Ledger Attestation Demo

Stefan More

2022-12-12

Contents

1 Introduction

A distributed ledger (DL) is an attractive system to decentralize storage and computation and thus decentralize the trust. For example, in our Edu WoT project, we use a DL to make available a Web of Trust as decentralized (root of) trust store to (among other things) authenticate education credentials and universities around the world. Since the project uses a smart contract published on a DL as data store for the Web of Trust graph, the governance of the system is only determined by the rules encoded in that contract. The advantage of this approach is that no trust in a single organization is needed.

Retrieving of the web of trust graph is usually done by a verifying party by communicating with one of the DL's nodes. While this ensures freshness

of the data, this process requires a network connection to one of the DL's nodes. If the verifier is offline, it cannot reach the DL, and thus cannot retrieve an authenticated copy of the web of trust.

The idea of this project is to enable a DL to issue attestations about the state of smart contracts published on it. In specific, we propose to extend the functionality of DL nodes in a way that users can request such attestations, the nodes create this attestation based on the ledger blocks available to them, and sign the attestations using a multi signature scheme. The user can then combine the attestations of all nodes to one attestation, thereby proving consent of the nodes about a certain state to an offline verifier.

1.1 Demo Introduction

In this demo we use the Leder Attestation (LA) tool to retrieve the attestation of some data stored on the DL. The relevant actors are the nodes of a Distributed Ledger (DL), a User who wants to retrieve the attestation, and a Verifier to whom the User wants to show the attestation in an offline setting.

In this demonstration we show the following:

1. installation of the LA tool dependencies
2. deployment of a demo smart contract
3. retrieve attestations from DL node(s) and aggregate them
4. verify the aggregated attestation

2 Demo setup

To initialize the demo, we first set some environment variables. Afterwards, we continue by installing the required dependencies.

```
CODE="`pwd`/.."
DEMO="../../LAdemo"

mkdir -p $DEMO

echo "Code: $CODE"
echo "Demo: $DEMO"
```

```
Code: /home/smore/projects/lsa/demo/..
Demo: ../../LAdemo
```

2.1 Install dependencies: web3iaik

Before performing the demonstration process on your own, you need to setup the modified `web3.js` library. web3 is the Ethereum JavaScript API used to interact with nodes of the Ethereum ledger. To call smart contracts functions the library offers a `call` method, which is used to invoke the respective `eth_call` method on the DL's API. We extend web3 by adding the `callSigned` method. This method is used to invoke a DL's `eth_callSigned` method, and handles the attestation appended to the result.

Thus, we build our modified web3 library, and install it into the user's folder:

```
cp -r $CODE/web3iaik/ $DEMO
cd $DEMO/web3iaik

npm install lerna
npm run build

mkdir $DEMO/user && cd "$_"
npm init -y
npm install ../web3iaik
```

2.2 Install dependencies: DL nodes and Verifier

We install dependencies required by the Python webserver components (used by DL nodes and the Verifier).

```
cd $CODE/DLnode
pip3 install -r requirements.txt
```

2.3 Deploy demo smart contract

Since our project is attesting the return value of a smart contract deployed on a distributed ledger, we provide a demo smart contract deployed on an Ethereum-based test ledger. You can also execute this demo by deploying your own smart contract.

- Deploy the `RevokedAccounts.sol` contract on an Ethereum-based DL
  - e.g., the tau1 test ledger on `rpc.tau1.artis.network`
  - e.g., using `https://remix.ethereum.org`

- Address of our demo deployment: `0xbd1Eec567211e5Ee94ca92e84F6a2d7D21Ed25C0` on tau1

3 Demo run

Our demo consists of three components.

On the ledger side, we provide a webserver that provides a API compatible to the standard Ethereum JSON RPC API. The webserver wraps the RPC API of a DL node, extending it to provide attestations of calls to smart contract functions. Besides this extension, the wrapper does not change any API functionality, it can therefore be used like the normal RPC API of an Ethereum node. In the background, our API is calling the RPC API of a (tau1) node. After receiving the result to the function call from the DL, the wrapper is then signing the result using the private key of the DL node, thus attesting its provenance. In a real-world deployment this wrapper API is hosted on the DL node itself, thus inside the same trust boundary. While this attestation acts as the proof by one node, to acquire an attestation by the ledger itself, the user needs to retrieve attestation from many (or all) DL nodes. Instead of increasing the trustworthiness of the attestation, this also increases the likelihood that the attestation was signed by a DL node that is trusted by an (previously unknown) offline verifier. After retrieving the attestation of several DL nodes, the user can combine the attestation into an aggregated attestation (by aggregating the signatures). In our demo, the task of collecting multiple attestation and aggregating them is performed by the first wrapped node called by the user, therefore the user does not need to call all nodes themselves.

On the user side, we provide a demo client that uses our modified web3 library to initialize and execute a call to a smart contract. The web3 library provides the same functional API established in the Ethereum world. Instead of interacting with the RPC API of a DL node itself, the web3 library instead interacts with our API wrapper. By doing so it not only receives the result of the function call, but also an attestation of the result. In our demo, this attestation is already an aggregated attestation jointly issued by a set of DL nodes. The user can then store this attestation, and use it at a later point, e.g., in an offline showing to a verifier.

On the verifier side, we also provide a webserver that provides an easy to use verification API. This verification API is configured by the verifier by specifying a set of trusted (DL node) keys as well as trusted smart contract addresses and expected return values. To verify an attestation they retrieved

from an user, the verifier then simply calls the verification API with this attestation (a JSON data structure). In this process, the verification API checks if the attestation was indeed signed by the trusted keys, and also verifies the (aggregated) signature of the attestation. After establishing trust in the attestation, the API also checks if the function call represented in the attestation was issued to the trusted smart contract, and whether the result is as expected.

3.1 Start the DL Node wrappers

As a first step, we start the DL API webserver, which implements the attestation and aggregating functionality described above. In our demo, we start two webservers, simulating a DL with two nodes. Those nodes are aware of each other (via the provided configuration) which is necessary for the confenience functionality that directly collects attestations of all nodes.

```
PIDFILE="$DEMO/demo.pid"
function startNodeAPI()
{
        pass="node$1_passhprase"
        port=$((4999 + $1))

        echo "# Starting API for node $1 at port $port ..."
        python3 $CODE/DLnode/api.py --port $port --key $pass --config  $CODE/DLnode/co
        echo $! >> $PIDFILE
}

startNodeAPI 1
startNodeAPI 2
```

```
# Starting API for node 1 at port 5000 ...
[1] 497451
# Starting API for node 2 at port 5001 ...
[2] 497452
```

3.2 Start the Verifier API

As a second step, we also start the webserver of the verifier API. This tool can be used by any verifier, also in an offline setting. In the configuration we specify which DL nodes the verifier considers trusted. As a convenience functionality in an online setting, the verifier API can directly fetch the

public keys of trusted DL nodes. Additionally, the configuration also defines the trusted smart contract address and expected return value of a call to it.

```
function startVerifierAPI()
{
        echo "# Starting API for verifier ..."
         python3 $CODE/LAverifier/verifier.py --config $CODE/LAverifier/config-verifye
        echo $! >> $PIDFILE
}
startVerifierAPI

# Starting API for verifier ...
[1] 588358
```

3.3 Call smart contract & retrieve attestation

Next, we issue a call to the `isRevoked` function of the smart contract we deployed. Since we use the `callSigned()` function (instead of the standard `call()` function), we retrieve an attestation alongside the functions return value. This attestation states which smart contract address was called (in the `to` field), and also specifies the call (function name and parameters, encoded in the `data` field) and return value (in `result`). Additionally, it (in the `proof` block) the attestation contains the (aggregated) attestation signature, and the public keys of the set of DL nodes which issued the attestation. This attestation can then be saved into a wallet, and later used, e.g., in an offline setting.

```
function runWeb3demo()
{
        echo "# Executing $1 ..."

  cp $CODE/demo/demo2.js $DEMO/user
  cd $DEMO/user
        node $1
}

runWeb3demo demo2.js

org_babel_sh_prompt> # Executing demo2.js ...
address 1 (0x11223344556677889900) revoked? -> true
address 2 (0x11223344556677889911) revoked? -> false
```

address 2, signed response:


```
{
   "call":{
      "id":2,
      "jsonrpc":"2.0",
      "method":"eth_callSigned",
      "params":[
         {
            "data":"0x5154d6491122334455667788991100000000000000000000000000000000000000
            "to":"0xbd1eec567211e5ee94ca92e84f6a2d7d21ed25c0"
         },
         "latest"
      ]
   },
   "id":2,
   "jsonrpc":"2.0",
   "proof":{
      "pubkeys":[ "a9923e8e5031360f7ff7ca3681194f94e9e5ec7b2388dd5aa055d2ab42d0091d8ad0
      ],
      "sig":"92d11b08c222e4b1bb712308eb9e759bc654f9123e88d45f25ee0ef5606fd4ad9d67968a62
   },
   "result":"0x0000000000000000000000000000000000000000000000000000000000000000"
}
```


3.4 Verify attestation

Later, the user provides the attestation to an (offline) verifier. This verifier uses the verification tool to verify the trustworthiness of the attestation (as described above).

```
cd $CODE/demo
curl -X POST http://localhost:1234 -H "Content-Type: application/json" -d @demo-attesta

{"result":"Success: Contract accepted and result okay",  "status":"True"}
```

3.5 Stop demo: kill the webservers

```
while read pid; do
        process=`ps u | grep $pid | tr -s ' ' | cut -d ' ' -f 12-14`
```

```
        myecho "# killing $process ..."
        kill $pid
done <$PIDFILE

rm $PIDFILE
```