Plugging a Scalable Authentication Framework into Shibboleth

By N. Zhang, L. Yao, J. Chin, Q. Shi, A. Nenadic, A. McNab, A. Rector and C. Goble

School of Computer Science,
University of Manchester,
Oxford Road, Manchester,
M13 9PL, UK
{nzhang, yao1, jchin, nenadic, rector, cgoble}@cs.man.ac.uk

A. Nenadic, A. McNab

School of Computing and Math Sciences,
Liverpool John Moores University,
Byrom Building, Liverpool,
L3 3AF, UK
q.shi@livjm.ac.uk

Abstract

Shibboleth is an internet2 project developing security architectures and protocol messages supporting inter-institutional sharing of resources that require access control. Users identify and authenticate themselves locally to their respective institutions, and the institutions assert the users' identities through the use of attribute certificates signed by the institutions and SAML. The service providers then make access control decisions based upon users' privileges contained in the attribute certificates. In this VO (virtual organisation) environment that the Shibboleth solution is designed for, services are provided by dissimilar organisations which are from different security domains and have dissimilar security requirements. Therefore there is a need for a dynamic authentication mechanism that supports a whole range of authentication tokens and the authentication strengths derived from these tokens should be fed into the access decision making. This paper reports our ongoing efforts in designing and implementing such an authentication framework to facilitate multi-level and multi-factor authentication and authentication strength linked fine-grained access control in Shibboleth. The proof-of-concept prototype using a Java smart card is reported.

Index Terms – Grid authentication, security architecture, smart token.

1. Introduction

Shibboleth [Shib] is an internet2 project developing security architecture and protocol messages supporting inter-institutional sharing of resources that require access control. The Shibboleth security architecture uses the concept of federated administration, in which institutions are mutually trustworthy, users are identified in their home institutions (i.e. origin sites), and the home institutions assert the users' identities through the use of attribute certificates and SAML (Security Assertion Markup Language) [SAML]. The attribute certificates that are signed and dispatched by the origin sites are then used by service providers (i.e. target sites) to make access control decisions. The Shibboleth framework itself does not address the issues as how the users should be identified and whether access control should be linked to the authentication strength of the token used in the identification process. Rather it merely provides a three-party framework (as shown in Figure 1) and the related protocols to support inter-institutional virtual data/resource sharing.

User electronic authentication (e-authentication), a process of positively identifying and verifying a user’s identity through the use of an authentication factor over telecommunication networks, is a prerequisite to allowing access to inter-institutional services and resources. Authentication factors are typically grouped into three categories [O’Gor]: knowledge-based such as passwords, object-based such as memory or smart tokens/cards, and ID-based such as biometrics. Different
authentication factors have different strengths and weaknesses and provide different levels of assurance in identifying a user. For example, a password is easier to forge than a smart token equipped with a cryptographic key, but the latter is more susceptible to theft or loss. Though biometrics is more difficult to forge, it alone can not be used for e-authentication. In addition, to achieve a higher level assurance, two or more authentication factors can be combined and used to identify a user. For example, a smart token locked with a fingerprint, which is a two-factor authenticator, is a better choice than using the token alone.

In an inter-institutional distributed system (i.e. VO or Grid) environment, such as the one addressed by the Shibboleth solution, users are typically from dissimilar organisations and administrative domains, and may be allocated with different roles. Similarly, services and resources are provided by dissimilar institutions usually with dissimilar authorisation policies and access requirements. There is, therefore, a need for a fine-grained access control framework to satisfy the complex VO resource management requirement, and one way to do this is to link a user’s access privilege to the authentication strength of the authenticator used by the user. In other words, for a user to have a higher access privilege, an authenticator(s) with a higher level of assurance (LoA) should be required, vice versa.

In a Health Grid, where electronic patient records (EPRs) and/or electronic health records (HERs) are shared among GPs, clinicians, and clinical and biomedical researchers across different institutions and organisations, the need for an authentication strength linked fine-grained access control framework is even more obvious. EPRs/EHRs have high levels of privacy requirements due to legal and ethical reasons. Therefore, it is usually expected that EPRs and HERs be de-personalised (i.e. sensitive information that can be used to identify the owner of a record be removed) before being released to entities outside hospital premises or before the researchers are allowed to access them [Nelk]. Password-based authentication methods may be sufficient to identify the researchers for them to access these de-personalised records. However, the suppliers of the records, e.g. GPs and hospitals, may need to regularly update the records or feed new data into the de-personalised data repository for continued and quality care, and should use a strong form of authenticators due to accountability and data integrity concerns.

The multi-level and multi-factor authentication framework that we are developing can authenticate users with a whole range of authenticators, derive authentication strengths based upon the authenticators used, and feed the authentication strengths to an authorisation decision engine that controls data, service or resource accesses. The framework is called FAME (Flexible Authentication Middleware Extension), and is being developed in the context of Shibboleth. As part of this work, we have designed and implemented a Java Card enabled Web Authentication Service (JC-WAS). The rest of this paper reports our work on the design and implementation of FAME and JC-WAS.

In detail, Section 2 introduces various authenticators and their levels of assurance (LoA). Section 3 describes our FAME framework and its integration with the Shibboleth architecture. Section 4 introduces smart cards and their application in e-authentication. Section 5 is devoted to JC-WAS. Sections 6 outlines the lessons learnt during the implementation of JC-WAS, and finally, Section 6 gives our conclusion.

2. Authenticators and their Levels of Assurance (LoA)

As discussed above, different authenticators provide different LoA in identifying a user. NIST (US National Institute of Standard and Technology) [NIST] has defined a draft standard on authentication tokens versus LoA. As shown in Table 1, Level 1 authenticators have the lowest LoA, and Level 4 have the highest. Using a Level 1 authenticator, the likelihood of a successful identity forgery or impersonation is higher, and therefore the security risks are higher. On the contrary, to compromise a Level 4 authenticator, say a smart card hard token locked with a PIN number, the perpetrator would have to firstly get hold of the card; and secondly guess the PIN number. Hard tokens are therefore more difficult to compromise and have the highest LoA in e-authentication. Our FAME authentication framework is aimed at integrating all the authenticators shown in Table 1.

<table>
<thead>
<tr>
<th>Table 1: Authentication tokens and their levels of assurance [NIST]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard token</td>
</tr>
<tr>
<td>Soft token</td>
</tr>
<tr>
<td>Zero knowledge password</td>
</tr>
<tr>
<td>One-time password device</td>
</tr>
<tr>
<td>Password</td>
</tr>
<tr>
<td>PIN</td>
</tr>
</tbody>
</table>
3. Plugging FAME into Shibboleth

The integration of FAME into the Shibboleth architecture [Shib] is through the Shibboleth Handle Service (i.e. Shib-HS). The Shib-HS is the origin site component responsible for identifying and authenticating a user and providing the SHAR (Shibboleth Attribute Requester) with a handle (i.e. a user reference) that can be used to later retrieve the user attributes from the origin site Attribute Authority (AA). The target side then makes the access decision based upon the attributes fetched. The handle is embedded within an XML instance document, signed by the HS, and packaged in a SAML response to the target SHAR. Our FAME Login Server (F-LS) protects the Shib-HS making sure that a user’s handle is returned only after the user is authenticated. Authenticators are classified into groups, each of which has a specified LoA. Upon authentication, the F-LS will derive the LoA based upon the authenticator(s) used and forward it together with the Handle to the target site in the signed SAML message. The integration of our FAME and the Shibboleth solutions has been illustrated in Figure 2, in which the FAME architectural components are indicated with shaded blocks.

![Figure 2. Integrate FAME into Shibboleth](image)

From the Figure, it can be seen that our FAME framework consists of the following components:

- **The FAME Login Server (F-LS):** This is the authentication login server used to protect the Shibboleth HS. Its main roles are (1) to support integration of different authentication services, (2) to derive LoA from an authentication token(s)/method(s) used during an authentication instance, and (3) feed LoA/authentication method to the SHIRE.

- **A Host Authentication Module (HAM):** HAM is the component running on a user host (e.g. within the user’s browser) to which a smart token device is attached. It mediates the user’s interaction with the smart token and the server authentication service.

- **Token Interface API (TI-API):** It plays the role of Java software adapter providing a unified API for accessing smart tokens from different manufacturers and vendors.

- **The Authentication Services Interface API (ASI-API):** This defines the API for communication between different external authentication services and HAM.

- **Authentication Services (AuthService) with necessary auxiliary services such as time and directory services:** an AuthService is a Web based service that authenticates a user using some defined authenticators. FAME will support authenticators including IP address, passwords, certificate-based soft and hard tokens, one-time passwords, Kerberos, as well as traditional authentication systems such as RADIUS, LDAP, NIS, etc, all of which should have a Web front-end. An example of the AuthServices is our Java Card based Web Authentication Service (JC-WAS) to be discussed in Section 5 of this paper.
Figure 3. FAME functional flows

Figure 3 shows the functional flows of the FAME authentication process. In the remaining part of this paper, we focus on a proof-of-concept prototype to highlight the design and construction of one of the authentication services supported by FAME, the JC-WAS. JC-WAS consists of three software/functional components: Java Card Applet, the HAM (host authentication module), and the Java Card authentication service, the JC-WAS servlet.

4. Smart Cards

A smart card is a plastic card embedded with a microprocessor and a memory chip, or just a memory chip with non-programmable logic [Hans]. The former is an IC (Integrated Circuit) microprocessor card and the latter is an IC memory card. The IC memory cards, as the name indicates, are just memory cards, and can typically store up to 1-4 KBytes of data. They rely on card readers (also called card-accepting terminals) for data processing. Therefore, they can only be used for applications with prior defined functionality and terminals. The IC microprocessor cards (also known as Chip Cards or multi-application smart cards), on the other hand, can perform some ‘intelligent’ work, in addition to the ability of storing data. They have on-card operating systems. Some smart cards, such as Java cards, have effective access control to the data stored on the cards; different applications within different Execution Contexts (i.e. different packages) on the same card can not access each other’s data [SUN_1].

Java smart cards (hereafter will be referred to as Java cards) are portable and more resilient to tampering, so they are ideal candidates for storing users’ digital IDs or credentials. They can be used to assist users to perform cryptographic operations and to allow them to access data or services on the Web securely anywhere and at anytime through different devices, a laptop PC or a smart phone. In addition, as one Java card can support multiple applications, it is more secure and convenient to use one card to implement many different Internet-based service access and e-Commerce transactions, than using many different cards each with a specific purpose. At the moment, we use different cards for different services, e.g. we have bank and credit cards, social security cards, telephone cards, store cards, etc. For this reason, we have chosen to use Java cards in our work presented in this paper. Our design has adhered to the Java Card Platform Specification [SUN_1], initially defined by SUN Microsystems, and later improved and extended by the Java Card Forum1. The Platform consists of a customized subset of the Java programming language and a Java run-time environment dedicated to smart cards. Due to the limitations of computational and memory capabilities in smart cards, Java Card Platform only supports a number of carefully selected subsets from Java standard language. For example, it only supports byte and short as its primary types and does not support dynamic class loading and garbage collection [SUN_1].

5. The Java Card based Web Authentication Service (JC-WAS)

5.1 Design Consideration and Pre-requisites

Following assumptions are made in the design of JC–WAS.

- Users have subscribed to some Credential Service Provider (CSP), e.g. Verisign. Their identities have been verified by a Registration Authority (RA) that may be the CSP, or a separate organisation but trusted by the CSP. Upon the successful identity verification, the users are each issued with a Java card.
- The users only trust a manually and locally installed browser plug-in or a signed Java applet from a trusted server to talk to his/her Java card.

5.2. JC-WAS Architecture

This service is developed based upon the eBusiness Application Architecture designed by IBM [Hama]. It uses certificate-based authentication technology and challenge-response authentication protocol. A Java card is used to store its owner’s authentication credential, namely the public-key certificate and the private (signature) key. The key pair is generated inside the Java card, the public key is exported to a CA for key certification, and the private key never comes out of the card. Once the certificate is generated and signed, it is loaded back into the card. The card, in addition to being a data storage device, is capable of executing Java applets to perform cryptographic operations such as public key encryption/decryption, digital signature generation/verification, and hash operations.

As shown in Figure 4, the architecture has the following main architectural components:

- Java Card Applets executed in a Java Card;
- A Web Browser Plug-in, the HAM, acting as a gateway between the User and the Card and between the Card and the Java Card Authentication Servlet;
- A JC-WAS Servlet that implements the server side authentication logic using challenge-response protocol, SSL (Secure Socket Layer), or WS-Security.

![Figure 4. The JC-WAS Architecture](image)

The communication between the browser and the server can be done through the use of HTML (or SOAP) over HTTP. The Java card authentication solution is in fact a two-factor authentication method. The proof of control of the private key stored in the Java card demonstrates that the sender is actually in possession of the card. In addition, as the card is locked with a PIN number, the card user must know the right PIN to activate the card. To impersonate the rightful owner of the card, a perpetrator would have to steal the card in addition to working out the PIN number.

5.3. The Development Environment

The following gives the development environment for the JC-WAS prototype. The operating system used is Windows XP Professional SP1. The software Development Kits (SDK) are Java 2 SDK, Standard Edition, v1.4.2 for HAM and Servlet developments; Java 2 SDK, Standard Edition, v1.3.1 and Java Card API 2.1.1 for Java Card Applet development. Bouncy Castle Java Cryptography Provider v1.24, Schlumberger Cyberflex Access SDK v4.5, and Tomcat 5.0 are also used. The
Java card used is Schlumberger Cyberflex e-gate smart card with 32K EEROM and a cryptographic co-processor. The smart card reader is Schlumberger USB smart card reader.

5.4. Java Card Applet Development

Java cards have a well-defined architecture [Orti] as shown in Figure 5. The operating system (OS) is situated in the bottom layer of the stack. From the communication point of view, Java Cards are fully compliant with the smart card standard defined in ISO/IEC 7816 [ISO7816]. The Java Card run-time environment (JCRE) consists of the smart card OS, the Java Card Virtual Machine (JCVM) and the Java Card APIs that are also referred to as the Java Card Framework. The Java Card APIs are formed by a number of packages containing classes dedicated to various purposes.

![Java Card Architecture](image)

Java Card applications, also called Java Card Applets, are located on the top-most of the architecture. More than one applets can be run on a card, and each applet is uniquely identified by an Application Identity (AID) [Orti]. All the applets must extend the Applet abstract base class that defines the methods used by the JCRE to control an applet lifecycle. The communication between an off-card application and an on-card application is done through the use of command and response Application Protocol Data Units (APDUs) [ISO7816]. Our JC-WAS prototype uses this communication method.

Two Java card applications have been developed for JC-WAS; one is the Java Card Certificate Issuance Application (or Applet) (CIA) that run on the client’s host machine, and the other is the Java Card PKI Authentication Applet (PKIA²) that runs inside the Java card.

The CIA is specifically developed for public key certification. To issue a certificate for a Java card, a pair of public and private keys should first be generated, which is done inside the card and commanded by the CIA. Once the keys are generated, the public key should be certified. There are two ways of certifying the key. One is to use a remote CA, in which case, the card is commanded by the CIA to generate a certification request by constructing a certificate in PKCS#10 format [RSA] containing the public key, signing the request using the card private key just generated, and sending the request via the Browser Java Applet to the remote CA. The CA will verify the request, and if positive, generate, sign and return a proper X.509 certificate that can then be imported back into the card. The second method is to generate and sign the certificate locally using third-party software such as the Java Cryptography Provider [SUN_2]. To certify it, the public key needs to be exported from the card. To retrieve the public key from the Java card, a method is needed to separate the public key object into its Exponent and Modulus components and to retrieve them separately from the card. Then, another method is needed to reconstruct the public key object from the Exponent and Modulus components and to retrieve them separately from the card. In detail, four methods are defined in the CIA class each for the following tasks: PIN validation (PIN_Validate()), so that the Applet can only be invoked after successful user-to-card authentication; retrieving the public key from the Java Card (Retrieve_PublicKey()); constructing a certificate request (Build_CertificateReq()); and finally, putting the signed certificate back into the Card (Import_Certificate()).

Caress should be taken when importing the certificate back into the card, as the actual size of a certificate (527 bytes in our prototype) is much longer than the APDU buffer size that is 250 bytes. Multiple APDU commands are therefore needed to import a single certificate. We use the two header fields, Instruction Parameter 1 (P1) and Instruction Parameter 2 (P2), in the APDU command to help the Java card applet to determine the offset of the certificate array.

The card PKI Authentication Applet (PKIA²), run inside the Java card, communicates with, and performs functions in collaboration with, the Browser applet and CIA host module to accomplish Java card based Web authentication, and certification related tasks as just mentioned above. For example, for the public key retrieval operation, the CIA module
sends an APDU command to the PKI² applet invoking the Retrieve_PublicKey() method, and the latter, upon receiving the command, will execute the method and send the public key to the CIA module through APDU response message.

Developing Java card applet is different from developing standard Java applications. The first consideration that a Java card applet developer needs to take into account is to design an applet with as less memory and computation power as possible, as a smart card has very limited power and memory space. A typical smart card has an 8- or 16-bit processor with speed up to 24MHz and 32Kb EEROM.

The second is to assign an Applet AID and a Package AID to both the applet and the package containing the applet class, and to define command APDUs for the functions. Most applications are named and identified by a string name. In Java Card technology, however, each applet is identified and selected by an AID. Also, each Java package is assigned an AID. This is because a package, when loaded onto a card, is linked with other packages, which have already been placed on the card via their AIDs. This naming convention is in conformance with the smart card specification as defined in ISO 7816 [ISO7816]. The first 5 bytes of Applet AID must remain identical with its package AID.

5.5. HAM (Host Authentication Module) Development

This module is executed inside Web browser on the client machine. It is responsible for user’s local authentication with the Java card using a PIN and acts as a gateway passing data from the Java card to the Web Authentication Servlet, and vice versa. HAM is implemented as a Java applet executed in the browser. The applet has less privilege than a Java application. It cannot access local resource such as reading and writing files on the client file system, or access the smart card reader. In order to access the card, the HAM Java applet needs to be digitally signed by its creator, and the digital signature provides a means of verifying the originator and the integrity of the applet. This also means that a signed applet may interact with files on the user’s computer as a Java application does. So from security standpoint, only the signed applets from trusted sources should be used.

The communication between the user’s Web browser and the Java Card, as shown in Figure 4, is done through the communication middleware that we have developed based upon OCF (Open Card Framework). The middleware uses PC/SC driver and provides the libraries for the interoperability (IOP) layer for Java (i.e. the IOP JNI) link to IOP C++ API. The IOP JNI and C++ are built on the smart card interoperability (IOP) layer [Schl]. It is vendor independent.

5.6. JC-WAS Servlet Development

The JC-WAS Servlet is developed using the Java servlet [SUN_3] and Java Server Page (JSP) technologies. The reason for using Java servlet instead of CGI is that a Java servlet is persistent while CGI is not. This means that once servlet is started, it stays in the memory and can fulfill multiple requests. In contrast, a CGI program disappears once it has fulfilled a request. The persistence feature of Java servlets makes them faster because there is no overhead in setting up and tearing down the process. JSP is an extension to the Java servlet. It has a dynamic scripting capability that works in tandem with HTML code, separating the page logic from the static elements and helping make the HTML more functional.

The JC-WAS Servlet handles the server side PKI authentication logic. If the user has been successfully authenticated, the Servlet will add a Security State to an HTTP session [SUN_3] to indicate that the user has been authenticated. The server uses standard HTTP protocol to communicate with HAM in the Web browser.

6. Lessons Learnt

At the beginning of this project, we were trying to find a smart card communication API that would allow us to use any smart card regardless of its vendor or manufacturer. The result was disappointing. There was such an effort in the standardisation of Smart/Java card application development, but the effort is no longer active. In 1997, computer manufacturers, solution providers, card manufacturers, and card reading device manufacturers organised a consortium to work on an open Java framework for smart card access. This framework was named “OpenCard Framework” [OCF]. The OCF consortium designed an OpenCard Architecture aimed at allowing Java card application developers to develop applications that can run on smart cards or other ISO 7816-compliant [ISO7816] devices on different target platforms such as Windows, network computers, UNIX workstations, etc. They provided an application programming interface (API), which allows developer to register cards, look for cards in readers, and optionally have Java agents start up when cards are inserted in the reader. However, since 2002 the OCF is no longer active. There is no official announcement but the OCF API development has stopped for nearly four years. And the smart card manufacturers do not provide CardTerminals and CardServices for developers. Although the older version of OCF is still available, developing smart card applications using OCF is no longer encouraged by manufacturers. The trend seems, at the moment, to use vendor’s middleware API.

The second observation made during the development of the JC-WAS is the complexity in the process of user certificate issuance. The Schlumberger Cyberflex SDK provides a method for importing a digital certificate into a smart card by using an online Web service provided by the Verisign [Schl]. However, this method is only available to users who use MS CryptoAPI [MS] or RSA Cryptoki [RSA]. In addition, due to Java card firewall facility, a certificate written into a card
using this method can not be access by other applets. This means that, unless you use MS CryptoAPI or RSA Cryptoki, this online certification service is not available.

Finally, it is very important to make sure that any browser extension or applet that plays the role of communicating with your Java/smart card must be from trusted sources, or must be a signed applet by a trusted entity if the applet is downloaded from the Internet.

7. Conclusion

In this paper, our preliminary work on the design and implementation of a multi-level and multi-factor authentication framework in the context of Shibboleth is presented. We have prototyped a Java card based authentication service for the proof of concept. The work has enabled us to understand the challenges involved in the use of smart cards for Web based security solutions. Most urgently, there needs to be an international standard for host-to-card communication APIs to achieve interoperability and to facilitate large-scale use of smart card technology. As our future work, we will continue to integrate other authentication systems and services into our FAME system and to fulfil our vision of linking authentication strengths to authorisation decisions and fine-grained access control for Grids. In addition, we will also be looking into the possibility of generating proxy credentials inside Java cards.

Acknowledgement

The work presented in this paper is partially sponsored by the UK Engineering and Physical Sciences Research Council (EPSRC) and the Department of Trade and Industry (DTI) (LINK, GR/R55177/01) and partially by the Joint Information Systems Committee (JISC).

References


