LINGI2144 - Secured Systems Engineering
RFID Business Card

O. Goletti, L. Lamouline, T. Leruitte

May 12, 2011

Contents

In this report, after the installation instruction, we will present the general architecture of our software. Then we will describe each layer one by one. In these descriptions will be discussed several points such as the implementation choices, the efficiency and the possible improvements. Finally, we will have a discussion about the security analysis of our whole solution.

1 Installing the program

Installing our program is really simple. First, you have to unzip the program.zip archive, and you must make the newly created `src` folder your working directory. Then the command `./install.sh` will install our whole program. The usage of our program will be explained along this report.

2 Architecture

Our software is organized in three layers. A graphical representation is provided in Figure 1. The main advantage of this structure is that each layer is usable separately, if the lower layers are presents, of course.

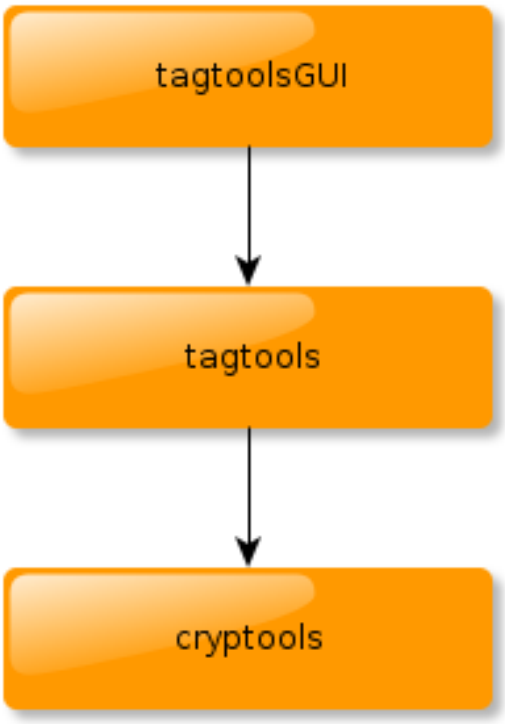

Figure 1: Arhitecture in three layers

`cryptools` is the lowest layer of our application. It is written in `C`. This module implements the cryptographic part of our application. `tagtools` takes care of the communication between the tag and our application through the use of the reader. This module is written in `Python`. The top layer, `tagtoolsgui`, is the GUI of our application. This part is written in `Java`. In the following, each layer will be described in details in a bottom to top fashion.

3 cryptools

So, let us begin with `cryptools`. This layer provides cryptographic features. It allows an user to use the cipher-block chaining (*CBC*) mode of operation. Such an operation mode allows one to repeatedly and securely use a block cipher. It is possible to use CBC either in *send mode* or in *receive mode* and therefore `cryptools` enable the use of those two modes.

3.1 Implementation

This layer provides the implementation of two block ciphers (based on [FIP]) : Data Encryption Standard (*DES*) and Triple Data Encryption Algorithm (*TDEA*). Both are usable either in encryption or decryption.

Figure 2 is a graphical representation of the architecture of this layer. The goal of this design was to separate each module and to have the possibility to use it separately. Indeed, the `cbc` module could be used with any block cypher with the the right signature, i.e. the one provided in the `cbc.h` file :

```
void (*f)(char binInput[], char binKey[], char binOutput[])
```

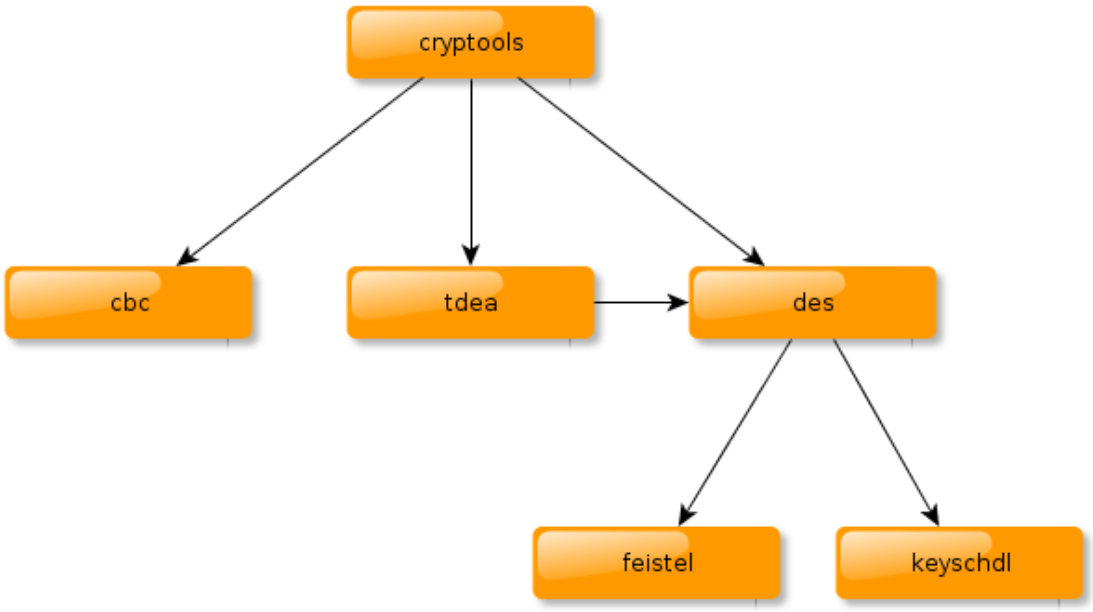

Figure 2: Arhitecture of cryptools

The `feistel` and the `keyschdl` modules are the core implementation of DES. They respectively take care of the `f` function of DES, which is a feistel network, and of the scheduling of the keys required in this `f` function.

The `des` module implements the DES block cypher itself. As said before, this implementation is based on [FIP]. The main difficulty while using this standard was to know the order in which the bits was expressed. We did not know whether the LSB was the first or the last one. Eventually, we had to test our implementation and to compare the results with the SSL ones (the answer is LSB first). This module is based on the two preceding ones which implement the main tools of the DES algorithm. Indeed, in this package is also provided the inverse algorithm for deciphering.

We can note here that the signature of the two functions in this module are compliant with the one used in the CBC implementation :

```
void des(char binInput[], char binKey[], char binOutput[]);
void desInv(char binInput[], char binKey[], char binOutput[]);
```

The `tdea` module is based on the `des` one since the algorithm is the sequential use of a DES encryption, a DES decryption and again a DES encryption. This module also provides the deciphering algorithm and the two signatures are again compatible with the one required by our CBC implementation :

```
void tdea(char binInput[], char binKey[], char binOutput[]);
void tdeaInv(char binInput[], char binKey[], char binOutput[]);
```

Before talking of the `cryptools` layer itself, let us first discuss the `cbc` one. The main advantage of this module is that it is generic. It means that it could use any function following the required signature in either send or receive mode. This is very convenient since `OpenSSL` does not provide this latitude (which is one of the reason for which we had to do our own implementation).

Now the `cryptools` layer itself is just a wrapper of the functions provided by the lower levels. It allows the user to make an easy use of them through a CLI interface. For the details of the use of the tool, we advise the reader to go to the readme subsection of this module.

3.2 Usage

`cryptools` is located in the `cryptools` folder.

The usage of this tool is quite simple. Here is provided the `help` of it :

```
Usage : cryptools [size] | -b [des|3des] | [-d|-e] | -h | -m [snd|rcv]
    [size] is the size of the plain text to give to stdin
    -b [des|3des] allows the user to choose its block-cipher                    3
        des selects the implementation of DES (default)
        3des selects the implementation of TDEA
    [-d|-e] enables respectively the decryption or the encryption (default)
    -h displays this help
    -m [snd|rcv] allows the user to choose its CBC operation mode               8
        snd selects the CBC send operation mode (default)
        rcv selects the CBC receive operation mode
cryptools then waits for two inputs on stdin separated by an "end of line"
character. The first one has to be a string representing the hexadecimal
encoding of the key (i.e. 16 char if the algorithm is DES, 48 if it is TDEA. The   13
second one is the message to encode. It should also be a string representing its
hexadecimal value and this string is of length size*2.
```

3.3 Security

We made some efforts in order to keep our tool as secure as possible. The first one is that we made a scrupulous implementation of the standard. This is quite important since we know that the security of this block cipher is mainly based on the *S-boxes* and so the smallest mistake could lead to a insecure algorithm.

The second security point is that in order to require the key and respectively the plain text to encrypt or the cipher text to decrypt as arguments, the tool retrieves them from `stdin`. The advantage of this usage is to avoid them to appear in a command line and to leave traces of those critical informations in the shell historic.

3.4 Efficiency

About the efficiency of our implementation. The main advantage is structural, i.e. it is quite easy to add a block cipher with the same signature as the one required by `cbc`. This means that anyone could easily extend our code to add its own implementation of another block cipher.

Compared with `OpenSSL`, the use of `cryptools` allows a user to enble either the send or the receive mode of CBC with any block cipher. Which is not allowed in `OpenSSL` in which encryption is always used in send mode while decryption is always used in receive mode.

We have compared the efficiency of our tool with the one of `openssl` and it appears that our implementation is the fastest :

	`cryptools`	`OpenSSL`
Time for 1000 computations [s]	9.987	45.127

3.5 Further work

The main drawback of our implementation is the way we represent binary words. Indeed, instead of using a `char` in order to represent 8 bits, we use arrays of `char`. The size of such an array is the size in bits of the binary word and the i^{th} `char` of the array is either $'1'$ or $'0'$ depending on the value of the i^{th} bit of the binary word.

We made the choice of using such a representation for the convenience of the access at a specific bit of a binary word. Indeed, instead of having to shift the word and to use a mask in order to recover one specific bit, we just have to access the required index in the array. Since many operations in the DES algorithm are permutations on a binary word, it seemed easier to represent them like that. Nevertheless, it seems obvious that such a choice slows down our implementation, hence an optimization of our application.

4 tagtools

4.1 Implementation

`tagtools` is actually composed of two programs: `tagw` and `tagr`. `tagr` is used to read the tag and `tagw` is used to write the tag.

4.1.1 `tagr`

`tagr` will first read the tag and show the business data defined by [AM]. It will then read the certificate and check that the certificate has been issued by the certificate authority. Finally, it will verify the signature of the data, and print the subject of the certificate.

4.1.2 `tagw`

`tagw` takes a bunch of arguments, which are explained in the help (`tagw -h`). Some of those are the data that will be written, the private key of the user, and the certificate of the public key, signed by the certificate authority.

When `tagw` is used with a formatted tag (the master key is 0x00), it will change the master key, the app master key, and the app write key. Those three keys will be encrypted with the public key of the user, and then written in the file represented by the argument `save`. (This file must not exist before.)

When `tagw` is used with an already written tag, the file containing the tag's keys must be provided. The file will be decrypted with the user's private key. The keys will be replaced by new ones. The new ones will be returned to the user the same way than explained before.

Besides, the tag keys generated during the writing are also logged unencrypted. This is a huge security issue, and thus our program must not be use in any context but academic one. We have decided to do so because if the user loose the save file, the master key of the tag can not be retrieved, and the tag become locked for life. Actually, this has happened during our tests, and the log has allowed us to unlock the tag.

We have designed our application in such a way that the user has always the control of his data. We have decided that a user should be able to write his own tag, and should read the tags of all others users. Therefore, a user shouldn't write the data of another user, and a user shouldn't give his private key to another user or admin. Besides, our application rest on the fact that each user has in own certificate, signed by the same certificate authority.

Finally, it is also possible to format a tag. In this case, the current tag keys and the private key of the user must be provided. The master key of the tag will be reset to 0x00.

4.1.3 Complete the data

We have decided not to implement the complete operation, in order to keep our program simple [Wik]. This operation is intended to write the signature and certificate on a tag, considering the data already present. This operation is however possible with the code listed below (see Section 4.2 for the meaning of the others arguments).

```
tagr -u --csv > data.csv
tagw tagkeys data.csv key cert save
```

4.1.4 Overall architecture

The overall architecture is represented in Figure 3. `tagr` and `tagw` use both two python classes: `DESFireCon` and `SSLWrapper`.

`DESFireCon` is responsible for the connection between the reader and the tag. Each of its method are the ADPU methods explained in the project document. `DESFireCon` uses

`pyscard`.

`SSLWrapper` is the wrapper for the functions of `OpenSSL` and `cryptools`. `OpenSSL` is only used to sign some data and to verify some signature or certificate. `cryptools` is used for all the cryptographic operations (DES and TDEA).

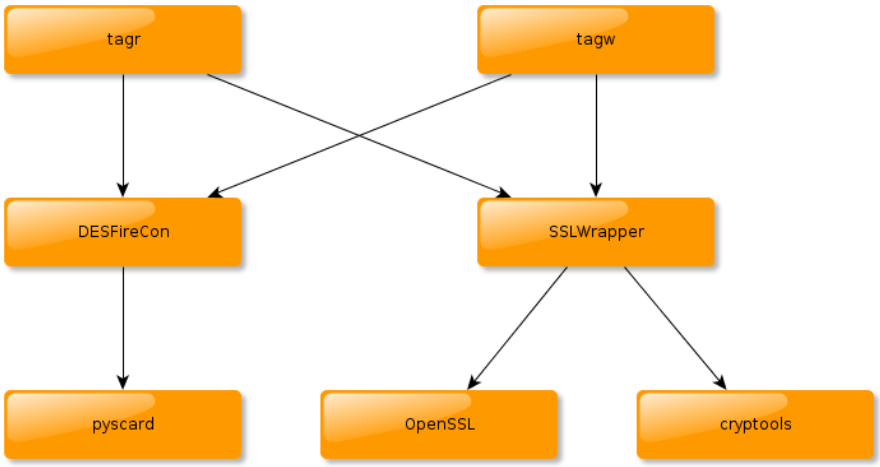

Figure 3: The tagtools architecture

4.2 Usage

`tagr` and `tagw` are located in the `tagtools` folder.

`tagr` is used to read the tag.

```
Usage: tagr [options]

Options:
  -h, --help      show this help message and exit
  --csv           print the result in the csv format.
  -u, --unsigned  only read the business data. Doesn't verify the signature.
  -v, --verbose   print the ADPU exchanged between the reader and the tag on
                  stdout.
```

`tagw` is used to write the tag.

```
Usage: tagw <options> tagkeys csv key cert save

  tagkeys: file where the keys has been saved during the last run of tagw.
  csv: file containing the business data to write on the tag (CVS format).
  key: file containing the private key which will be used to sign the data.
  cert: file containing the certificate of the public key.
  save: file in which will be saved the generated tag keys (encrypted with public
      key).

Options:
  -h, --help      show this help message and exit
  -f, --format    format the card. The arguments tagkeys and key are the only
                  one which must be provided.
  -n, --new       init a new tag. The argument tagkeys must not be providded.
  -v, --verbose   print the ADPU exchanged between the reader and the tag on
                  stdout.
```

4.3 Security

Our implementation suffers from two security issues.

As a remainder, a log file of all the tag keys is generated (see section 4.1.2). But this is only done to avoid to kill a tag. It could be removed now, since we are pretty sure that we can handle the management of the master key. Nevertheless, in the academic context, we decided to keep it since it is easily removable.

Secondly, the private key and the public key are used to encrypt and decrypt the tag keys. However, the private and public keys are also used to sign and to verify the signature of the business data. The pair of public and private keys are thus used for two different things, and a pair of public and private keys should never be used for two different things.

Nevertheless, we have tried to optimize the security of or our implementation.

`SSLWrapper` must call `OpenSSL` and `cryptools`. We have tried not to put any sensitive information as the arguments of those calls, because others users would be able to see them. Instead, we use the standard input and output.

When `SSLWrapper` must create temporary file, for example to store the public key, we create those temporary file in a way such that only the user have the read and write rights on those files.

4.4 Efficiency

The reading and writing of the tag is the bottleneck of our program. In order to improve the performance, we have tried to minimize the data that should be read or written. We have especially cut the data contained in the certificate file, in order to write only the relevant part.

4.5 Further work

One major point to improve is the second issue pointed out in the security section (Section 4.3), i.e. the use of the same key pair for the signature and the encryption. We haven't had the time to correct this issue, but it would be quite straightforward. One would have to modify the argument in order to ask for an other key pair, and use this new pair to encrypt and decrypt the tag keys file.

5 tagtoolsgui

The GUI is the third layer of our software and is the interface between the user and the underlying layers previously mentioned.

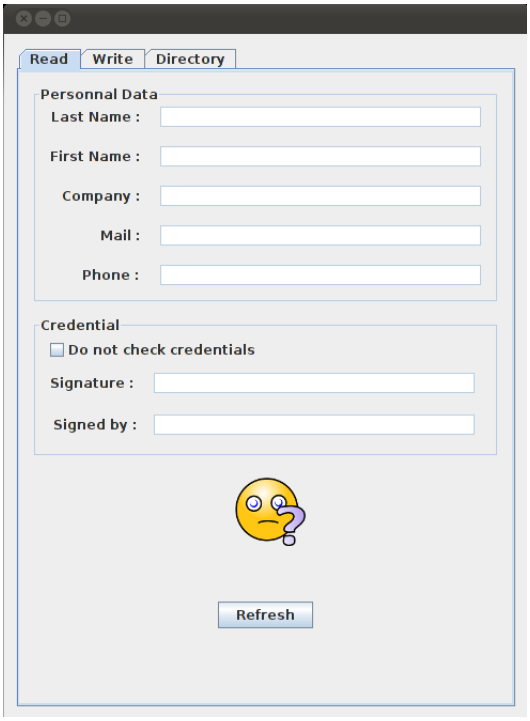

Figure 4: Read panel allowing to read tags.

5.1 Implementation

The graphical interface is composed of three main parts, one panel to read the data from tags, another one to write data on a tag and the last one is a directory summarizing all the information previously read from the tags.

The first one allows the user to read a tag thank to the `Refresh` button. This reading can be performed in two ways: by checking the credential or simply reading the content of the data part without checking the authenticity of these latter. The purpose of this second reading technique is to get the data part faster, for example, if we already know by a previous check that the data are signed with a certificate trusted by an authority.

The second part allow the user to write data on a card. This process can also be done in multiple ways: by providing the information about the owner of the card manually or by specifying a CSV file containing these data to the program.

With the first technique depicted in Fig.5, the user has to provide the personal information of the owner of the tag in the fields provided for this purpose. Furthermore, it also has to provide the files containing the keys of the tag (if it has not been formatted), its private key, its certificate and the destination file that will receive the encrypted tag keys. If the tag had been previously formatted, the user can check the "Formatted Tag" check box, so it does not need to provide the tag keys file since they have been set to 0 by default. This panel also allow the user to erase the data present on a tag thank to the format button. In order to format a tag, it has to provide the location of the tag keys file and the file containing its private key.

The second way to write data on a tag is to use a CSV file directly (see Fig.6). In the

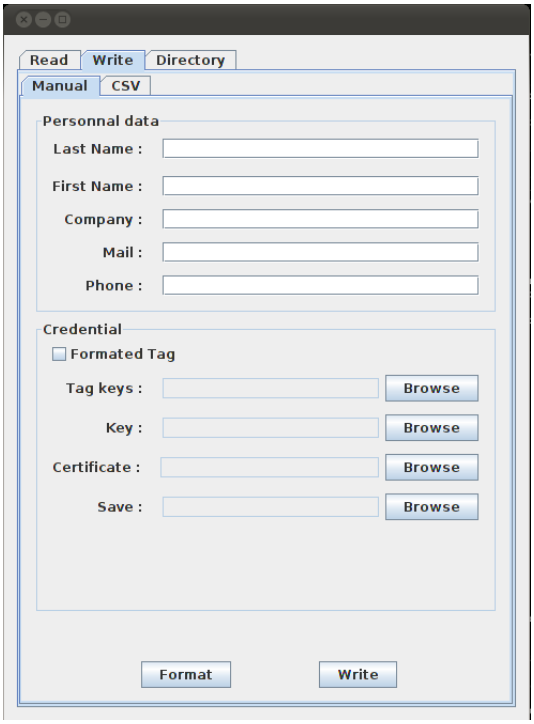

Figure 5: Panel used to write data manually on a tag.

second interface called "CSV" of the `Write` panel, the user firstly specify the path to the CSV file and then if the tag has not already been formatted, it also has to provide the same file than the previous method. For the same reasons as before, the tag keys file is not required if the tag has been previously formatted.

The third main part of this interface is the `Directory` (see Fig.7). As its name imply, it contains the information extracted from all the current session reads. It allows the user to see the data related to all the tags previously read. Only data belonging to tags whose signature has been verified are added in this directory, it is a way to avoid keeping uncertified data. This list is implemented such that two same entries (all personal data are the same) are recorded only once. That is, if the second one has the same information, it is ignored. The list can be managed in the sense that the user can choose to remove some entry if needed. It just has to select the one to remove and click one the `Remove` button.

5.2 Usage

To run the GUI, you have to be in the `tagtoolsgui` directory and simply execute the script `tagtoolgui`.

5.3 Security

The only sensitive part of this layer is located in the writing technique. As we have to provide a CSV file to the underneath layer (the layer communicating with the reader

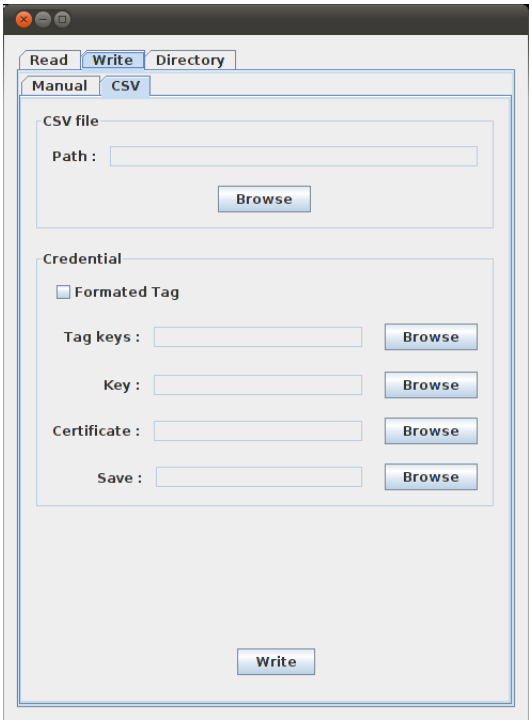

Figure 6: Panel used to write data thank to a CSV file.

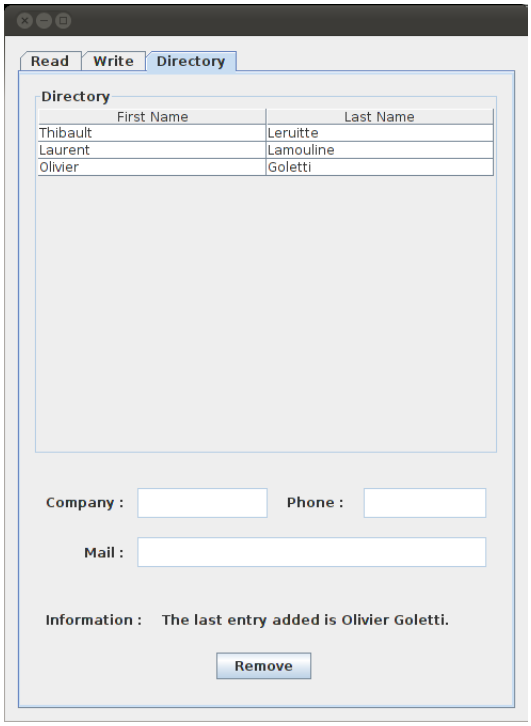

Figure 7: Directory containing information about previously read tags.

implemented in python) when we want to write data on the tag, when the user provide manually the information of the owner, this file must be created temporarily. It is created as a temporary file and is erased immediately after the call. In this way, personal information of the owner never stays more than the writing time (less than a second) on the computer.

5.4 Further work

It is still possible to improve the GUI to make it more user friendly. We try to make it more intuitive as we can but some improvements can still be made. Especially in the `Write` panel. When the user wants to format a tag, it only has to specify the `Key` and the `Tag Keys` files but it's not really clear. Even if a small message indicates it on the interface in case of bad manipulation.

An other improvement can be made to enforce the security when removing the temporary file created when a user write on a tag with the manual method. Instead of simply removing it, it could be more secure to write some random data several time in it and then remove the file. In this way, once the file is removed, it is pretty impossible to retrieve it.

The last graphical improvement that can be done is to correct a bug in the display when we remove the tag before the program has read it. But we have not enough time to remedy to this.

6 Security analysis

In this section, we will discuss four main points widely studied in a classical security analysis :

Impersonation is the stealth of the credentials of an user of a system.

Information leakage : this point concerns the unwilling disclosure and spreading of information about an user of a system.

Denial of service (*DOS*) : the security problem in this category are raised by an attempt of the adversary to make a system anavailable.

Malicious tracability is the fact that through the use of a system, the user is made recognisable at any time.

6.1 Impersonation

There are severals attacks that are possible in the impersonation point of view.

First, an adversary can write any information on a tag, giving that she signs the information with a valid certificate. In order to prevent this attack, we print the subject of the certificate (i.e. the name of the owner of the certificate) used to sign the data. The user has to verify that the subject of the certificate correspond to the name in the data.

Secondly, the adversary can create a fake certificate, in order to sign the data. We prevent this attack by checking that the certificate present on the tag has been issued by the certificate authority.

Finally, since the tag is completely and freely readable, so it is possible copy it. Nevertheless, we don't consider this as an impersonation attack, since a tag can't be used as an authentication mean. Indeed, like a paper business card, the tag only carry something that we own. An authentication could be done if something that we are, or something that we know, are also provided. However, we will consider the copy attack in the information leakage (Section 6.2) and in the malicious traceability (Section 6.4) sections.

6.2 Information leakage

The main point about our application is that the goal is precisely to spread information about the user (since it is a business card). So the information leakage is inherent to the application. Nevertheless, an adversary could produce a copy of a tag and spread it wherever she wants to. But this issue is the same with a classical business card.

6.3 DOS

A DOS is always possible in the physical layer, which tools such a blugger. Nevertheless, such attacks are related on the technologie used, and thus cannnot be avoided.

6.4 Malicious traceability

In our case, since no log are stored on the card, it is not possible to track someone by exploiting this kind of file. The principle of our application is to identify someone, so it is possible to read the data contained on the tag whenever the owner wants. If the owner of the business tag is in the field of a reader, it can be identified.

Another way to identify a business card is to get the UID of the card that is unique. So, if an adversary can monitor several readers, she can track a business card since it has a unique UID.

If the owner of the card does not want to be authenticated unwittingly, it has to protect its card in a metallic pocket for example.

7 Conclusion

At the end of this project, we have build a program that meet the requirements. We are now able to read and write tags.

It was interesting to see the implementation of a standard and to see the impact of taking into account in an application the security aspects.

It seems we have understood the assignment in a different way that most of the other groups since we did not use the "admin-user" architecture. This choice of implementation

was motivated in this report and base the security of the all application on the certificates. But nevertheless, it remains compliant with the assignment.

We are now aware of the different aspects to take care of in such an implementation and are ready to do so in a company and to distribute our software, given some minor modifications discussed along this report. Under free license of course.

References

[AM] Gildas Avoine and Tania Martin. Rfid business card - encoding format.

[FIP] PUB FIPS. 46-3. *Data Encryption Standard (DES)*, 25.

[Wik] Wikipédia. Kiss principle. `http://en.wikipedia.org/wiki/KISS_principle`.