Homework 5

COM402 - Information Security and Privacy 2017

- The homework is due **Thursday, 2nd of June 2017, 11pm UTC+2 on Moodle**. Submission instructions are on Moodle. Submissions sent after the deadline **WILL NOT** be graded.
- In the event that you find vulnerabilities, you are welcome to disclose them to us (can even have a bonus !)
- For Exercises 2, 3 and 4, do not forget to submit your source files to Moodle along with your tokens.
- **Docker setup issues**: Hopefully most of you have a working docker setup on your machine. In case you still experience difficulties, please work in the VM that we provided on moodle, which has docker installed. Also, in case you're using the VPN and experience weird behavior, please run your scripts while connect to the epfl wifi network. **The only setup we provide support for is VM & epfl wifi network**. Also, there will be a few laptops available in BC263 that are already set up for you to solve the homework. If you prefer to work on these please send an e-mail to the TAs.

Exercise 1: Yao's Garbled Circuits

Solve the exercise-sheet on
http://moodle.epfl.ch/pluginfile.php/1788457/mod_resource/content/1/hw5_ex1_yao.pdf
By entering the values in the moodle-quiz:
http://moodle.epfl.ch/mod/quiz/view.php?id=952906

Take care: you only have 2 attempts! So solve it first on the sheet, and only if you're sure it's OK, solve it on the webpage.

Exercise 2: Blockchain

Everybody has a blockchain - so why not you, too? We set up a simple blockchain at com402.epfl.ch that waits for new blocks. You don't have to include any broadcast-protocol like in bitcoin, but we ask you to perform some mining and send the blockchain to the com402-server.

Installation

To interact with our cothority where the blockchain is handled, you need to install some packages using pip. Please note that only versions >= 3.5 of python work! If you get

```
async def getStatus(url):
            ^
```

```
SyntaxError: invalid syntax
```

during the installation, this means you're running a version of python that is too old. So, install the packages using:

```
pip3 install cothority~=0.1.0 websockets protobuf
```

and you're ready to go.

Communicating with the blockchain

To communicate with the blockchain, you have two methods:

```
# returns a list of all blocks that are stored in the skipchain
# identified by the genesis-string.
# Input:
# - url where to contact the skipchain
# - genesis-block (in binary format)
# Output:
# - all blocks since the genesis-block
# Error: websockets.exceptions.ConnectionClosed indicates that
# something went wrong. The error-message has more information.
cothority.Cothority.getBlocks(url, genesis)

# Asks the skipchain to add a block
# Input:
# - url where to contact the skipchain
# - block to store on the skipchain
# Output:
# - previous block of the blockchain
# - latest block of the blockchain which should be yours
# Error: websockets.exceptions.ConnectionClosed indicates that
# something went wrong. The error-message has more information.
cothority.Cothority.storeBlock(url, block)
```

For creating a block, you can use the convenience method:

```
# returns a new block that appends to the last block
# Input:
# - last valid block from the blockchain
# - data that will be stored in the new block.
cothority.Cothority.createNextBlock(last, data)
```

The url for the blockchain is

```
ws://com402.epfl.ch:7003
```

And the genesis-id (which corresponds to the skipchain-id):

```
a902de6ecd1a61cd8d7a456e18406e559898e530524ed2ba1422077a9d705c21
```

Mining

To mine a block, you need to create binary data that conforms to some sha256-value. The data-format is:

```
nonce = data[0:32]
hash_of_last_block = data[32:64]
full_email = data[64:]
```

You find the hash of the last block in the return-value of getBlocks. Only the last block is important. To create a new block, you can use the *createNextBlock* method and give it your data.

For the data to be accepted by the server, the first three bytes of the sha256-hash of the data need to be 0! This is what the nonce is for. You either use it as a counter or take random data (both should give the same result) and iterate, creating new sha256-hashes, until you find one hash that gives 0x000000 in the beginning. With my simple sample-script it takes about 4 minutes, so don't worry! For testing-reasons you can start with 0x0000 to see if your method works. Even though it will be rejected by the blockchain...

If you find a better way than to try out different combinations, be sure to write a paper about it ;)

Adding the block

Once you found a correct data for your block, create a block with *createNextBlock* and submit it to the blockchain with *storeBlock*. Be sure to check the return-message. In case the block is not accepted, you will get an error-message.

You can also check on http://com402.epfl.ch/static/conode.html for error-messages from the nodes.

Finished

You need to add at least 3 blocks to the blockchain to get your token from the website.

Exercise 3: Just In Time

In this exercise, you're being asked to guess credentials on the website.

To login, you must send a POST request with a JSON body looking like:

```
{ "email": <youremail>, "token": <yourguessedtoken> }
```

to

```
http://com402.epfl.ch/hw5/ex3
```

Don’t try to brute force the token. There are much faster ways to guess the correct token.

For example, the developer here used a modified function to compare strings which express some very specific timing behavior for each valid character in the submitted token…

The response code is 500 when the token is invalid and 200 when the token is valid. Look at the body of the response, you can get some useful information too. The “real” token that you can submit to moodle will be in the response body.

This exercise will require some patience and trial-and-error, as time in networks is never 100% accurate. In order to be precise, you should calibrate your measurements first, before trying to do any guessing on the token.

Don’t forget to have fun and good luck !

Exercise 4: PIR with a twist

In this exercise you will implement a very simplified version of the Riffle protocol (https://people.csail.mit.edu/devadas/pubs/riffle.pdf). Riffle is a communication system with strong anonymity, for all the detail please read the paper. The emphasis in the exercise will be on the PIR (Private Information Retrieval) phase of the protocol. The original protocol consists of 2 phases: the setup phase where clients and servers exchange cryptographic keys and the communication phase where clients upload/download messages from the servers. In our simplified case, we will have the following steps in the protocol:

- **Setup phase**:
  - Servers generate their public/private key pairs and start listening for UDP packets on a given ip and port
  - Client, let’s call it *Com402Client,* first generates a shared key (AES key) for each of the servers
  - For each server, client encrypts the shared key with server’s public key and send it to the server
  - So, at the end of the phase we will have:
    - Let’s say *m* servers running, each of them having a shared key which is shared with *Com402Client*
    - *Com402Client* which has all the shared keys *sk1, sk2, …, skm*
- **Communication phase - Upload**:
  - *Com402Client* client generates *n* chat messages and onion-encrypts them with the shared keys. Onion encryption of a message *msg* means the following:
    cipher = AES_sk1(AES_sk2(...(AES_skm(msg))...))
  - *Com402Client* then sends *n* onion-encrypted messages to the primary server
  - Servers then onion-decrypt all the messages:

        - Primary server removes one layer of encryption - decrypts the ciphertext with his shared key, and send the result to the next server in chain
        - Next server does the same, until eventually the last server in the chain obtains the plaintext message.
        - The last server then broadcasts the plaintext msg and all the servers store it.
    - Order of onion-encryption/decryption is determined by the servers ids. Each server has an assigned integer *id*, and the chain is defined by the increasing order of ids (server0 -> server1 ->...).
    - So, at the end of this phase all the servers have *n* plaintext chat messages. All the messages have an index, so the servers have them in the same order as well.
- **Communication phase - Download**:
    - In this phase a client, let's call *StudentClient*, want to download a message with a specific index, but doesn't want servers to learn which index is that. So, *StudentClient* performs a simple PIR.
    - *StudentClient* generates *m* bitmasks(one for each server) of length *n* (total number of messages), in such way that when xored, the result will be the target index (index of a message which *StudentClient* wants to retrieve).
    - *StudentClient* sends bitmasks to the servers and asks each server to select messages according to the bitmask, XOR the selected messages and send the result back
    - *StudentClient* can the recover the desired message by combining all the responses from the servers.
    - If this is not clear to you, please read the PIR phase in the paper!

We provide you with two skeleton scripts, one for the servers and one for the clients. You can download them on the com402 website. In the scripts you will find detailed instructions on what and how you should implement. We will test and grade two things:

1. Your server code: once you finish your implementation upload your script to com402
2. Your client code: we will grade only the client which performs the PIR

Update: When submitting your solutions, please submit your server and client separately (i.e. submit your server and get a token for it, then submit your client and get a second token for it)

You are provided with enough code to make the whole system work locally on your machine, so please test thoroughly before submitting the solution.

Note: you will have to install the pycrypto package by:

```
pip3 install pycrypto
```

Good luck!