Proposal for Standardization of Access to Smartcard Services from Remote Web Applications

Bud P. Bruegger, Comune di Grosseto, Italy

Antonino Iacono, Opensignature Project, Italy

Matteo Artuso, Consultant, Italy

Abstract

This essay reasons that there would be great benefit in standardizing an API through which web-applications can access services provided by client-side smartcards, in particular eID cards. It further reasons that the Url Programming Interface approach that was originally proposed by Sun Microsystems Laboratories is the best suited architecture for such an endeavor. The essay is written in the hope to help create a consensus among stakeholders to start a corresponding standardization effort.

Introduction

EID and other kinds of smartcards are increasingly used in web applications. These run on remote HTTP servers but access services provided by client-side smartcards. Examples for such services are digital signing of documents or the extraction of an elementary file containing personal data.

Currently, only the service of client authentication is standardized via the SSL/TLS handshake that is supported by all major browsers. For all other functionality, different web applications make different choices from several available approaches and provide their own implementation of the solution.

This essay reasons that this diversity of solutions is excessive and that a standardization on a single multi-platform and multi-browser interface would provide great benefits in respect of manageability, integration, and security. This is of particular relevance when considering cross-boundary e-government services.

For this purpose, the essay proposes to standardize an Url Programming Interface, i.e., an API accessible through HTTP. This approach was originally proposed by Sun Microsystems Laboratories (http://www.javaworld.com/javaworld/jw-11-1999/jw-11-javadev.html, http://www.javaworld.com/columns/jw-developer-index.shtml, http://research.sun.com/projects/dashboard.php?id=45) and has since been adopted by others (e.g., http://www.microexpert.com/smartcardurl.html). It provides a solid multi-platform and multi-browser solution on top of which a standardization of functionality and interface is possible.

Current Situation

This section briefly reviews the most currently used solutions for accessing smartcard services from web applications.

Java Applets

Probably the most popular way to accessing smartcard services from remote web applications are java applets. Applets are pieces of software written in Java and embedded in web pages. After download, they are executed in the browser's Java Virtual Machine (JVM). In order to be

successful, openings have to be made in the JVM's "sandbox" security policy in order to access the required local resources.

To avoid the dangers of executing untrusted code that can access local resources, only signed applets from trusted sources are allowed to run. It is the user's decision who can be trusted. As Bruce Schneier states (http://www.schneier.com/essay-031.pdf, Reliance on Intelligent Users), average users fail to understand the security implications of such dialogs, consider them annoying, and click wherever necessary to just activate the desired functionality. A specific discussion on how code signing affects security can be found here http://www.schneier.com/crypto-gram-0101.html#10.

There are two possible types of applets:

- Single-platform applets that contain all the necessary code including libraries (DLLs, Shared Objects) that are platform dependent. These applets can be rather large; but caching can ease this problem for repeated use. Single-platform applets require the server to successfully detect the platform and to provide the libraries for all possible (or supported) client platforms. This can be a challenging requirement.
- Multi-platform applets that consists solely of the java code and assume that the platform-dependent libraries have already been installed on the client.

In both cases, the client machine has to be prepared for successful use. Both types require the configuration of the security policy of the JVM, only the latter requires the installation of libraries.

Note that, obviously, applets only work on browsers that have an integrated JVM and that are configured to activate it.

Plug-ins

An alternative to applets are browser-plugins. Once installed, they permit Java Applets or JavaScript that is embedded in html pages to access smartcard services. To some respect, they are very similar to the multi-platform Applet approach.

One of the specific problems is the lack of standardization of the plug-in API. In particularly, browsers use either the Microsoft or the Netscape API, and some browsers (for example on cell-phones or PDAs) lack support for plugins alltogether.

But even assuming a single plugin API, the lack of standardization causes every application to come with its own, different, and proprietary plugin.

ActiveX

... only Windows ... similar to Applets

ActiveX security report: http://www.cert.org/reports/activeX_report.pdf

Problems inherent in Diversity

A problem of all the above described approaches is their lack of standardization. For satisfying a single functionality, every web-application uses a different software component (applet/plugin/library) with a different API.

This diversity renders system administration on the client complex. For organizations who have

several hundred of desktop machines, installing and configuring a single client component (library, plugin, etc.) is already challenging, to install and configure several and manage version upgrades is easily becoming a management nightmare.

Further, client configuration, for example the JVMs security policy, is problematic for two reasons. First this configuration is typically managed by end users who lack understanding of the underlying concepts. Even if the a tech support person correctly implemented the organization's policy, users can willingly or inadvertently change this later. The second problem is the interaction of several software components from multiple vendors where each poses different requirements on a secure but functional configuration. Since no one is responsible for the overall configuration, in the real world, client machines are likely to end up with configurations that ensures functionality by avoiding to enforce a tight security policy.

From a security point of view, the strongest guarantees come from a security inspection (or at least the possibility of one) before installing the code. This is obviously only possible for libraries and other code that gets installed on the client machine, but not for code that is embedded in web pages.

It would be desirable that the installed libraries limit the possible smartcard functionality that is exposed. For example, it is saver to offer the function to sign a single document much rather than signing an arbitrary digest string whose origin is a remote server.

The currently used approaches render this kind of security management rather difficult. Apart from the untrusted code in many approaches, the multitude of solutions from various applications multiply the effort necessary for security auditing and significantly increase the probability of oversights.

The probably most serious problem, however, is the lack of a single and consistent decision on how to present smartcard functionality to users. Legally signing a document may take many different forms and various levels of making users aware of the importance of their approval to sign.

The authors believe that, in a large user base, it is already difficult to foster basic literacy in digital cryptography; we believe that this becomes a potentially lost battle if in absence of a single consistent interface that users can learn once and understand. The big danger is a significant occurrence of unintentional, but legally binding signatures when average users “click anywhere necessary” to get back to some controlled situation. This may well be the most serious threat to large-scale deployments of digital signature technology.

UPI Architecture

The Architecture of the URL Programming Interface attempts to maximize portability in terms of being:

- multi-platform (Windows, Linux, PalmOS, etc.)
- multi-browser (including “embedded browsers” of PDAs)
- robust to browser configuration (java/javascript installed and enabled, JVM policy, etc.)

It achieves this by relying on a minimal infrastructure that is ubiquitous in todays computing world:

- HTTP
- HTML (in particular Forms)

The contrast to the above described solutions that require Java (installed, correct version, enabled, smartcard-specific library installed), plugins (browser dependent, not supported by "embedded browsers"), or ActiveX (Windows only) is obvious. The unprecedented wide range of applicability of the UPI approach makes it an ideal bases for a standardization effort that aims for interoperability of multiple applications on a wide range of platforms.

The base mechanism of how remote web applications access local smartcard services through html forms that submit to a dedicated http server that exposes the smartcard services. This is illustrated in figure 1.

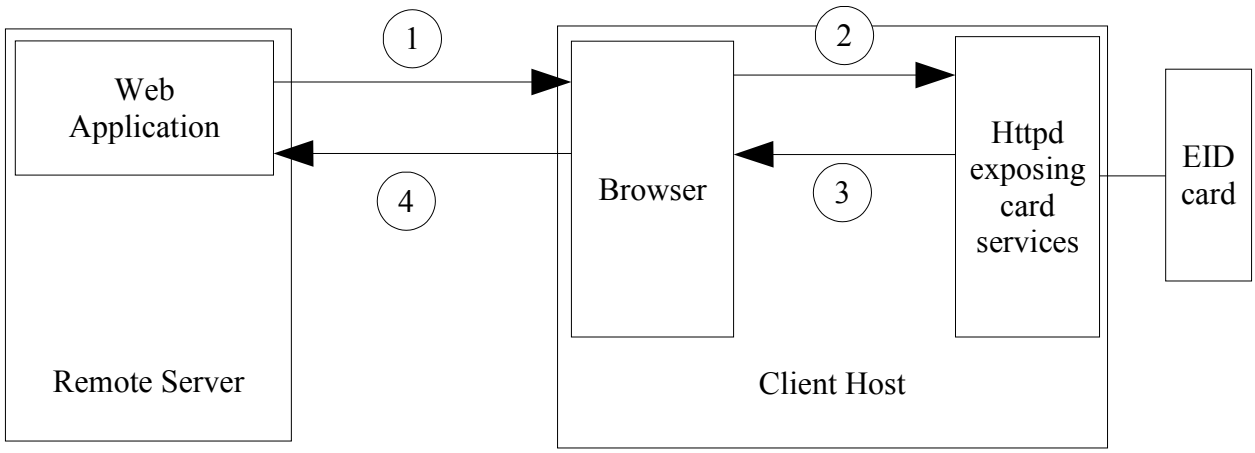

Figure 1: Overall communications flow.

The remote web application initiates the communication by sending an html page (1) to the browser that contains a form. The form is configured to submit to *localhost* to a standardized port and specifies the requested service in the URL. A standardized hidden form field contains the URL on which the web application expects to receive a response.

The browser renders the page that explains the requested smartcard operation to the user. The user explicitly has to approve processing[1] of the requested service by clicking the form's submit button. This sends the filled in form to the smartcard's http server (2). The request optionally contains data as POST parameters. The posted form data contains the URL provided by the remote web application, but can also contain additional components such as the cards PIN or a file to be uploaded for signing.

The local http daemon receives the request and, for security reasons, verifies that it originates from *localhost*. Within the limits of its security policy, it processes the requested card service, for example, it signs a document. It then formats the result as a http response that consists of a html page with a form (3). The form is configured to submit to the URL specified by the remote web application.

The browser renders the response received from the local server. Typically, and under the central control of the local http server, the page visualizes the result of the service and to whom it will be submitted. Since the local http server is the only possible source[2], it is easy to guarantee a consistent user interface that guarantees that users are aware of all critical actions and explicitly approve their execution. The possibility of security auditing gives guarantees that these security features cannot be circumvented.

1 It is possible to avoid this approval by using java or javascript to create an automatically submitting form. But the same is not true for the response (3), thus guaranteeing security.

2 For all web applications, not necessarily for local applications

The user then gives explicit approval to send the information to the remote web application (4). The format is a HTTP POST whose fields (some or all were hidden in the form) are specified by the standard.

Figure 2 illustrates the architecture of the local http server that exposes the smartcard services. The top component is a minimal and lightweight http daemon that listens on a standardized port. An access control layer restricts access to requests from *localhost*. It would be possible to also restrict on the remote web applications that can access to various services.

The actual services that are exposed are managed by a specific layer. It represents an important point of control to enforce policy and restrict the possible uses of the smartcard. For example, it may only allow to sign documents of a limited number of formats that must be submitted in whole as part of the request; but disallow the signature of arbitrary digest strings from unverified origins. A security audit before installation can verify that the policy is correctly implemented and that it is impossible for remote applications to use the card for undesired purposes.

The high level services may rely on a crypto library and on the low level services of the smartcard to satisfy requests. For example, the digest of a submitted document may be computed on the local CPU and but signed in the smartcard.

Different implementations can make different decisions on the kind of card access API to use. The simplest implementation would be to directly use a PKCS#11 provider for a single eID card; a more complex implementation would be some dispatcher approach that detects the card (e.g., based on its ATR) to then select the adequate PKCS#11 provider from those available on the system. OpenSC (http://www.opensc.org) may be an alternative that incorporates the support for a significant number of eID and other cards without the need for external drivers such as PKCS#11 providers. This latter approach seems to greatly facilitate support for multiple eID cards on multiple platforms[3].

The actual access to the card relies on of operating system services and a driver for the card reader that is used.

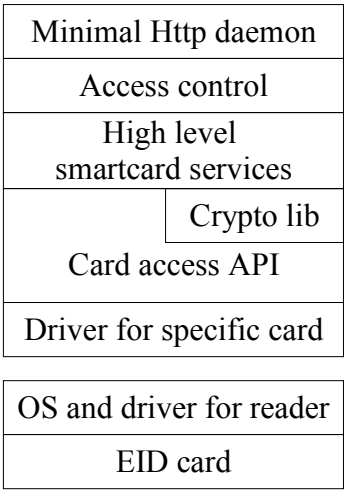

Figure 2: Architecture of the local http server

A prototype implementation of the UPI approach that demonstrates the concept is already available from the project OpenSignature (http://sourceforge.net/project/showfiles.php?group_id=67103&package_id=128958).

3 Note that in Italy, public administrations typically encounter one of two National eID cards (CIE and CNS) or one of the many cards for legal digital signature that are issued by certified private CAs.

Proposed Standardization

A standardization of the API to remotely access smartcard services could alleviate the above described problems. It can provide a *single point of control* for the following aspects:

- API used by remote web applications
- security auditing
- security policy
- look and feel of important functionality
- integration of diverse eID cards

Considering that the UPI approach works on literally any platform and any browser, the authors believe it is a perfect choice as a base for the standardization.

We expect that a relatively small standard can cover most common uses of smartcards. Of particular interest will be the following functionality, maybe with a few variations:

- digital signature of documents or web forms
- extraction of files from the smartcard, for example personal data from eID cards

The standard needs to address a base mechanism that is mandatory and a prerequisite for all services. It involves issues such as:

- the port on which the local http server is listening
- the name of the hidden field which specifies the URL where to submit the results

Further, the standard has to enumerate mandatory and optional services. Each service is specified by the following elements:

- the base URL that identifies the service
- possible additional options that are passed via the URL (for example, as GET parameters)
- the input and output parameters that are expected in the POST data and the HTML Form of the response, respectively.

The standard could optionally make recommendations on the presentation of various functionality to the user.

Conclusions

A relatively small standardization effort promises to alleviate a significant number of operational problems of today's excessively diverse use of smartcard services in web applications. It promises a single point of control for various aspects including security auditing, enforcement of security policy, and most importantly, a consistent presentation of functionality to users. The paper has reasoned the significant benefits of using an Url Programming Interface as bases for such a

standard.