Assignment 1
Secure Data Management

Rick van Galen - s0167746 (UT)
Erwin Middelesch - s0197106 (UT)
Jeroen Senden - s0191213 (UT)
Bas Stottelaar - s0199141 (UT)

November 28, 2013

1 Introduction

In this report, we describe our implementation for a Public Health Record (PHR) system. This system is implemented in Python and consists of a client and server. The role of the server is to announce the public parameters and store encrypted data while the client takes care of the encryption and decryption of data.

Our solution is based on a Ciphertext-Policy Attribute-Based Encryption (CP-ABE) scheme and uses no classic access control to authenticate users. Different categories of data can be read and written by different parties or sub parties.

First, we introduce the data model, including the categories of data and different parties. Then, we introduce the access policies. The third section will discuss the actual implementation and verifies the the requirements. The last section will discuss our implementation.

2 Data model

2.1 Data categories

The system defines the categories below.

Patient data In this segment of the health record, we store personal information of the user.

Health data This is the data inserted by a hospital where the user has been treated.

Training data This section contains training related data stored by the health club.

2.2 Parties

The following parties are defined and assumed to work with the system.

Patient The subject of the electronic health system. Each health record is associated with one patient and should be under full control of the patient. The patient has full read and write access to every category of data.

Doctor A patient's doctor treats the patient with health issues. The doctor may require read access to any part of a patient's health record. Therefore, the patient may choose to give the doctor read access to his personal information, health data and training data.

Insurance Health insurance companies may want to read a patient's records for fee calculations or health verification. A patient may give insurance companies read access to all of the segments of the his/her health record.

Employer Like the insurance company, an employer may need access to an employee's health information and may be granted read access.

Hospital Hospitals are given read and write access to the health data of the health record. An additional restriction is that a hospital may only write to the health record when a patient has been treated at that hospital.

Health club The health club provides the user with additional health services and training programs. Because this information is relevant to the health record, the health club gets read and write access to the training related segment of a patients health record.

3 Access model

3.1 Policies

3.2 Phases

Volgensmij kan dit wel weg, zie hieronder waar het ookk uitgelegd word..

3.2.1 Setup

3.2.2 Keygen

3.2.3 Encrypt

3.2.4 Decrypt

4 Implementation

This section describes the practical implementation of the assignment.

4.1 Architecture

The implementation consist of two parts. The first part is a storage server which does nothing more than saving records and providing records. The client part is a web application for all parties, but in a real situation, we assume that each type of party would have access to a own terminal to access PHR's. In fact, if it would be possible to have the server provide a GUI if we could implement CP-ABE in pure JavaScript (so encryption can stay local). Figure 1 depicts this architecture.

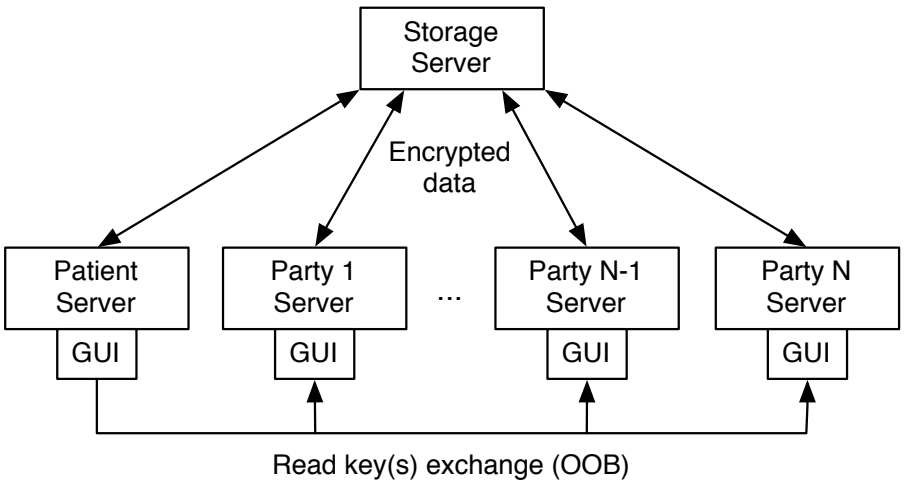

Figure 1: Overview of architecture where the patient and several parties exchange data.

All software is written in Python. To create web pages, the Django Framework[1] is used. The encryption is provided by the Charm Crypto[2] library.

When a patient wants to create a PHR, he queries the server for the public system parameters. The system parameters include the categories, the involved parties and the mappings. The mappings describe for which party the client will generate read key with what attributes. For example, a mapping from `PERSONAL => EMPLOYER, INSURANCE, HOSPITAL+A+B` will result in read keys based on the master key for the category personal data. Employer will receive the attribute `EMPLOYER` and the Insurance the attribute `INSURANCE`. Hospital-A and Hospital-B will both get their own attribute `A` and `B`, but also the shared attribute `HOSPITAL`. Keys should be distributed in a secure way, preferably out-of-band (OOB).

Encryption is done by specifying the plaintext, the category and the parties. Parties are unfolded to a policy. We use Python tuples and lists to define which parties have access and convert those to policy strings. Some examples (Python to policies):

- `[INSURANCE, DOCTOR]` → `(INSURANCE or DOCTOR)`
- `(INSURANCE, DOCTOR)` → `(INSURANCE and DOCTOR)`
- `[HOSPITAL, DOCTOR]` → `(HOSPITAL and A) or (HOSPITAL and B) or (HOSPITAL and C) or DOCTOR`
- `[(HOSPITAL-A, HOSPITAL-B), INSURANCE]` → `((HOSPITAL and A) and (HOSPITAL and B)) or INSURANCE`

This allows us to build policies much more dynamic and coupled with our protocol. Our implementation validates all categories and parties specified during encryption. This means one cannot specify parties or categories not in the system. A cipher text contains a little bit of meta data that can be read without decrypting. This way, a receiving party knows which category the cipher text belongs to and thus which read key(s) to use.

Sharing a write key is similar to encrypting data. There is only one write key per category and this key is generated during the creation of a PHR. The patient grants another party write access by encrypting the write key with the read key of the party or parties and uploads it to the server. Then, the party who will receive a key, retrieves this key from the server and decrypts it. The write key is added to the key chain. Now, the party can encrypt data.

In short, one can consider our implementation as a strict extension layer between the actual CP-ABE scheme and the user.

[1] `https://www.djangoproject.com/`
[2] `http://www.charm-crypto.com/`

All relevant parts[3] of the code is documented and tested. We refer to the source code for an in-depth overview.

4.2 Database model

The role of the database is minimal. This also resulted in a very small database model. The model is flexible, so we are not limited to any number of categories or parties.

Record table *ID, Record name*

Record item table *ID, Record ID, Category, Data*

Keystore table *ID, Record ID, Category, Data*

Data is saved in Base64 encoded string, representing the cipher text.

4.3 Verification of requirements

The assignment considers the following requirements:

1. A patient can insert personal health data into his *own* record.
2. A patient can provide his doctor, his insurance, and his employer with read access to (parts of) his health record.
3. A hospital can insert patient health data for any patient that has been treated by *that* hospital.
4. A health club can insert training-related health data to any patient record that is a member of *that* club.

To show that our system fulfills the requirements, we show a few use cases operated via the command line interface. One can play all use cases and should see similar results.

4.3.1 Creating a PHR

Since the patient is in control of his own PHR, he is responsible for creating one.

```
>>> python manage.py phr_create patient.json http://127.0.0.1:8000 "Patient record"
BEGIN SECRET READ FOR KEYS DOCTOR
(...truncated for clearance...)
END SECRET READ KEYS FOR DOCTOR

BEGIN SECRET READ FOR KEYS HOSPITAL-A
(...truncated for clearance...)
END SECRET READ KEYS FOR HOSPITAL-A

...

BEGIN SECRET READ FOR KEYS EMPLOYER
(...truncated for clearance...)
END SECRET READ KEYS FOR EMPLOYER

BEGIN SECRET READ FOR KEYS INSURANCE
(...truncated for clearance...)
END SECRET READ KEYS FOR INSURANCE
```

After running this command, all the secret reading keys for all the parties are generated. Each key has a bit of meta data encoded to connect to the right PHR. The mapping (see section XXX) defines who will get initial read access to which category of data. The patient should take care of distributing the keys, preferably via an out-of-band channel.

A patient can encrypt data for other parties even if the keys have not yet been distributed.

[3] All code not directly related to the Django framework.

4.3.2 Writing data

The patient decides to encrypt two messages. The first message can be read by his doctor, the second one by his insurance and his employer. They have access to the personal category of data.

```
>>> python manage.py phr_encrypt patient.json PERSONAL DOCTOR "Hi, Doctor!"
Uploaded to record item ID 55
>>> python manage.py phr_encrypt patient.json PERSONAL INSURANCE,EMPLOYER "Hi, Insurance
    and Employer!"
Uploaded to record item ID 56
```

The identifiers correspond to the items in the PHR of the patient. To verify that the patient has read access, he will decrypt the data he sent to the server.

```
>>> python manage.py phr_decrypt patient.json 55
Record item content:

Hi, Doctor
>>> python manage.py phr_decrypt patient.json 56
Record item content:

Hi, Insurance and Employer
```

4.3.3 Granting write access

By default, no other party has write access. The patient should grant write access for all parties or sub parties.

Let's assume the patient wants to grant Hospital-A write access.

```
>>> python manage.py phr_grant patient.json HEALTH HOSPITAL-A
Granted key for category HEALTH with key ID 22
```

The write key is encrypted for party Hospital-A. If the patient wanted, he could give all hospitals write access by specifying the Hospital party instead of Hospital-A.

4.3.4 Connecting to a PHR

In this step, the keys have been safely distributed to each party. We let each party connect to the PHR of the client.

```
>>> python manage.py phr_connect doctor.json http://127.0.0.1:8000
Paste the keys data, excluding the BEGIN and END block: (...paste key for doctor...)
Connected to record ID 28

...

>>> python manage.py phr_connect insurance.json http://127.0.0.1:8000
Paste the keys data, excluding the BEGIN and END block: (...paste key insurance...)
Connected to record ID 28
```

4.3.5 Reading data

In the first step, we encrypted two messages. One for only the doctor (#55), and one for both the insurance and employer (#56).

First the doctor:

```
>>> python manage.py phr_decrypt doctor.json 55
Record item content:

Hi, Doctor
>>> python manage.py phr_decrypt doctor.json 56
Cannot decrypt record item
```

Then the employer and insurance:

```
>>> python manage.py phr_decrypt employer.json 56
Record item content:

Hi, Insurance and Employer
>>> python manage.py phr_decrypt insurance.json 56
Record item content:

Hi, Insurance and Employer
>>> python manage.py phr_decrypt employer.json 55
Cannot decrypt record item
>>> python manage.py phr_decrypt insurance.json 55
Cannot decrypt record item
```

And to be sure, we verify the health club and hospital cannot read the messages (they have read access to different categories).

```
>>> python manage.py phr_decrypt healthclub-a.json 55
Cannot decrypt record item
>>> python manage.py phr_decrypt healthclub-a.json 56
Cannot decrypt record item
>>> python manage.py phr_decrypt hospital-a.json 55
Cannot decrypt record item
>>> python manage.py phr_decrypt hospital-a.json 56
Cannot decrypt record item
```

4.3.6 Retrieving write access

In use case 4.3.3 we provided Hospital-A with a write key. The party should retrieve the key from the server.

```
>>> python manage.py phr_retrieve hospital-a.json HEALTH
Key(s) imported for HEALTH.
```

The same does not work for other parties, e.g. Hospital-B and Health Club-A:

```
>>> python manage.py phr_retrieve hospital-b.json HEALTH
No new key(s) imported.
>>> python manage.py phr_retrieve healtclub-a.json HEALTH
No new key(s) imported.
```

Just to verify the keys are imported, one can issue the status command.

```
>>> python manage.py phr_status hospital-a.json
Host:        http://127.0.0.1:8000
Categories:  PERSONAL, HEALTH, TRAINING
Parties:     DOCTOR, INSURANCE, EMPLOYER, HOSPITAL+A+B+C, HEALTHCLUB+A+B+C
Mappings:    PERSONAL -> DOCTOR, INSURANCE, EMPLOYER
             TRAINING -> HEALTHCLUB
             HEALTH -> HOSPITAL

Record ID:   28
Record name: Patient record
Record role: HOSPITAL-A

Master keys: <not set>
Public keys: HEALTH
Secret keys: HOSPITAL-A -> HEALTH
```

For the above case, Hospital-A has connected to the right record, is able to decrypt messages for Hospital-A in the Health category and can encrypt messages for the Health category. Note that the master keys are not available, so Hospital-A cannot generate new keys (only the patient can do this).

4.3.7 Writing a reply

Now Hospital-A has write access, he can write data to the Health category. To verify this, we will write a message to the Health category only for the patient and one to all hospitals.

```
>>> python manage.py phr_encrypt hospital-a.json HEALTH HOSPITAL "Hello patient!""
Uploaded to record item ID 57
>>> python manage.py phr_encrypt hospital-a.json HEALTH HOSPITAL "Hello patient and
    other hospitals!""
Uploaded to record item ID 58
```

To conclude, each addressed party will decode it:

```
>>> python manage.py phr_decrypt hospital-a.json 57
Record item content:

Hello patient!
>>> python manage.py phr_decrypt hospital-b.json 57
Cannot decrypt record item
>>> python manage.py phr_decrypt hospital-c.json 57
Cannot decrypt record item
>>> python manage.py phr_decrypt hospital-a.json 58
Record item content:

Hello patient and other hospitals!
>>> python manage.py phr_decrypt hospital-b.json 58
Record item content:

Hello patient and other hospitals!

>>> python manage.py phr_decrypt hospital-c.json 58
Record item content:

Hello patient and other hospitals!
```

And the patient:

```
>>> python manage.py phr_decrypt patient.json 57
Record item content:

Hello patient!
>>> python manage.py phr_decrypt patient.json 58
Record item content:

Hello patient and other hospitals!
```

4.3.8 A second patient record

A patient is in control of his own PHR and has all the keys to read and write. In this use case, we show that a other patient cannot decrypt a record item that does not belong to him. Because the API to communicate with the server requires the user to specify a record item ID and the record ID, we modified it for this test to only require the record item ID. If we did not do this, the users would not be able to use the command line to request for a record item from a different record.

First, we create a second patient.

```
>>> python manage.py phr_create patient.json http://127.0.0.1:8000 "Patient two record"
BEGIN SECRET READ FOR KEYS DOCTOR
(...truncated for clearance...)
END SECRET READ KEYS FOR DOCTOR

BEGIN SECRET READ FOR KEYS HOSPITAL-A
(...truncated for clearance...)
END SECRET READ KEYS FOR HOSPITAL-A

...
```

```
BEGIN SECRET READ FOR KEYS EMPLOYER
(...truncated for clearance...)
END SECRET READ KEYS FOR EMPLOYER

BEGIN SECRET READ FOR KEYS INSURANCE
(...truncated for clearance...)
END SECRET READ KEYS FOR INSURANCE
```

Then, we connect the (same) doctor to that record, since patient one encrypted a message (#55) for him.

```
>>> python manage.py phr_connect doctor-patient-two.json http://127.0.0.1:8000
Paste the keys data, excluding the BEGIN and END block: (...paste key for doctor...)
Connected to record ID 29
```

It should be sufficient to show that both the patient two and the doctor with the other data file cannot read message from patient one.

```
>>> python manage.py phr_decrypt patient-two.json 55
Cannot decrypt record item
>>> python manage.py phr_decrypt doctor-patient-two.json 55
Cannot decrypt record item
```

4.3.9 Conclusion

We conclude this section with a table representing which requirement is fulfilled by which use case.

	Requirement 1	Requirement 2	Requirement 3	Requirement 4
Use case 4.3.1		X		
Use case 4.3.2	X			
Use case 4.3.3			X	X
Use case 4.3.4		X		
Use case 4.3.5		X		
Use case 4.3.6			X	X
Use case 4.3.7			X	X
Use case 4.3.8	X			

Table 1: Overview of which use cases cover which requirements.

5 Discussion

5.1 Authentication

Our implementation does not authenticate a party to the server and visa versa. This does not have any impact on the secrecy of the data, since only the once with the right keys, distributed via secure channels, can read or write data. The server cannot learn anything from the data. However, since anyone can upload data to our server, a lot of false and corrupt data may be stored in the database. A client cannot delete data, since the server does not know the client. The same holds for keys uploaded to the server. One data is sent, you should consider the cipher text compromised.

Since authentication is out of the scope of this assignment, we have not implemented a counter measure.

5.2 Number of keys

There are a lot of keys to be generated during the setup and keygen phase. There are N master keys and N public keys for N categories. Based on the mapping, each party having read access gets at least one read key. If the number of parties (and sub parties) grow, the number of read keys grows too.

Only the patient has to store all the keys. All other parties should only store their read and write keys.

5.3 Key sharing

For each category of data, there is only one master key and one write key. The write key is shared among the parties if they want write access. If a party is malicious, he could impersonate another party or spread the write key.

In the given assignment this is not a problem, since only write access is granted to the hospital and health club, both operating on a different category of data. It would be very easy to trace it back to a malicious party.

6 References

References