The Vault, an Architecture for Smartcards to Gain Infinite Memory

Patrick BIGET
Gemplus Research Group,
Avenue du Pic de Bertagne, B.P. 100,
13881 Gémenos Cedex France
Patrick.Biget@gemplus.com

Abstract. Smartcard chips vendors have always done their best to embed more memory inside cards. These efforts are driven to allow their customers - smartcard manufacturers - to mask more software inside cards (in ROM) but, above all, to help them to provide cards with more memory dedicated to the application (EEPROM). Even if the geometry is getting smaller and smaller, some applications do not match with the current memory limitations due to smartcard constraints making impossible for the chips to be more than just a few millimeter square. The goal of the Extended Memory Card project is to suggest an architecture in which smartcards can securely "contain" more data than their own memory allows it. The card acts as a key to access information stored outside of it.

1. Introduction: the WebCard prototype

1.1. Presentation

The basis of the *Extended Memory Card* project lies on an earlier work performed at the RD2P laboratory (Research and Development on Portable File), a research laboratory of USTL (University of Science and Technology of Lille - France). A prototype called WebCard has been developed and is described in [1].

WebCard has chosen a medical portable file as the reference application. A medical portable file smartcard is a secure container of personal medical information. Each patient is given a card in which his medical file may be stored (details, history of diseases...). The card is a part of a global medical information system in which information relative to each patient is secured.

The card used to store the personal medical information is the CQL card in which data can be directly modelled into an entity-relation scheme (CQL stands for Card Query Language and comes from SQL). The CQL card does not manage file, as all the other cards do, but tables. In each table, you can store character strings of any size (the memory is allocated dynamically at each cell writing or updating). Some access rights (or privileges) can be associated to each table to restrict its use to a category of users. In the application, depending on their profiles, doctors, nurses and medical hostesses will not be granted the same privileges. Thus they will not be able to read and/or write the same kind of information.

In this architecture, the card provides the security of personal and private information. With a 10-kilobyte memory inside the card dedicated to the application, the file must be restricted to the critical information. How may we store X-ray photographs that are part of the medical file and which require several hundreds of bytes?

To counterbalance these card restrictions, the idea is to store a part of the medical file outside the card. The information, too large to reside on the card, may be stored on servers that are part of the medical information system. In that case, instead of containing the whole information, the card just keeps a reference to it (a non-ambiguous way to address it).

1.2. The integration platform: the World Wide Web

The WWW platform offers a suitable support to this application since it provides a uniform way of location all around the Web as well as easy interfacing facilities via Web browsers applications managing HTML documents.

The WebCard has chosen to store URLs in its table to reference external data. As URLs are character strings, they can be easily stored into table cells. To browse and update information inside the card, a set of HTML forms has been designed to consult card content after having presented the correct login and password to identify the user's profile.

1.3. A CGI-based architecture

An architecture based on CGI scripts has been designed for the application that is located on a Web server of the medical information system to gain access to the card inserted in a smartcard reader on the client station. CGI is the *Common Gateway Interface* which allows to create Web pages on the fly based on information from buttons, checkboxes, text input and so on.

A script, called CGI-W2C (CGI-Web To Card), is implemented on the server station. It is a gateway between the server and a program running on the client station, called *CQL Socket Driver*, which is the piece of code that makes the real requests to the card. Fig. 1 presents this architecture by underlying the multiple data exchanges between all the entities taking part in any transactions involving the card.

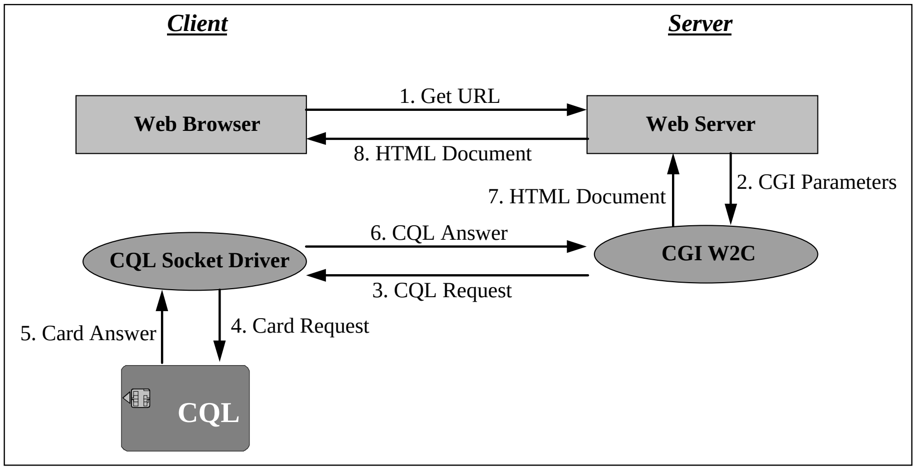

Fig. 1. The WebCard Architecture

2. Security Architecture

2.1. Security Requirements

The architecture described in the previous chapter shows two terrible weaknesses in terms of security.

First, the information stored on the Web servers outside the card are totally unsecured (not encrypted) although they may be private and confidential. Even if a part of the information retained by the card is outside it, the idea here is to provide the same level of security for those deported data.

The first solution that may be considered regarding this problem consists in securing servers physically and logically. But to reach your Web browser, an HTML page travels all around the net following an unpredictable path (possibly through all the WWW servers of the world!). At each node of the Web, this page may be easily copied for harmful usages. That is why data should never be transmitted plainly. This makes obvious that the information must at least travel over the net in an encrypted form. There are two solutions that meet these requirements.

The first one consists in building a security protocol performing a mutual authentication between the client and the server in order to establish a session key, when an external information must be accessed. This implies to set up a complete certification chain where certificates (like X.509 certificates) are distributed by an authority to all the entities involved in the transaction (servers and clients). According to the potential number of servers and mainly of clients, this solution is very difficult to implement.

Another method making the system much easier to manage is to encrypt the documents at their creation when they are stored in the remote severs. This is also a more economical way to solve the security issues of the architecture since the application may use a large number of external servers (in this case, securing all the servers may require lots of efforts). Here the key used to cipher documents is a private key owned by the card and documents are directly encrypted by the card. This method allows to avoid a mutual authentication. The server does not need to authenticate the client since the delivered document is encrypted and therefore usable only by the right person. The client does not need to authenticate the server since nobody could be able to build a valid document without the right key.

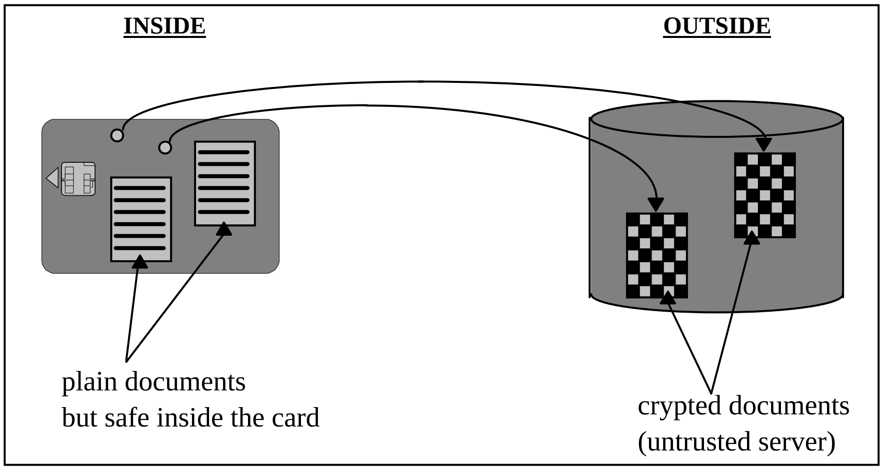

Fig. 2. The Vault principle

The issue for this solution is now to find a way to encrypt and decrypt the documents using some secrets stored inside the card pursuant to two constraints:

- the documents may be larger than card memory capacity so they cannot be totally encrypted/decrypted inside the card,
- and the secrets must never go out of the card.

The second security weakness of the WebCard architecture is that even data stored inside the card are made unsecure because of the architecture since all the answers from the card travel plainly on the network (see Fig. 1). The new architecture must be designed so that transactions with the card are locally performed to the client station.

2.2. The Remotely Keyed Encryption Protocol

[2] proposes a solution that authorizes a secure, but bandwidth-limited, cryptographic smartcard to function as a high-bandwidth secret-key encryption and decryption engine for an unsecure, but fast, host processor.

The host wants to encrypt and decrypt large blocks under a secret key stored in the card without knowing it. The card knows the key *K* but is computationally and bandwidth limited and cannot process entire blocks within the time required by the host. The Remotely Keyed Encryption Protocol (RKEP) allows the host to perform a single, fixed-size low-bandwidth interaction with the card to obtain enough information in order to encrypt or decrypt a given arbitrary length block.

RKEP requires from the smartcard and the host to share a block cipher algorithm that operates on *b*-bit cipherblocks keyed with a *k*-bit key. We have chosen to use the DES algorithm, then we can assume that we will use 64-bit cipherblocks and 56-bit key.

The host operates on large blocks of plaintext (*P*) and ciphertext (*C*), each consisting of a serie of *n* individual 64-bit cipherblocks, denoted $P_1 ... P_n$ and $C_1 ... C_n$, respectively. $I_1 ... I_n$ denote temporary "intermediate" cipherblocks internally used on the host by the protocol.

We denote encryption of plaintext block *p* under key *K* as $E_K(p)$ and decryption of ciphertext *c* under key *K* as $D_K(c)$. $\oplus$ denotes bitwise exclusive-OR. It is assumed that the host can efficiently calculate a public function *H(t)* that returns a 64-bit cryptographic (one-way and collision-free) hash of arbitrary length bitstring *t*.

The encryption of *n*-cipherblock plaintext block *P* producing ciphertext *C* is shown in Fig. 3. Decryption of *C* is shown in Fig. 4.

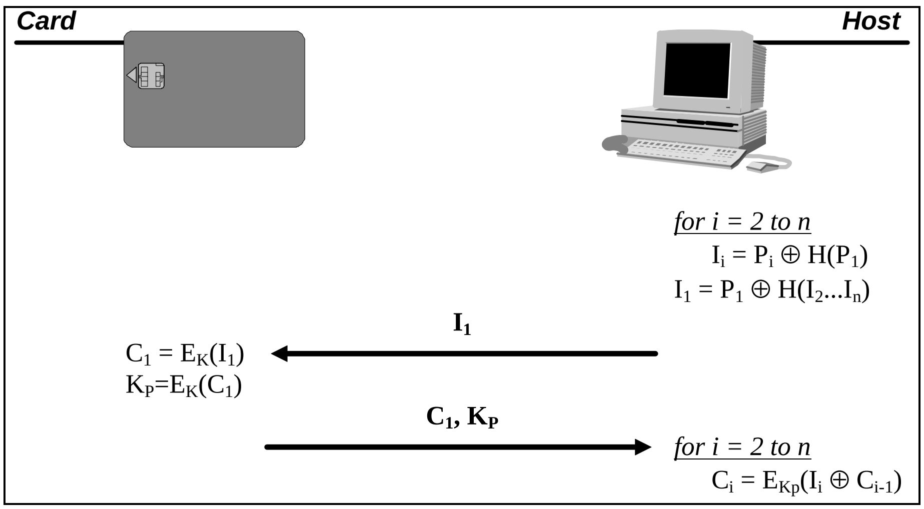

Fig. 3. RKEP encryption of P to obtain C

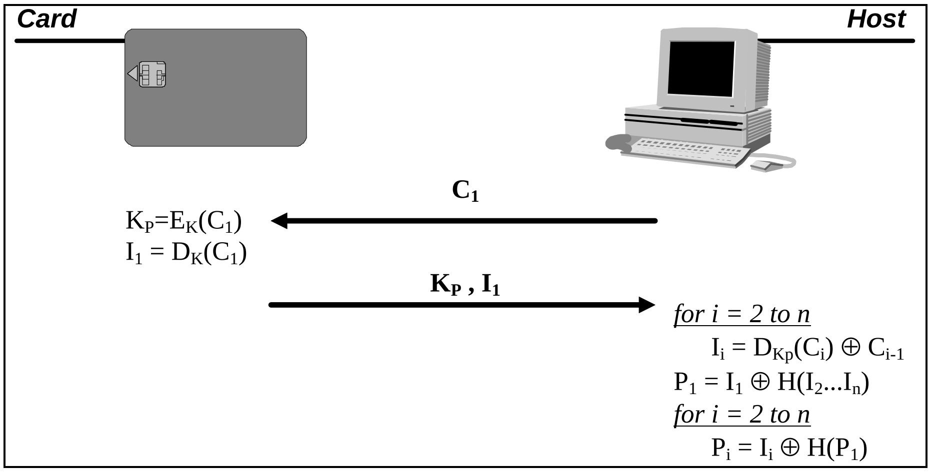

Fig. 4. RKEP decryption of C to obtain P

3. A New Application Architecture

3.1. Inadequacy of the WebCard Architecture

The choices we made by building the security architecture lead us to break the application architecture designed in the WebCard project. Once the external information are encrypted, it becomes impossible to build dynamically the HTML pages on the server. Before being displayed on the client browser, data need now to be decrypted by the smartcard which is attached the client station. Thus the manipulated data must be handled locally by the client.

We have actually to modify the architecture in the way the main process takes place on the client side. Then the role of the server moves to the simple data storage.

This solves another issue of the old architecture : the availability of the resources on the Web. Effectively, in the WebCard way of running, all the process is performed by the server. Even the accesses to smartcards that are performed on the client station by the *CQL Socket Driver* (see Fig. 1) are controlled directly by the server. This application « strategy » is to be reviewed for a more optimized version with respect to the Web transfer rates. So an architecture in which a major part of the work is performed on the client station is also be much preferable for efficiency reasons.

The last limitation of the WebCard prototype is the most pragmatic one. The prototype has been implemented on a Sun/Solaris platform in native code. It does not run on another platform. A more portable approach may be considered for the *Extended Memory Card* demonstration.

3.2. An Improved Architecture Using Java Technologies

Using a Java applet on the client station to access the card makes the architecture much more valuable. First, Java applets are dynamically loaded on the browser. Thus, there is no need to install a dedicated software on the possibly huge pool of client stations. Then, the browser may directly interact with the applet, consequently with the card, to retrieve the required information. In this way, the card accesses are stand-alone. Finally, before being executed, an applet is carefully checked by the sandbox of the Java Virtual Machine.

The sandbox is the module that verifies the adequacy of the applet with the security policy defined in the environment. For example, an applet would not satisfy the security rules of any environment if it tries to access the local disks or use any peripheral device. This last point solves the security issues raised in the previous chapter but introduces an important constraint: it forbids a "pure Java" applet to use a smartcard reader as it can only be a peripheral device. That is why the communication layers with the reader must be written in native code. This native code can be called from an applet using Java Native Interface, a standard interface to call external native programs. But this implies that the client has to trust this native library.

On the server side, the recently issued Java Servlet API proposes a very interesting alternative to CGI scripts. A servlet is the opposite end of an applet. It can almost be thought of as a server-side applet. Servlets run inside the Web server in the same way as applets run inside the Web browser to generate dynamic HTML pages. Basically, CGI scripts and servlets offer exactly the same services but servlets provide better features in term of security, efficiency and usability.

Here, our concerns are mostly focused on efficiency. In addition to the classical advantages brought by Java (platform independence, code reusability, programming language efficiency), the main improvement with servlets is performance. Servlets only need to be loaded once, while CGI programs need to be loaded for each request. The servlet `init()` method allows programmers to perform resource intensive actions (such as database connections) at startup and reuse them across servlet invocations. To know more about servlet, refer to [4] and [5].

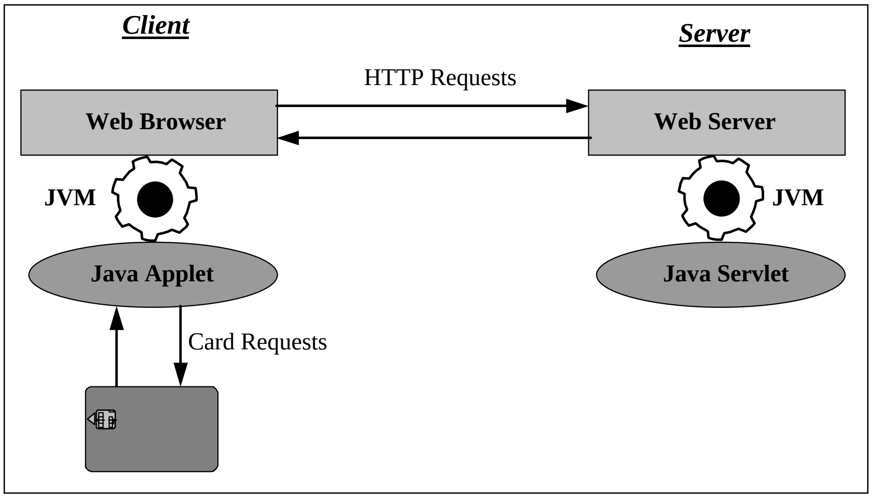

Fig. 5. A Full-Java Architecture

Fig. 5 illustrates the new architecture we have build for the project. *CGI W2C* is replaced by a servlet and the *CQL Socket Driver* is replaced by an applet. The applet on the browser allows to perform the main part of the application using local resources without any network request (we especially think here of card accesses). The servlet just provides a data storage service. This service is invoked by the applet using HTTP-POST requests that deliver to the server the data to be stored as well as a filename on the server.

The URL stored into the smart card as a reference to the information is broken up into two parts : a machine name identifying the server where to find the running servlet and a filename to be used as a local name on the server.

4. Industrial Challenges and Perspectives

The final goal of this project is to tackle the never-ending problem of the lack of memory inherent to smartcards. As said in the introduction, it's unreasonnable to think that we will have one day a smart card embedding « enough » memory since the requirements grow as fast as the technology improvements are able to provide. Storing data outside the card but secured by the card has appeared to us a good solution to solve the problem.

Our short-term objective in order to evaluate the potential market for such kind of solution is to build rapidly a demonstrator illustrating functionally the principles described in this paper. Then, the next step will be to use this demonstrator to promote the concept of an extended memory card and find some industrial partners to initiate a real application.

The application demonstrator we have chosen is an health-care portable file application since we think it illustrates perfectly the issues and it is easily understandable. Such a portable file requires indeed a strong security (since it refers to stricly private information) as well as large storage capacity (for a real and complete medical file).

Another objective of the project is to demonstrate the power of the new JavaCards and their efficiency to develop rapidly software inside smartcards. Effectively, even if the role of the card is here mainly limited to a data repository, the card takes an active part in the encryption/decryption process. That is why the card software has to be customized to follow our requirements. Moreover, using a JavaCard, gives a general coherence to the architecture from Web servers to smartcards passing through Web clients.

This project, based on a full-java architecture, has also been used to evaluate the valuable interest of the OpenCard Framework [6]. The OpenCard Framework is an architecture developped by IBM in collaboration with the major smart card manufacturers and IT industry actors which aims at providing a framework for smart card solutions interoperability. As the OpenCard Framework architecture is Java-based, using it in our context has been very natural and efficient.

5. Conclusion

The project brings an opportunity to investigate new businesses in the development of smart card solutions and promotes the use of smart cards to secure private data storage. Its development has been an occasion to gather a pool of recent standardization initiatives to illustrate the high interoperability introduced by these emerging standards and the simplicity induced in the development of applications using smartcards.

This will undoubtedly reinforce potential markets for smart cards by breaking once for all the false image generally attached to smart cards : small memory capacity and difficult integration into applications. These constraints will no more be a barrier for developping new innovative highly secured applications.

References

[1] Vandewalle, J.-J.: Projet OSMOSE: modélisation et implémentation pour l'interopérabilité de services carte à microprocesseur par l'approche orientée objet. PhD Thesis, Université des Sciences et Technologies de Lille (1997) 50-56

[2] Blaze, M.: High-Bandwith Encryption with Low-Bandwith Smartcards. `ftp://ftp.research.att.com/dist/mab/card_cipher.ps`

[3] Hui, E.: CGI, Plug-ins, & etc... Course of the Department of Computer Science at the University of Calgary. `http://www.cpsc.ucalgary.ca/~kremer/courses/547/CGIandPlugins.`

[4] Dougherty, D.: Understanding Java Servlets. `http://webreview.com/97/10/10/feature/colton.html`

[5] Crawford, W.: Developing Java Servlets. `http://webreview.com/97/10/10/feature/main.html`

[6] Hermann, R., Husemann, D.: OpenCard Framework 1.0 White Paper. `http://www.opencard.org/docs/whitepaper`