

INGI2144 Project (2011-2012)  
RFID Loyalty Card for the LLN Sandwich Shops

Nicolas Maître, Bernard Paulus, Arnaud Theismann

December 12, 2011



---

**UCL**  
**Université  
catholique  
de Louvain**

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Software Implementation</b>	<b>2</b>
2.1	Architecture . . . . .	2
2.2	External tools . . . . .	3
2.3	Protection against human mistakes and bugs . . . . .	3
<b>3</b>	<b>Security analysis</b>	<b>4</b>
3.1	Actual weaknesses of the system . . . . .	4
3.2	Towards a better solution . . . . .	5
3.2.1	With same technology and file/data structure . . . . .	5
3.2.2	With same technology only . . . . .	5
<b>4</b>	<b>Conclusion</b>	<b>5</b>

# 1 Introduction

The aim of this project was to design and implement a RFID-based system able to manage electronic loyalty cards for sandwich vendors in Louvain-La-Neuve.

In this document, we first present the architecture of our software and an overview of its implementation, then we give a security analysis of the whole system, along with ideas to improve it.

## 2 Software Implementation

### 2.1 Architecture

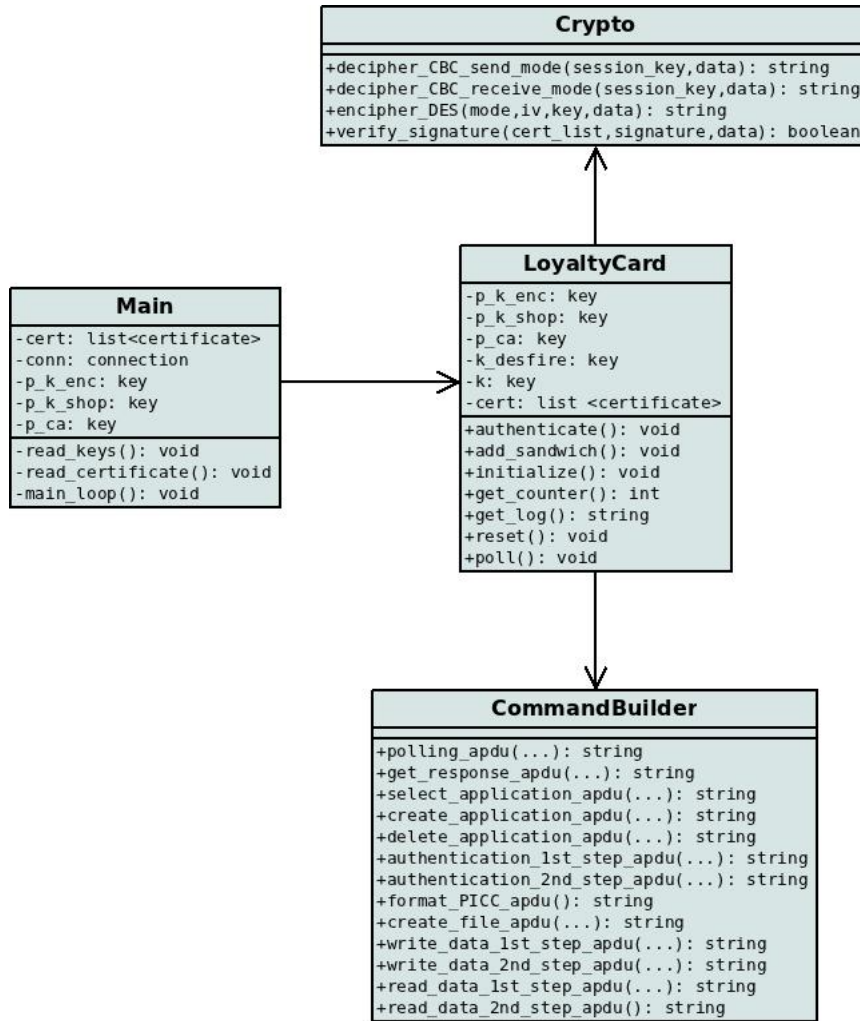


Figure 1: Architecture of the software

As depicted on figure 1, our python application is divided in 4 modules. **LoyaltyCard** class aims at representing an RFID loyalty card and offers methods to interact with.

**Main** is responsible to create and initialize instances of **LoyaltyCard** and offers to the user a command prompt to interact with the DesFire tag. Moreover, it is up to this module to load from different files the keys and certificates needed for the management of a loyalty card.

The **Crypto** module offers high-level cryptographic methods that are used by the **LoyaltyCard**

module. These methods essentially call other methods from external crypto libraries (see next section).

Finally, **CommandBuilder** is a low-level module that allows to build APDUs to communicate with the tag.

## 2.2 External tools

In order to communicate with the RFID card, we have used **pyscard** (<http://pyscard.sourceforge.net>) which is a python wrapper for pcsc-lite.

**PyCrypto** library (<https://www.dlitz.net/software/pycrypto>) has been used as it provides methods for dealing with DES/3DES and RSA encryption and decryption. This python library is very interesting from a performance point-of-view since speed-critical operations like ciphers and hash functions are actually written in C.

In addition to this, since PyCrypto is not able to read certificates in X.509 format, we have used **M2Crypto** (<http://chandlerproject.org/bin/view/Projects/MeTooCrypto>) for this purpose. Retrieving public keys from certificates and verifying signatures is therefore handled thanks to this library which is in fact a wrapper for OpenSSL.

## 2.3 Protection against human mistakes and bugs

A few measures have been implemented to deal with bugs and human mistakes. First, it is impossible for the user to crash the application by performing a command when there is no RFID tag in the reader's field. Indeed, the presence of a tag is checked after every user input. If no tag is detected, a timer of 3 seconds is launched at the end of which a warning message is printed to the standard output. No further operation can be made as long as the method `poll()` does not return.

Similarly, the presence of an RFID reader is checked at application start-up. The command prompt is enabled only after the system has detected a reader. The presence of a reader is then checked every second to prevent the user from performing an operation when there is no reader. If such a case occurs, a warning message is printed to the standard output.

Finally, we ask confirmation when the user wants to reset the loyalty card to factory settings. This prevents him from erasing the whole tag memory by mistake.

## 3 Security analysis

### 3.1 Actual weaknesses of the system

A first weakness is in the way we manage the symmetric key  $K_{DESFire}$ . Since this key needs to be the same for each card of a given shop, we can't generate it in a random way; the value of this key is hard-coded in the source code of the application. If an attacker were able to obtain the source code of the application, he would be able to recover this key and then erase the whole memory of a tag.

Similarly, the way asymmetric keys  $P_{enc}$ ,  $K_{enc}$ ,  $P_{shop}$  and  $K_{shop}$  are stored may put the system at risk. Indeed, these keys are stored in clear so as to be read by the application. Thus, each sandwich store where the application is deployed is a potential place where the keys can be stolen by an attacker.

Another weakness is the fact that the file containing the counter of bought sandwiches is free readable. That means that any person with a reader can determine whether a tag owner is a good client or not. This weakness may be exploited by competitors for example to target new potential clients.

An even more serious problem may be the fact that files containing the encryption  $E$  of key  $K$  and the signature  $S$  of this encryption are free readable as well. That means that an attacker would be able to obtain this data (by **skimming** the tag or **eavesdropping** the channel for example), clone it and then pretend owning a loyalty card from a legitimate sandwich vendor. Of course the attacker can't recover key  $K$  and then no shop can be abused since the logs would not be readable. But what if the attacker obtains key  $P_{enc}$  by one way or another? A loyalty card would be totally clonable and no vendor would figure out.

The system is also vulnerable to a **replay-attack** due to the authentication mechanism. Since authentication relies on a session key, every operation requiring authentication is allowed as long as the session key is known by the prover. For a write operation, the application proves its identity by encrypting the data with the session key. That means an attacker would be able to increase its counter of sandwiches simply listening to the channel and replaying the message sent to the tag when a sandwich is bought for real.

Naturally, like any other RFID-based system, our system may be vulnerable to **DOS attacks**. The first weakness described in this section is an example. An attacker may also use more generic DOS attacks such as making electronic noise to make the service unavailable.

## 3.2 Towards a better solution

### 3.2.1 With same technology and file/data structure

A first improvement would be to put keys  $P_{enc}$ ,  $K_{enc}$ ,  $P_{shop}$  and  $K_{shop}$  in encrypted files protected by a password that the user needs to enter when launching the application. That would prevent an attacker from retrieving the keys and/or using the application if the executable and the keys are stolen. Similarly,  $K_{DESFire}$  should be encrypted in such a file so as to make it impossible to retrieve it from the source-code.

Information leakage due to the free-readable counter can be easily avoided by forcing authentication with key  $K$ , just as for the logs. In this way, it is no longer possible to learn the value of this counter by skimming the tag or eavesdropping the communication channel.

Forcing authentication to read files containing S and E would be a solution against tag impersonation. If we want to force the authentication with key  $K_{M1}$  for example, we need this key to be shared by every shop. Of course the problem is not totally solved since an attacker may obtain  $K_{M1}$ , but this at least provides a good defence in depth.

### 3.2.2 With same technology only

## 4 Conclusion