HIPAA ABE Read Me

Nipun Dave

June 21, 2011

1 Introduction

The goal of the SHARPS project is to identify the security and privacy challenges in the healthcare industry and to provide solutions for the same. This readme provides information on the installation and implementation for the following:

- Formalization of HIPAA law in Prolog
- Utilizing JLog to create an access control policy
- Performing attribute-based encryption of some data with the policy
- Decrypting the encrypted message using the attributes of the person accessing it
- Online Demo

All the files are available on the Github IHI account. Eric Lam can provide access to the github account. Once you have setup your github account and also setup the keys on your computer, the files can be downloaded using the following commands:

```
mkdir your_dir
cd your_dir
git init
git clone git@github.com:healthcareprivacy/IHI.git
```

Thereafter, create an eclipse project by importing the folders. Ensure that your eclipse has Java EE Developer Tools installed. We use three environment variables in the code :- HIPAA_HOME for the location of the HIPAA_ABE_Encryption project (for example ~/Desktop/IHI/HIPAA_ABE_Encryption)

and the other, ABE_LIB_LOCATION, specifying the location of the tools folder in the Functional Encryption library (for example: ~/Desktop/ABE_HIPAA/libfenc-read-only/tools). The final one ABE_DIR specifies the directory of ABE files (for example:~/Desktop/ABE) These are setup by using the following command:

```
export HIPAA_HOME=/path/to/the/projectfolder
export ABE_LIB_LOCATION =/path/to/the/library
export ABE_DIR =/path/to/ABEfolder
```

2 Formalization of HIPAA Law in Prolog

The HIPAA law states the conditions under which an entity can share someone's data with another entity. This section of the law has been formalized and expressed in Prolog according to the technique described in the paper by Peifun Eric Lam, John Mitchell and Sharada Sundaram. The prolog files are available on Google Code. It can be downloaded to the *dir1* folder using the following command:

```
svn checkout http://hipaa.googlecode.com/svn/trunk/ dir1
```

Each statement of the law is represented by the eight-tuple:

$$a = (Sender, Receiver, Owner, Purpose, Type, Message, Belief, Consent)$$

For our application, the type element has a constant value of *PHI* i.e. Personal Health Information. The other seven elements vary. We refer to the PHI data as a message in the rest of the document.

The eight-tuple statement can be broken up into two parts: *HIPAAQuery* which contains elements specified during the sharing process and *HIPAAResult* which specifies who can legally access the shared message. The *HIPAAQuery* is a 3-tuple consisting of *(Sender, Owner, Purpose)* while the *HIPAAResult* consists of *(Receiver, Belief, Consent)*.

3 Creating the Access Control Policy

JLog is a Prolog interpreter written in Java. The Jlog library provides an embedded Prolog engine in Java. It enables us to consult prolog code, construct queries and evaluate query results. It also provides the translation between the Prolog terms and standard Java objects. We are using this library to generate the access control policy. It can be downloaded from

http://jlogic.sourceforge.net/. We have combined all the HIPAA prolog files into one file *HIPAAPolicy.pl* which is available in the ABEPolicyGeneration project. This is because JLog (v1.3.6) could not process prolog code spread across multiple files. We have also deleted the lines which resulted in an output in the terminal.

For each value of *HIPAAQuery*, there is a set A of *HIPAAResult*s such that each combination of *HIPAAQuery* and a member of this set A is allowed according to the HIPAA law. For each possible value of *HIPAAQuery*, we repeatedly query the prolog code using JLog to obtain a valid *HIPAAResult* until there are no more. We store this in a Java object of the format $HashMap < HIPAAQuery.toString(), Set < HIPAAResult >>$. We are using the String version of HIPAAQuery as it can be quickly used for comparison. The policy generation is done in the project *ABEPolicyGeneration* in the class *GenerateHIPAAPolicy.java*. The resulting Java object is serialized and stored in a file and is made available for access later. The *ReadHIPAApolicy.java* class provides the sample code for unserializing the file and thereafter querying the resulting Java object.

The person sharing the message specifies the values for the 3 elements of *HIPAAQuery*. Using those values, we query the HashMap object to obtain the set of legal *HIPAAResults*. The person accessing the message provides the values for the elements in the *HIPAAResult* tuple. The individual can access the message only if the tuple of 3 values provided matches one of the elements in the set. This is the basic concept of the access control that satisfies HIPAA law.

We are enforcing this access control using Attribute-based Encryption as described in the next section. This allows us to encrypt a particular message and publish it publicly. Only those who have the attributes that allow them to legally access the message can decrypt it. This is one way of distributing the access control.

4 Attribute-Based Encryption

The Functional Encryption library *libfenc* is a general purpose framework for implementing *functional encryption* schemes. It is a generalization for a wide range of encryption technologies including Attribute-Based Encryption (ABE), Identity-Based Encryption and others. It is available here. This needs to be setup for the website demo to successfully run.

5 Website

The website provides a demonstration of enforcing access control according to the HIPAA law using Attribute-Based Encryption. The front-end has Javascript code (*messageDemoEncrypt.jsp*) to acquire the values of the parameters and then transmit them to the server. It also displays the results obtained from the server. At the back-end, the Java class *ProcessInput.java* is a servlet that receives the values of the parameters and performs some computation depending on the values received. The Functional Encryption library has been developed in C. We have written the interface with this library to perform key-management, encryption and decryption in Java. These processes require a lot of computational power. Consequently, they are all executed at the back-end which is running an Apache Tomcat server.

We demonstrate the entire process from encryption to decryption on the website. As mentioned before, the person sharing the message provides the values for the parameters *Sender, Owner* and *Purpose*. The message is first encrypted using symmetric DES encryption as that is quick. The key used in the DES encryption is thereafter encrypted using the policy corresponding to the values given.

5.1 Demo

The workflow of the demo is:

1. Documents are encrypted using DES encryption. We currently use DES encryption purely for convenience reasons. The demo could easily be modified to use AES or any other form of symmetric encryption.

2. We then encrypt the DES key of a document generated from the previous step using ABE. The attribute inputs and the boolean conditions between them is provided by the prolog implementation.

3. Each user receives a key generated on the basis of his attributes which allows him to access the documents that he is entitled to. This key is generated with the help of the ABE master key. Thus if two of more hospitals want to use ABE based encryption to exchange information amongst themselves, they need to use the same ABE master key.

4. When the user wishes to access a particular document, we use his private ABE key to decrypt the ABE encrypted DES key of the document. If the decryption is successful, the decrypted DES is key is used to decrypt the man document. The information contained in the decrypted document is then displayed to the user.

5.2 Code Design and Structure

Access Controller: This is the main class of the demo, which allows the user to access the encrypted documents. This class assumes that an ABE based private key exists for the user. Thus once we specify the key path and the path of the encrypted document the object instance will try to decrypt the concerned document.
ABEKeyGenerator: This class generates the ABE key for a user, based on his attributes. The attributes of the user need to be specified in a list format, using comma separated values. We use the Libfenc library for the purpose of key generation.
ABE Encrypter: As the name suggests this class encrypts the DES key of the document based on the attribute policy specified by HIPPAA. We get these policies by running the prolog code and specifying the values of the parameters: Sender, Owner and Purpose. We then receive a list of tuples of allowable parameter values for the receiver, comment and belief parameters. Each tuple of the list specifies not only the attribute values but the boolean conditions between them. Because of the structure of the prolog code the parameter values in a single tuple are anded while those across multiple tuples are ored. The following example makes the above statement clear:
TODO GIVE EXAMPLE: The ABE encryptor object stores the encrypted DES key at the provided path
DES Encrypter Decrypter This class uses DES encryption for encrypting the main file which contains the actual medical records. The functionality of this class is primarily hidden from the user of the system. The objective of this class is to provide a level of indirection which makes the overall system more efficient. The intuition behind this design is that, access control policy for a document can change frequently. Thus, given the size of these files, it would be inefficient to re-encrypt the entire document again and again everytime the policy changes. By introducing DES encryption we only need to re-encrypt the DES key which is of a very small size. Similar to the ABE encryptor, this class allows you set the path where the encrypted DES file should be stored. The Decrypter section of this class picks up the DES key from the specified location and decrypts the corresponding document and returns the decrypted information.
Utility Classes: The code also contains several utility classes which made the entire framework of the demo more generic and flexible. These classes include a Command Class for executing bash commands for ABE encryption/Decryption. We also have a HIPAAFormatter class which returns the policy structure we use for encryption based on the tuple values obtained from the prolog code. There are also various wrapper classes which further

simplify the implementation of the various features of the above mentioned core classes.

Demo Specifics While coding up the ABE encryption module, we realized that there is a character limit to the attribute policy (which is passed as a parameter of data type String) we provide to the libfenc function. We found that accommodating 8 tuples of the serializable object in a single encryption is possible. Thus we generate multiple encrypted copies the DES key based on subsequent sets of eight or less tuples. This variation is incorporated in the ABE-Encrypter class which returns a count of the number of encrypted versions that were created for a particular DES key. During the decryption phase we need to specify the number of encrypted versions for the DES key. The decrypted will then use the user's private key to sequentially decrypt the above encrypted versions. As soon as the ABE Decrypter is able to successfully decrypt any one of the encrypted versions, it passes the Decrypted DES key to the DESEncrypterdecrypter to decrypt the actual document.

6 Trouble Shooting

If you run into errors while importing a project (i.e.ABEPolicyGeneration), it could be that the external JAR paths are not recognized. To solve this. In the Package Explorer in Eclipse, right click on the Project (i.e. ABEPolicyGeneration) and go to Properties. Go to Java Build Path, and then Libraries. Remove all the .jar files and choose Add External JARs and add the .jar files in the lib directory of your project (i.e. ABEPolicyGeneration/lib).