How do blockchain wallets work

Assignment 04: Implement public/private keys, a Wallet, and a client server.

Due Date: May 14th

This homework is to implement private key signature. This is how a user keep other people from spending their coins.

Recent Code Changes

There are 3 pieces to this assignment.
1. There is a command line tool for key management and signing stuff. 2. There is a “server” this has the transaction stuff in it. 3. There is a “client”. The :wq

1. The server is changed to be a HTTP server so that a second process, the wallet, can call it and sign transactions. This means large changed to take some extra command line parameters in `./bsvr/main/main.go` and to have a running HTTP server.
2. The wallet and it’s client code is in `./wallet-client`.

Overview

The blockchain keeps a “distributed ledger” (DLT) but has no information on who can update the ledger data.

The wallet is a list of public/private keys that the user keeps on their computer. The user needs to keep the private key part private. The private key is used to digitally sign a message to the blockchain so that the blockchain knows that the message is valid. The message tells the blockchain what to do. In the simple case the message will tell the blockchain to send funds from one account to some other account. For this homework this is the primary thing that we will be implementing.

I have updated Assignment 4 to include a HTTP server that will listen for a number of requests. One of these is a “send-funds-to” operation. This operation requires a signed message. When it receives a properly signed message it will call the underlying code from Assignment 4 to send funds form the owners account to some other account.

This home work is to implement the client code that will keep a wallet and send messages that are signed to the server.

To Start the Server

The server is a HTTP server. You will need to have created a genisis block. After that you can run a server on localhost with a port of 9022 with:

```
$ cd 05/bsvr/main
$ go build
$ ./main --server 127.0.0.1:9191
```

You will want to run this in it's own window as it runs until it is killed or until you send it a shutdown message.

To call the server from the client.

A client has already been built that has a number of commands in it. It is in `./wallet-client`. You can use a browser to enter the URLs and call the server. A browser is a complicated set of software that hides lots of details. Instead we will just use a simple command line client that performs GET requests.

Already Implemented CLI commands.

```
./wallet-client --cmd echo
./wallet-client --cmd list-accts
./wallet-client --cmd shutdown-server
./wallet-client --cmd server-status
./wallet-client --cmd acct-value --acct 0x0000Some-Acct-Number
./wallet-client --cmd new-key-file --password "SomePassword"
./wallet-client --cmd list-my-accts
```

Your homework CLI commands.

```
./wallet-client --cmd validate-signed-message \
        --acct 0x000SomeAcct --password PasswordForAcct
./wallet-client --cmd send-funds-to --form 0x00FromAcct \
        --to 0x00000000ToAcct --password "PasswordOfFrom" \
        --memo "a-memo" --amount  ###
```

Cmd	Description
echo	just print out a note that the CLI got called to verify the command lie process.
list-accts	This works the same as the server version - list the accounts on the server

Cmd	Description
shutdown-server	send a message to the server to shutdown.
server-status	Find out the status of the server - is it up - is it running.
acct-value	find out the value of an account
new-key-file	Create a new account - this runs purely locally - no server call
list-my-accts	list the accounts in your local wallet. no server call.

Homework Overview.

Implement the client side to create and sign messages first. Add print statements to the server to show that you are getting the correct data passed from client to server. Then when you have correct data on the server go and implement the server signature verify code. Use the supplied command line tool, `sig-test` to validate signatures before you implement the server code. Use the `sig-test` code to see what a signature should look like.

The provided client code.

The sample client code is in `wallet-client`. It is in the file `client-main.go`. This progam is partially complete. The code that is currently there can:

1. Client Side: Create a new key file.
2. Client Side: List keys in the wallet.
3. List the set of accounts that the server has.
4. List the value of an existing account on the server.
5. Make a request to just validate a signature with a random generated message. This will require implementing the serer code to validate signatures.
   This is in the `client-main.go` at line 129. When you get this to work the part (5) should be easy. The randomly generated message will be replace with the request to send funds. All the hard work for signing is done.
6. Send a request to the server to send funds from one account to a destination account. The “sign” message part is missing. You will need to implement this.
   1. Take the request message in JSON.
   2. Create the `Keccak256` hash of the message string.
   3. Create the signature for the hash in (2) using the users key.
   4. Send the request to the server with `msg`, `msgStr`, `signature`. This is in the client-main.go near line 133.

Pseudo code for the send funds function on the client

1. If the Memo parameter is “” then put in a constant memo - something like “none”.
2. Use ‘RequiredOption’ function to get `from`, `to`, `amount` from the command line.
3. Format a JSON message to sign

```
msg := fmt.Sprintf(`{"from":%q, "to":%q, "amount":%d}`, From, To, Amount)
```

4. read in the key file for the `From` account using getKeyFileFromAcct.
5. Read in a password for this using getPassphrase
6. Call GenerateSigntature to sign the message.
7. Generate the URL to send - the communication end point is `/api/send-funds-to`. It requires “from”, “to”, “amount”, “signature”, “msg”, and “memo” parameters to be passed.
8. Call DoGet with the set of parameters.
9. Check the return status of the “GET” call, 200 indicates success, all other codes indicate an error.

The provided server code.

The server should run and do lots of stuff. You need to test the section where the signature is actually validated. This is in ./cli/svr-lib.go `line 115. If the signature is valid and the message is valid then it should return` isValid `as true. If an error occurs, for example a bad address or some other error during the validation process, then return` isValid` as false and the error. Most of the code for this part of the assignment has been adapted from signMessage.go in sig-test.

Helpful things (Pay special attention to this section!)

`./sig-test` has a full command line tool that implements signing and validation of messages. 95% of the code is pulled form go-ethereum. This has the code in it to build the message signing.

Look in sig-test/verifyMessage.go near the bottom for `func VerifySignature(addr, sig, msg string) (recoveredAddress, recoveredPublicKey string, err error) {` This function takes an address, `addr`, a signature `sig` and a `msg` and returns the recovered address and the public key for the signature. If `err != nil` then it verified, if not the the message did not verify. Look at line 95 - where it return empty strings for the first two return values and an error - this is a non-verified message.

Look at ./sig-test/sig-test/signMessage.go near the bottom. `func GenerateSignature(keyFile, password string, inMessage []byte) (message, signature string, err error) {` signs messages. Since the client you are building signs messages this would be a good chunk of code to poke through.

what to submit.

Your modified version of `./wallet-client/client-main.go`. If you create more files for the client then submit those also.

Notes

1. Uses some library functions to create a public/private key pair. (Implemented for you by the instructor)
2. Lists the set of public/private key pairs in the “wallet.json” file. (Implemented for you by the instructor)
3. Makes a status request to the blockchain to find out if it is up and running. (Implemented) Do this part first.
4. Updates the password on a public/private key pair. This is an all-local operation that will not require the blockchain server to be running. This is optional. It has not been implemented but it would be useful. It is in the sample command line tool.
5. Makes a request to a blockchain node with a signed message to transfer funds. Homework.
6. Makes a request to a remote node to list all the accounts (Implemented for you by the instructor)
7. Makes a request to a remote node to list the value of an accounts (Implemented for you by the instructor)