**GPT Oraclizer Tutorials**

***Description***: This document is created to get one started in calling the Verification Oraclizer, which allows was to verify a signature came from a public key. It also is created to get one started in calling the Bigchain Oraclizer, which allows one to assign a digital asset to BigchainDB.

**Part 1: Calling the Verification Oraclizer**

In order to verify a digital signature, we need three things. The public key of the signer, their message hash from which the signature was created, as well as their signature.

The verification Oraclizer takes these parameters as a string. If you have these parameters in a different format, you will have to change them to type string.

Now, there are two things you need to know about the Verification Oraclizer. There exists a file, RequesterApp.js, in the following directory:

~/.eris/apps/SOLIDITY/VerifyOraclizerEthereum/wallet2

This application is where verification is actually done. You can think of it as a service. If it is not running, then your contract will not be able to be verified. However, it is very important that there is only one instance of this service running. If there are multiple instances running, it may threaten the integrity of the Oraclizer. So before you run it, make sure it is not already running. To check if it is running, you can call the following command:

$ ps -e|grep node

Let XXXX denote a process number. To kill that running process, you can do:

$kill -9 XXXX

The other thing you need to know about the Verification Oraclizer is that when you call its smart contract, you may not get the result right away. When you result is ready, an event is fired, which allows you to call a function in the Oraclizer to see your result. You may wonder why the event does not just display the result. If that were the case, it would be unnecessary to call a function to see your result. The reason this is done is a security consideration. This hides the result if one is able to mine your event.

Now, we are ready to get started. Start the requester app with the command,

$node RequesterApp.js

Recall that the directory D of this RequesterApp is:

~/.eris/apps/SOLIDITY/VerifyOraclizerEthereum/wallet2

Now that we are done with that, we are ready to make calls to the Oraclizer. Note that you will be making calls to the smart contract **Verification.sol**. You can in principle make a copy of the smart contract and move it to the directory of your smart contract. However, I recommend interacting with this contract as follows. It will make integration of multiple smart contracts and their javascripts simpler.

Let D’ denote the directory you are in. When you are finished writing your contract in solidity, you can interact with it, as well as the Verification contract in javascript. So let’s say you are in Directory D’, which is not the same directory D as the RequesterApp and Verification.sol contract. When you are finished writing your contract, you can decide in the javascript which functions are only allowed to be called after verification.

You can access Verification.sol (let’s denote its object as **VerificationContract**) simply by denoting directory D’ in your contract.

Here is a javascript template for doing so. Let **contract** denote your contract object with location of the contract and abi in this directory. Note, if you use parts of this template, you may have to change the parenthesis ‘’ in linux, as they need to be in the linux format for it to compile.

‘use strict’

//import appropriate libraries

var contracts = require(‘eris-contracts’)

var fs = require(‘fs’)

var http = require(‘http)//only necessary if you make a server for API calls

var accounts = require(‘./accounts.json’)

var contract\_address = require(‘./epm.json’).contractName //change to contract name

var contract\_abi = JSON.parse(fs.readFileSync(‘./abi/’ + contract\_address, ‘utf8’))

var requester\_address = require(‘/home/demoadmin/.eris/apps/SOLIDITY/VerifyOraclizerEthereum/wallet2/epm.json’).deployStorageK

var requester\_abi = JSON.parse(fs.readFileSync(‘/home/demoadmin/.eris/apps/SOLIDITY/VerifyOraclizerEthereum/wallet2/abi/’ + requester\_address, ‘utf8’))

var chainUrl = ‘http://localhost:1337/rpc’

//Instantiate the contract object manager (let xyz be your account)

var manager\_full = contracts.newContractManagerDev(chainUrl,accounts.xyz)

//Now we can access contracts

var contract = manager\_full.newContractFactory(contract\_abi).at(contract\_address)

var VerificationContract = manager\_full.newContractFactory(requester\_address).at(requester\_abi)

//Verification Parameters, normally they will be inputs to the javascript app by //creating server using the http library.

var msg = "8836a77b68579d1d8d4427c0cda24960f6c123f17ccf751328cc621d6237da22";

var pubKey = "02d7ceb37a16fde15a5237652b31a52d94def283a2ab09aaf5d1af48db1b84e20a";

var sig = "7051442bbf18bb2c86cbc8951a07e27ec6ba05ac3fa427e4c6b948e3dcf91a94046b048edf52445fb22cc776a94b87c3f55426f993458ec744f61f09fb46eeaa";

VerificationContract.VerificationQuery(msg,sig,pubKey, (function (error, result) {

VerificationContract.CallbackReady.once(function () {

VerificationContract.myCallback((function (error, result) {

if(result == “true”)

{

contract.someFunction(<…params…callback>)

}

else

{

console.log(“Verification Failed”);

}

}))//end myCallback

}//end CallbackReady.once

}))//end VerificationQuery

Some comments above. You will notice by the sequence of callbacks that the desired contract function is only implemented if the result of verification is true. First *VerificationContract.VerificationQuery* puts the query in the verification contract. The function *VerificationContract.CallbackReady.once* implements the logic at the next event. If you want the function to be implemented every time an event is fired from the VerificationContract, then simply remove the (.once) part. Next, the callback is obtained from the VerificationContract, and if the result is true, the contract function is called.

**Part 2: Calling the Bigchain Oraclizer**

Before you start, there are two things you will have to do. The first thing is that you must make sure that the driver is running in python. To do this, you will have to run this in the default directory in the bigchain server,

$python BigchainDriver.py

The second thing you will have to do is make sure you are in the BigchainOraclizer directory to run the app. This is back in the Eris Server. Just like with the VerificationOraclizer, make sure it is not running twice!

The directory is:

/home/demoadmin/.eris/apps/SOLIDITY/BigchainOraclizer/

Calling the Bigchain Oraclizer is analogous two calling the Verification Oraclizer. There are only a couple of different things you need to know. The first thing is that in making your address and abi in javascript, you will have to use the appropriate directory.

var bigchain\_query\_addr = require(‘/home/demoadmin/.eris/apps/SOLIDITY/BigchainOraclizer/epm.json’).deployStorageK

var bigchain\_abi = JSON.parse(fs.readFileSync(‘home/demoadmin/.eris/apps/SOLIDITY/BigchainOraclizer/abi/’ + bigchain\_query\_addr, ‘utf8’)

Additionally, you can look for the structure of a query in the bigchain directory. You will have to specify the name of the endpoint (ex. getKeys).

Note that talking to bigchain is to be done ONLY after public key verification. Here is an example of a bigchain query using the bigchain contract.

var bigchainInput = {“name”: username}

var bigchainEndpoint = ‘addData/’ + thePubkey + ‘/1’

var theobj = {“method”: “POST”, “stringJSONData”: bigchainInput, “endpoint”: bigchainEndpoint}

bigchain\_contract.BigChainQuery(JSON.stringify(theobj),function(error,result{

bigchain\_contract.CallbackReady.once(function(){

bigchain\_contract.myCallback(function(error,result){

console.log(result)

})

})

})