**Verification Oraclizer Tutorial**

Traditionally, you can see who calls a contract in solidity by the command *msg.sender*. However, this may not always possible. For example, suppose user U calls contract A. Then suppose contract A calls contract B. When msg.sender is called in contract B, it will return the address of the deployed contract A; not the address of user U.

For example, when the ballot contract wants to see who a validator is, *msg.sender* will always return the same address. The reason for this is the ballot contract will **always have the same *msg.sender***. The msg.sender will be the eris account in the javascript app that allows validators to interact with it.

This can be modified as follows. Suppose you have a function:

*function f(…parameters…)*

*{*

*address a = msg.sender();*

*if(a == 0x123…)*

*{*

*//some logic*

*}*

*}*

In order to overcome this, you need to add three parameters to your function f. Note that we make them strings. These three parameters are *a signature, a message hash and a signature.*

*function f(…parameters…, string signature, string msgHash, string theAddress )*

*{*

*address a = theAddress;*

*if(a == 0x123… && VERIFICATION(signature, msgHash, a)==TRUE)*

*{*

*//some logic*

*}*

*}*

Basically, we are using verification to verify that the person calling the contract is indeed the address passed in the function f (for otherwise, someone could call it without actually being the msg.sender).

Now, how do we call verification in a smart contract? There are two avenues. One is the Eris Verification Oraclizer, for Eris accounts. The other Oraclizer is for accounts consistent in Spencer’s wallet.

You can find these Oraclizers at:

**.eris/apps/SOLIDITY/ErisVerifyOraclizer**

**.eris/apps/SOLIDITY/VerifyOraclizerEthereum/wallet2**

The one noted as **VerifyOraclizerEthereum/wallet2** is an oraclizer that is consistent with Spencer’s wallets accounts. This tutorial will focus on these.

So first, make sure you have the file **RequesterApp.js** running. The command **node RequesterApp.js** will cause the verification service to run.

Now that the application is running, you can copy the code of the contract **Verification.sol**

Copy the code into your environment and deploy it.

Now, you will have to use the code **import Verification.sol** into your contract which you want to call verification inside.

Here is an example of such a contract:

*import “Verification.sol”;*

*contract example*

*{*

*Verification V = new Verification();*

*function VerifyIt(string msgHash, string signature, string pubkey)*

*{*

*V. VerificationQuery(msgHash, signature, pubkey); //this returns nothing. It sends*

*}*

*function VerifyItResult()*

*{*

*String result = V.myCallback(); //this returns your call back. It will tell you the transaction //is still in progress if it is not ready*

*}*

*}*

*}*

As visible in the example above, doing the verification will not give you your result. You have to call it by the function myCallback in your contract. So how do you know when your callback is ready? The answer is events. The event is **CallbackReady**. In javascript, you can see callback ready as exampled here:

contract.CallbackReady.once(function(error,result)

{

//put your logic here

});

Using “.once” allows you the next time the event is fired. If you fail to use “.once”, your code will be implemented every time the event is called.

Note, using the oraclizer, you can only make one request at a time. Also be wary that you must use the function removeMyRequest() to remove your request—it allows you to put in a new request before your current is processed.