4. compile the contract
5. Add the compiled byte code to constructor (see Picture 2)



Picture 1



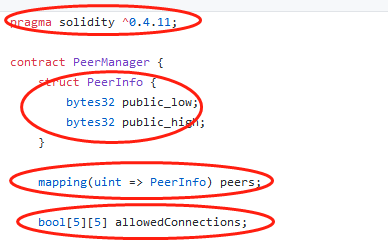
Picture 2

### Write the network layer permission smart contract

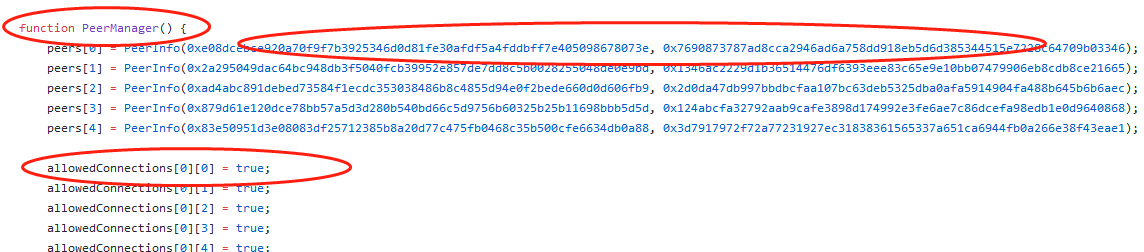
The smart contract can be downloaded from:

<https://github.com/onebit256/poa/tree/master/premissioning_contracts>

Code Explanation:



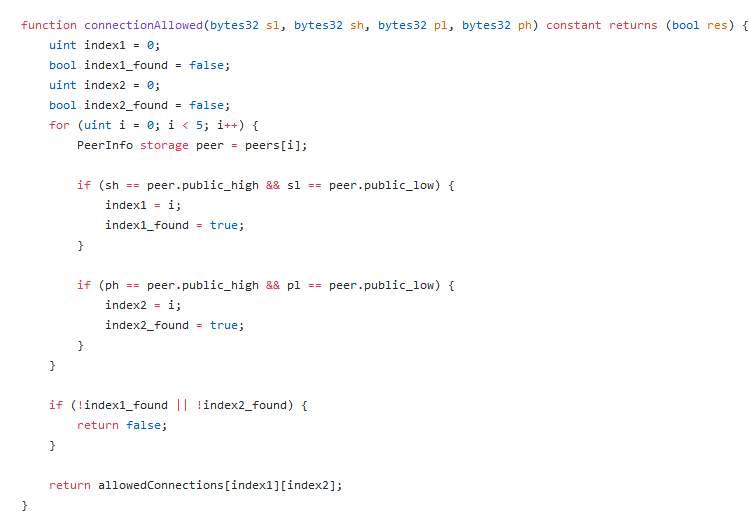
1. Specifies the solidity version
2. Struct is a kind of data structure in Solidity language, and we can customized our data structure using this data type.
3. Mapping is a kind of data structure in Solidity language, it is analogous to dictionary in other languages. Key and value pair
4. Array is a kind of data structure in Solidity language. Like arraylist in other languages



1. peerManager is the constructor of this contract. Constructor is used to initialize the smart contract
2. The EVM architecture allows 32 bytes words. As enodes addresses are 64 bytes long, they need to be cut in two parts for a smart contract to handle them. For example the enode enode://841015562d43c8037b127ee2a89f861d39beb468fecab72ad4bf369d3db8a01a5adeee0e0422cb021acea7ffeb0516db9e1211510ad353dc353b8c52165003c8 would be represented using :

* sl: 0x841015562d43c8037b127ee2a89f861d39beb468fecab72ad4bf369d3db8a01a
* sh: 0x5adeee0e0422cb021acea7ffeb0516db9e1211510ad353dc353b8c52165003c8

****



It fetches data from the list, and return true or false to allow or disallow the connection between 2 peers

### How to obtain Bytecode?

1. remix

<https://remix.ethereum.org/>

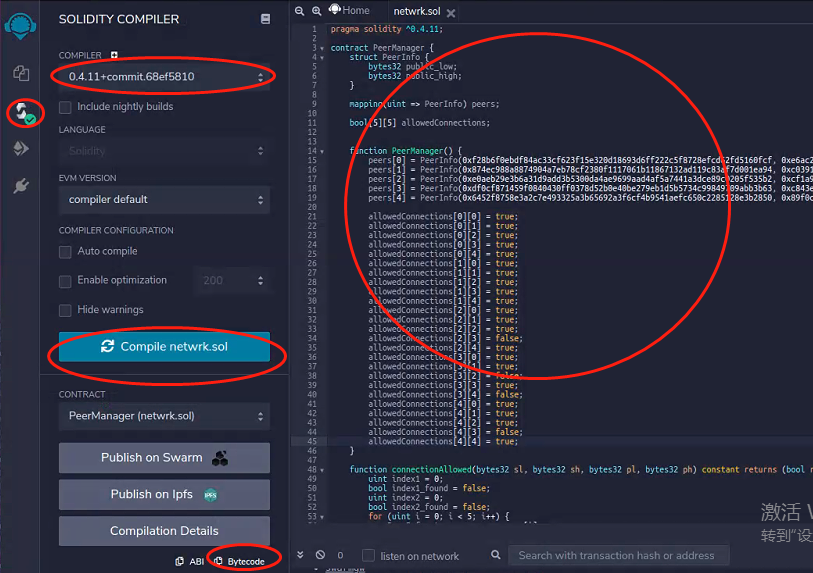
There are two ways to use remix:

1 use remix to write smart contract

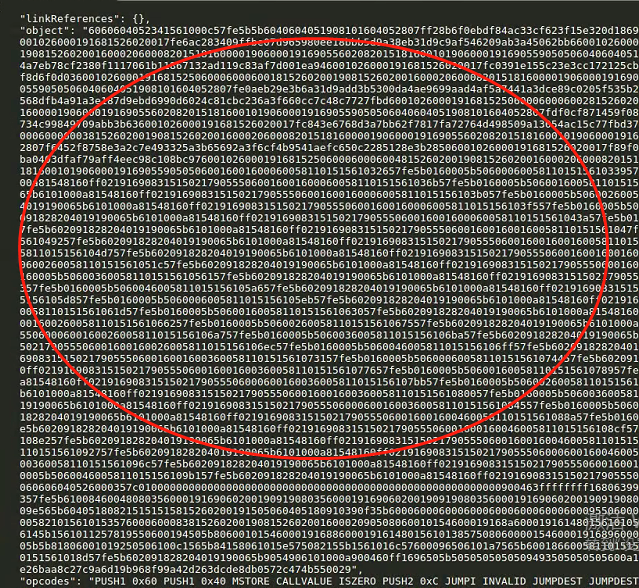
2 use remixd to connect with the local files

3 use truffle compile

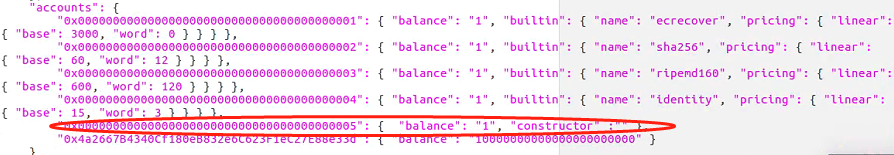
**1 use remix to write smart contract**



1. Create a new file and write smart contract on it
2. Go to compile panel and select the solidity version
3. Click compile button
4. Click Bytecode, and copy the bytecode to a text editor



You will get something like above, only take “object” part from 606.....0029, and add 0x in front, and paste that to constructor



**2 use remixd to connect with the local files**

**2.1 Install remixd:**

npm -i remixd

**2.2 Init truffle:**

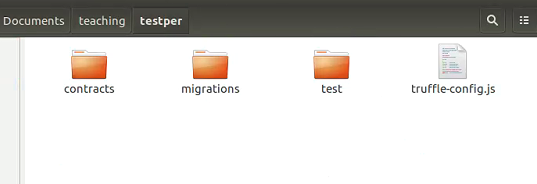
mkdir testper

cd testper

Truffle init

Ls

Cd contracts



You will see truffle auto generates contracts folder, migrations folder and test folder

Usage of these folders:

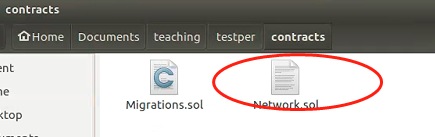
Contracts: use to store the contracts

Migrations: use to store all the contract deployment js files

Test: use to store the unit test js file

Truffle-config.js: use to write the configuration for the smart contract

**2.3 write peermamger smart contract and store it to contracts folder**



**2.4 Start Remixd:**



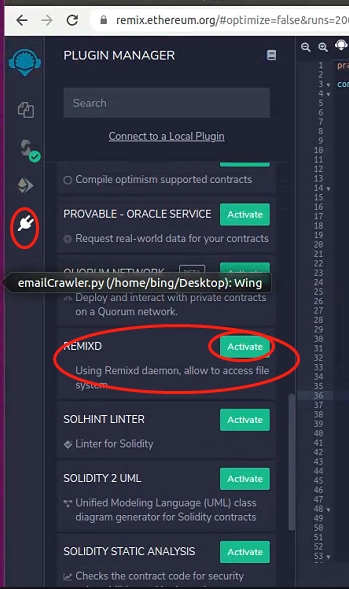
Type remixd --remix-ide <https://remix.ethereum.org> -s (absolute path of your truffle project directory)

In our case, it should be

remixd --remix-ide <https://remix.ethereum.org> -s /home/root/Documents/teaching/testper/

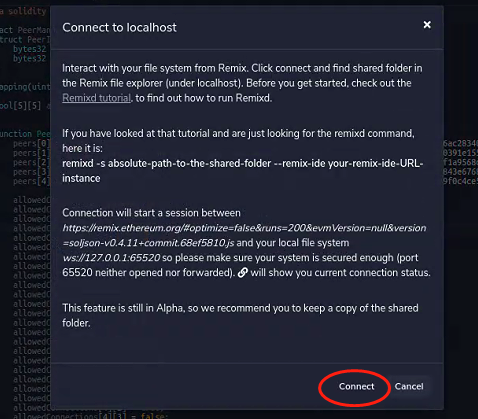
After you press enter, you should see above console log, it mean the program is monitoring 127.0.0.1:65520 port

2.5 Go to remix plugin and click remixd

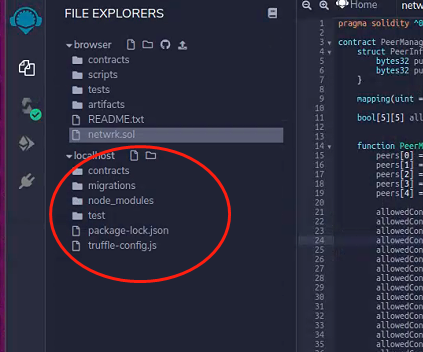


Click plug button

Click active



It will pop up this window, then click connect



Then you will see your local host files have been mapped to this remote remix ide

Then repeat the same compilation process and get byte code

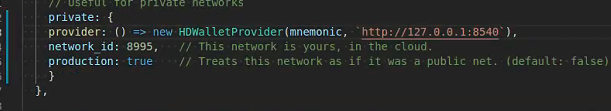
1. **Truffle**
   1. **download vscode and import truffle to vscode**

3.1 config truffle-config.js

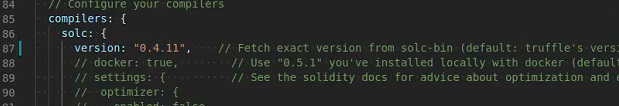
1. Comment this out and hard code the mnemonic



1. Comment private out and set the rpc url to be http://127.0.0.1:8540



1. add the right solc version

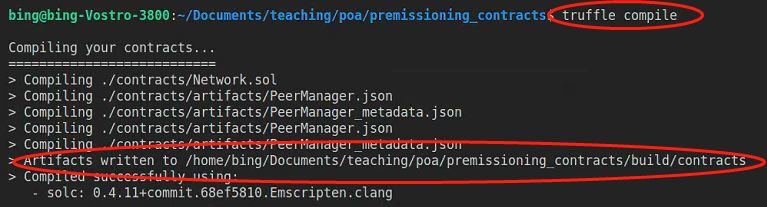


1. Install hdwallet-provider

Npm i @truffle/hdwallet-provider

Then open a terminal and type:

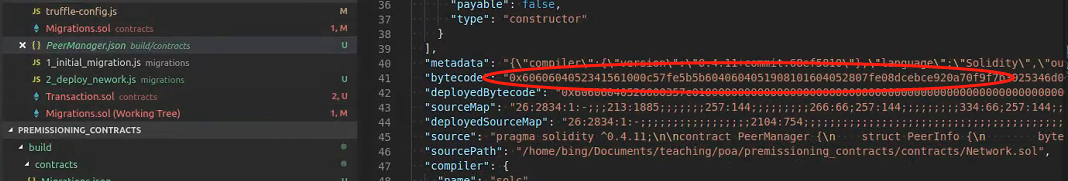
Truffle compile



You will be able to see the above console log.

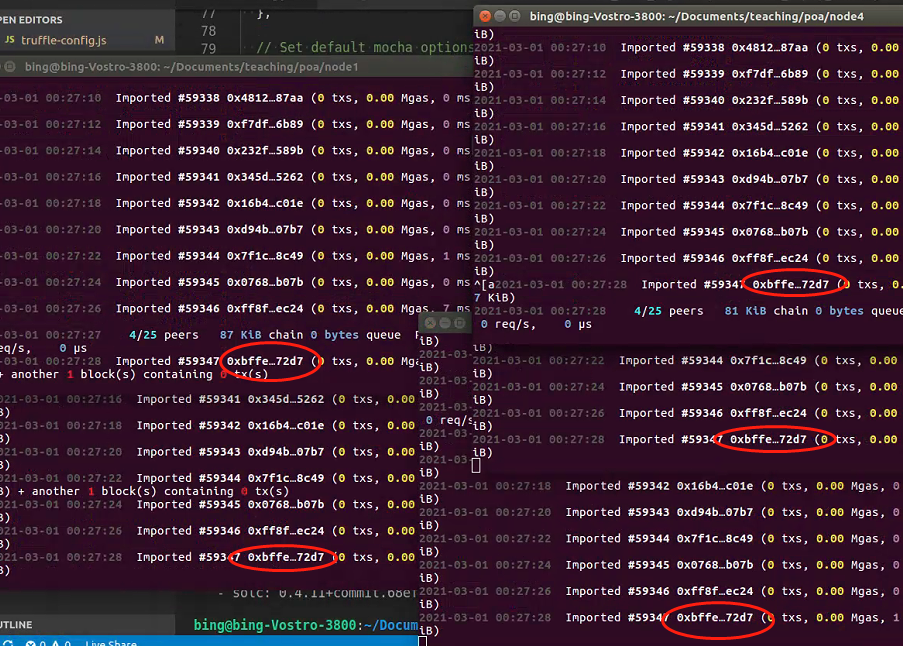
All the compiled files are stored in build/contracts folder

1. Go to build folder, then open compiled peermanager.json to obtain the bytecode



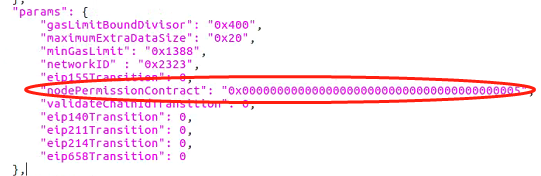
### 1.1.3 Test if network authorization works

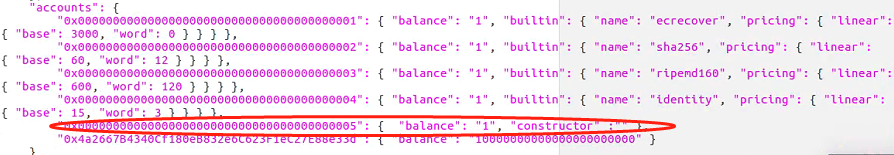
1. Start a 5 node network with demo-spec.json and test if they are connected by looking the block # hash



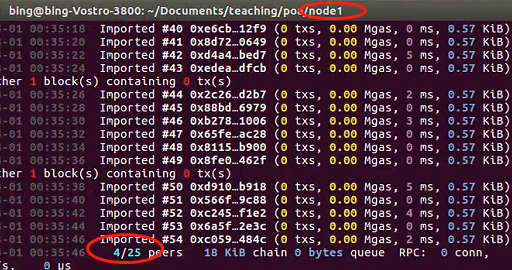
They all connected

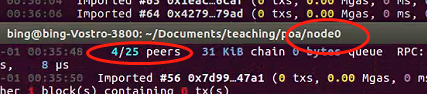
1. Create a new demo-spec-np.json and add compiled bytecode to constructor

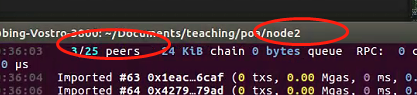


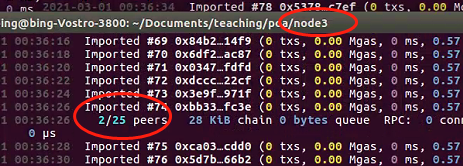


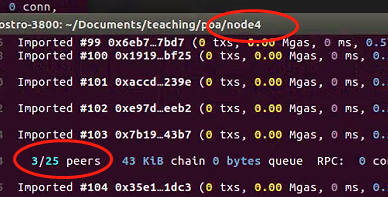
Then start the 5-node network with demo-spec-np.json and check the connected peer numbers



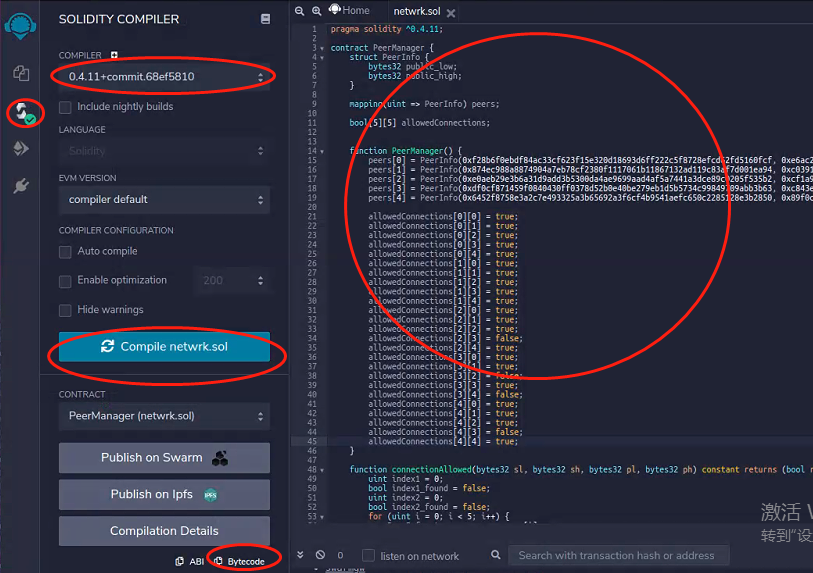






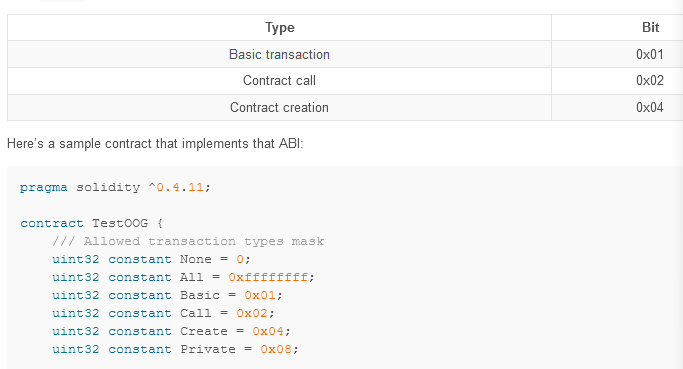


This result is what we expected, if we compare it with the code



## Transaction Layer Permission

### 1.2.1 code explanation





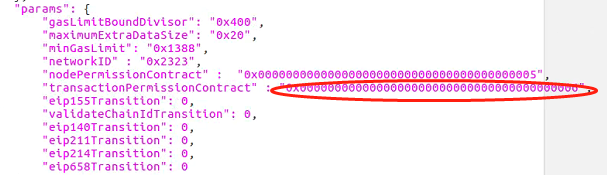
1. State variables define the transaction type in uint32
2. allowedTxType is the interface to set the transaction permission

### 1.2.2 how to add the permission to the chain spec json?

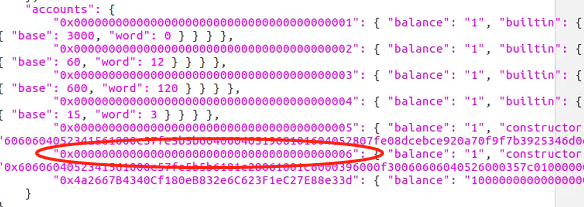
Step 1: use either remix or truffle compile the smart contract as what we did in section 1.1

Step 2: put the bytecode to chain spec file

1. Add transactionPermissionContract parameter and reserve address 0x000.......0006 for the tx permission contract

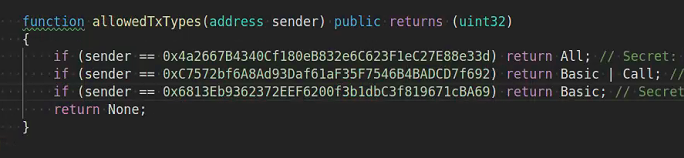


1. Add the bytecode to the contract



### 1.2.3 verify if the transaction layer permission works

Just start the node one with the demo-spec-tx.json



If we try to make a transfer, we are expecting to see that:

0x4a2 has the permission to deploy contracts, make transfers, and interact with call functions in contracts.

And 0xc757 has no permission to deploy contracts, but has permission to make transfers and interact with call functions

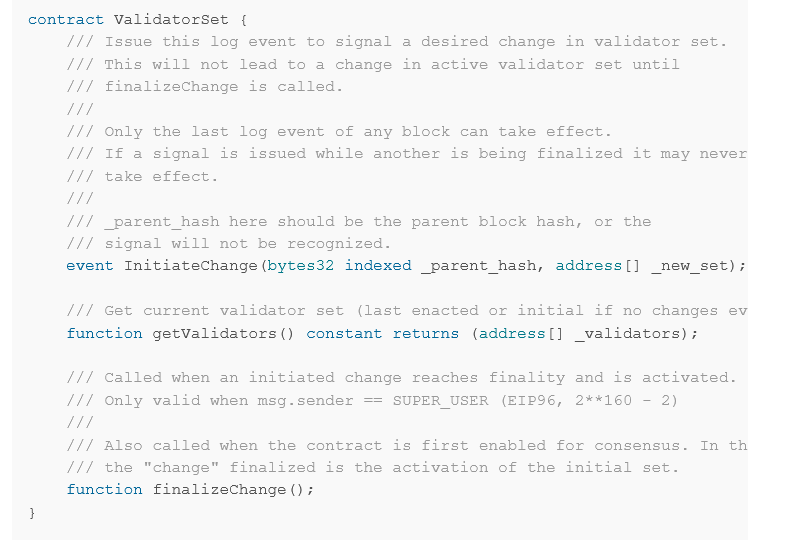
0x68 only has permission to make transfers

And other address have no permission to do anything

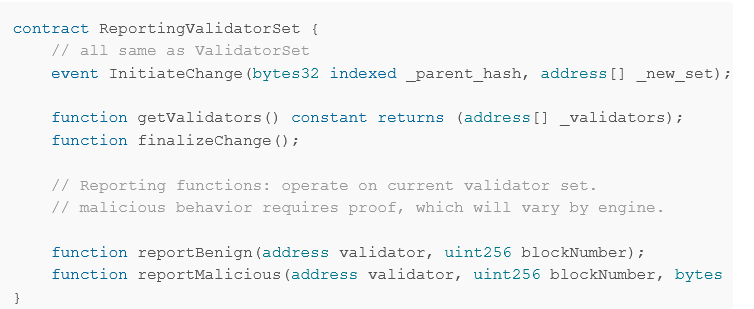
## 1.3 Validator Set Layer Permission

### 1.3.1 Interface Explanation

https://openethereum.github.io/Validator-Set

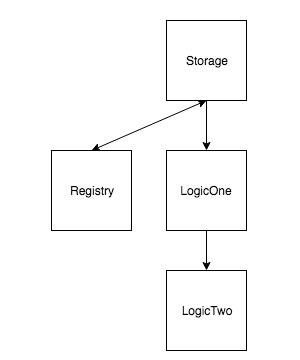


1. initiateChange just use to log the initiate change event
2. getValidators literally means get current validators
3. finalizeChange: when this is called the contract will set the new set of validators



1. ReportBenign: Monitor Benign nodes, check if this validator works well
2. ReportMalicious: Monitor malicious nodes

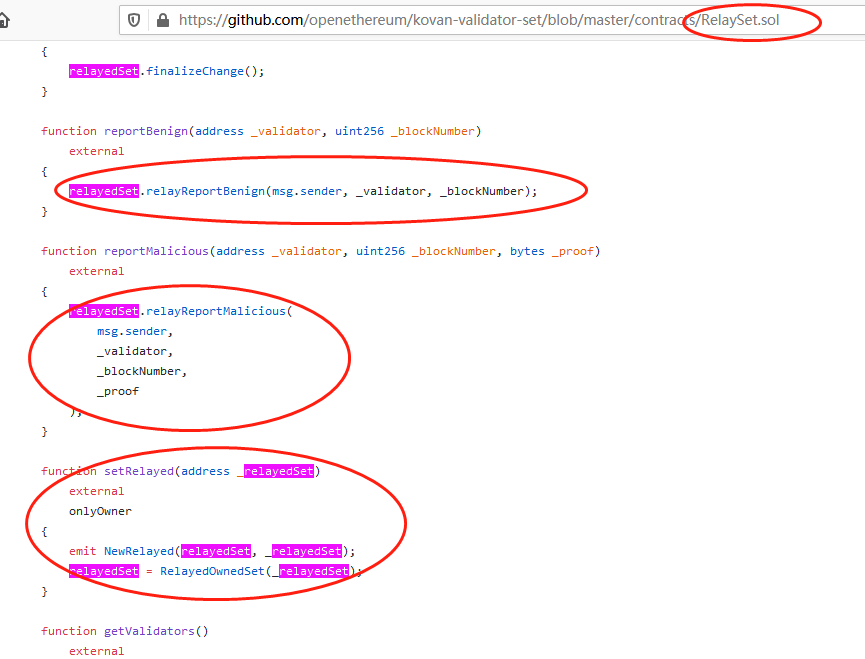
### 1.3.2 What is a upgradable contract?



### 1.3.3 Kovan Validator Set Explanation

<https://github.com/openethereum/kovan-validator-set>

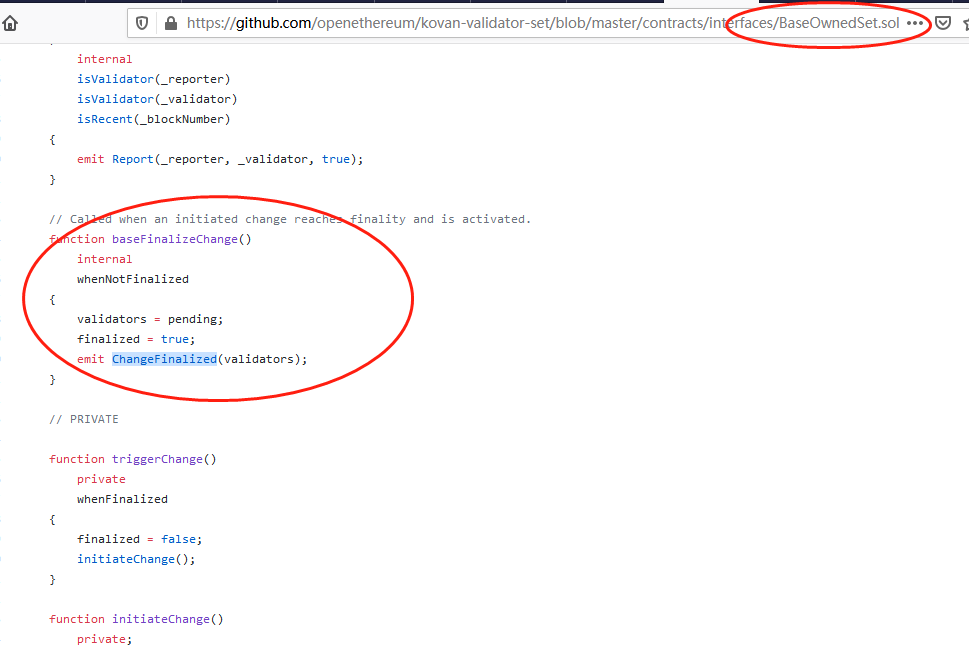
This is a upgradable smart contract



This is register / Proxy contract. It creates a logic contract instance, and call functions from the logic contract directly



This is the actual logic function, and the logic is implemented in the following contract



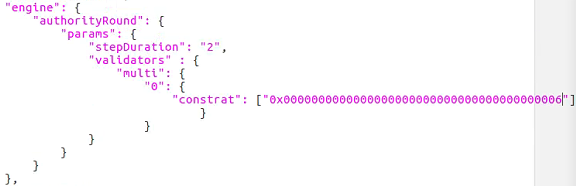
### 1.3.4 How to implement the validator set? (no need to implement)

Step 1: compile relayedSet and replayedOwnSet and reserve 2 address slots for these two contracts



Step2: change list to contract in engine section and put proxy contract address to it

Set ValidatorSet by Contract



Step 3: restart the network