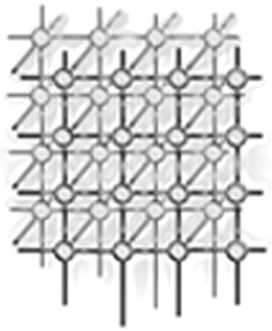
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:1333–1352
Published online 12 October 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1099



Achieving fine-grained access control in virtual organizations

N. Zhang[1,*,†], L. Yao[1], A. Nenadic[1], J. Chin[1], C. Goble[1], A. Rector[1], D. Chadwick[2], S. Otenko[2] and Q. Shi[3]

[1]*School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.*
[2]*The Computing Laboratory, University of Kent, Canterbury CT2 7NF, U.K.*
[3]*School of Computing and Math Sciences, Liverpool John Moores University, Byrom Building, Liverpool L3 3AF, U.K.*

SUMMARY

In a virtual organization environment, where services and data are provided and shared among organizations from different administrative domains and protected with dissimilar security policies and measures, there is a need for a flexible authentication framework that supports the use of various authentication methods and tokens. The authentication strengths derived from the authentication methods and tokens should be incorporated into an access-control decision-making process, so that more sensitive resources are available only to users authenticated with stronger methods. This paper reports our ongoing efforts in designing and implementing such a framework to facilitate multi-level and multi-factor adaptive authentication and authentication strength linked fine-grained access control. The proof-of-concept prototype is designed and implemented in the Shibboleth and PERMIS infrastructures, which specifies protocols to federate authentication and authorization information and provides a policy-driven, role-based, access-control decision-making capability. Copyright © 2006 John Wiley & Sons, Ltd.

Received 6 March 2006; Accepted 1 June 2006

KEY WORDS: authentication; authorization; virtual organization; Shibboleth; PERMIS; smart tokens

1. INTRODUCTION

A virtual organization (VO) refers to a group of users and service providers, regardless of their geographic locations and administrative domains, who work together on some joint project or

*Correspondence to: N. Zhang, School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.
†E-mail: nzhang@cs.man.ac.uk

Contract/grant sponsor: Joint Information Systems Committee (JISC) FAME-PERMIS Project

share resources (data, services and CPU cycles) through communication infrastructures such as the Internet. Grid computing [1], the enabling technology for VOs, is aimed at providing users with easy and transparent accesses to resources. Depending on the nature of the shared resources, existing Grids may be divided into three types.

- A computational Grid—a virtual supercomputer that dynamically aggregates computing power from multiple providers providing a platform for high-performance and high-throughput applications.
- A data Grid—a cluster of distributed and specialized data repositories and replica systems that store a large number of datasets supporting data-intensive applications. Users can access data as if they are from the same location.
- A collaborative Grid—a virtual environment enabling disparate organizations pursuing common goals of collaborative work. Examples are virtual laboratories or remote control and management of equipment and instruments.

A prominent characteristic of existing Grids [2–4] is that they are of 'closed' nature. They are typically designed for stable, persistent and medium to long-term collaborations, or assume that users of the same VO use the same security credentials and/or enjoy the same level of trust. To access resources, an organization will have to join a VO federation *a priori*, and its users will need to have their credentials established and access roles defined prior to the use of the Grid resources. Often authorization is simply based on the authenticated name of the user. Advanced access-control features such as dynamic delegation of authority are typically not available.

Recently, it has been proposed that Grid facilities should be accessible not only to 'an elite few', but also to the 'general masses' [5]. Individuals, even those not belonging to a VO member institution, should be able to access or contribute resources, and collaborate with fellow peers in a Grid environment. Such collaborations may be short-lived and/or may not be client–server based. In other words, an individual with ample or spare computing power should be allowed to join Grids as a service provider, and at the same time as a service consumer. This vision of *ad hoc* and peer-to-peer (P2P) collaborations should be supported by *federated Grids* that not only support pre-defined or 'closed' collaborations, but also dynamic, pervasive and *ad hoc* collaborations that are expected in the near future. Obviously, the security design for this federated Grid environment imposes some interesting challenges. Firstly, different VO members, or collaborating entities, can no longer assume the same level of trust towards each other, as existing Grids do. There is therefore a need for a dynamic and adaptive security framework that incrementally builds trust among participating entities. Secondly, for scalability reasons, access to Grid resources should not be based solely on the authenticated names of the users, but rather on their attributes, so that common groups of users can be given the same access rights. Thirdly, access should be controlled based on the trust level that a service provider has towards a particular (group of) consumers. Developing such an adaptive security framework requires the design of a number of security building blocks including adaptive authentication and authorization methods, reputation management and trust evaluation algorithms, and resource usage monitoring mechanisms. This paper sets the first step towards achieving this vision by proposing an adaptive authentication framework so as to achieve policy-driven, fine-grained authorization and access control conditional on the strength of authentication undergone by the user.

In detail, Section 2 reviews existing Grid security architectures focusing on authentication services. Section 3 discusses the need for linking access privileges to authentication assurance levels.

DOI: 10.1002/cpe

Section 4 proposes our flexible authentication and fine-grained access-control framework, the Flexible Authentication Middleware Extension to PERMIS (FAME-PERMIS) architecture and its integration within the Shibboleth infrastructure. Section 5 describes the FAME system prototype with an exemplar Java-Card-based authentication service (AS). Finally, Section 6 gives our conclusions.

2. EXISTING GRID SECURITY SOLUTIONS

Our vision is to have an adaptive authentication framework that can authenticate a user using a variety of authentication tokens and protocols. These various authentication tokens and protocols provide different levels of assurance in identifying a Grid user. Access privileges granted to the user should be linked to the assurance level of the authentication token/protocol used in the particular authentication instance. Such a linkage is necessary for the provision of fine-grained access control and privilege allocation in Grid environments in which the same or different applications may have dissimilar authentication requirements as dictated by varying levels of resource sensitivity and access mode towards different groups of users. For example, services such as e-journal subscription or e-learning services may have a relatively low sensitivity level and therefore can be accessible to everybody who can be identified by the IP address of his/her machine. Tutors/examiners may need to use a stronger form of authentication than that used by students in order to access, say, exam papers, as the former bear more responsibility with regards to the confidentiality and integrity of the data resource. Similarly, in a health Grid context, electronic patient records (EPRs) and electronic health records (EHRs) are shared among GPs, clinicians, and clinical and biomedical researchers across different institutions and organizations. EPRs/EHRs have high levels of privacy requirements due to legal and ethical reasons. Therefore, it is usually expected that EPRs/EHRs are de-personalized and sensitive information that can be used to identify the owner of a record are removed, before being released to entities outside hospital premises or before researchers are allowed to access them [6]. Password-based authentication methods may be sufficient to identify researchers when accessing these de-personalized records. However, the suppliers of the records, e.g. GPs and hospitals, should use a stronger form of authenticators when uploading new records into the de-personalized data repository due to privacy and accountability concerns. Therefore, there is clearly a need for a fine-grained access-control framework to satisfy the complex VO access-control requirements, and one important element of the access-control decision making is the authentication strength of the authenticator used by the user.

Existing Grid security solutions do not provide such a framework. Currently, Grid authentication is mainly achieved through the use of digital certificates. For example, the Grid Security Infrastructure (GSI) [7], the *de facto* security standard in the Grid community, relies on the use of the Public Key Infrastructure (PKI), X.509 certificates and access-control lists (ACLs) for entity authentication, authorization and access control to Grid services [8]. Privilege delegation to a Grid job and single sign on (SSO) are made possible through the use of a X.509 proxy credential (consisting of a proxy key pair and proxy certificate). A proxy certificate allows the delegation of a subset of a user's rights to his Grid job by creating a temporary key pair and a subordinate identity called a proxy. A proxy certificate is a short-lived certificate given to the Grid job, and signed by the user, or by a previous proxy in the case where the original job spawns a subtask. It is this chain of credentials that enable a remote entity to operate or perform tasks on behalf of the original user. The control on the use of a proxy credential is achieved by limiting the life span of the proxy certificate and ensuring

DOI: 10.1002/cpe

that the proxy name is subordinate to the user's name. This controlled credential delegation and certificate-based authentication scheme assumes that different VO entities, including service providers and intermediaries, either all trust each other unconditionally or not at all. There is no support for conditional trust. The certificate-based 'one-method-fits-all' authentication method and homogenous trust model are obviously not suited to the vision of federated Grids.

Another notable VO security infrastructure is Shibboleth [9]. It defines a security infrastructure and protocols for inter-institutional sharing of Web resources that are subject to authentication and access control. The Shibboleth infrastructure uses the concept of federated administration, in which institutions are assumed to be mutually trustworthy and users are identified in their respective home institutions (i.e. origin sites). On authentication, the origin sites assert the users' attributes through the use of the Security Assertion Markup Language (SAML) [10]. These signed attribute assertions are then dispatched by the origin sites to the service providers (i.e. target sites), which then use them to make access-control decisions for their resources. The Shibboleth framework itself does not address the issues of authentication, authorization and access control, although it does provide a simple access-control mechanism. It primarily provides a framework and the related protocols to support inter-site virtual attributes passing, and leaves the implementations of the authentication and authorization services to the respective origin and target sites. PERMIS [11] is a policy-driven authorization infrastructure that provides fine-grained, role-based access control (RBAC). Policies are written in XML, and interpreted by a policy decision point (PDP) that grants or denies access based on a user's attributes (or roles) and various conditions such as time of day or parameters of the request. PERMIS has already been integrated with Shibboleth [12] and the Globus Toolkit [13]. The FAME-PERMIS system described in this paper can be incorporated into the Shibboleth infrastructure to provide fine-grained access control based on the strength of authentication undergone by the user.

3. LINKING AUTHENTICATION STRENGTH TO ACCESS PRIVILEGES

Authentication factors are typically grouped into three categories [14]: knowledge-based, such as passwords; token-based, such as memory or smart tokens; and ID-based, such as biometrics. These authentication factors have different strengths providing different levels of assurance (LoA) in identifying a user. For example, a password is easier to forge than a smart token equipped with a cryptographic key because the former is normally easier to guess. Although biometrics are more difficult to forge, alone they cannot be used for remote electronic authentication due to the lack of secrets. To achieve a higher LoA, two or more authentication factors can be combined together to identify a user. A smart token locked with a fingerprint or a personal identification number (PIN), which is a two-factor authenticator, is a better choice than using an unlocked token alone as it is more susceptible to theft or loss.

Authenticators and their associated LoAs have been classified into four levels in a specification published by the NIST (U.S. National Institute of Standards and Technology) [15] according to the likely consequences of an authentication error when using each of them. As shown in Table I, Level 1 authenticators have the lowest LoA, whilst Level 4 have the highest. To compromise a Level 4 authenticator, say a smart card token locked with a PIN number, the perpetrator would first have to obtain the card and, then work out the PIN number. It therefore provides a higher LoA than a soft token

DOI: 10.1002/cpe

Table I. Authenticators and their levels of assurance.

	Level 1	Level 2	Level 3	Level 4
Hard token	×	×	×	×
Soft token	×	×	×	
One-time password device	×	×	×	
Strong passwords	×	×		
Passwords and PINs	×			

Table II. Authentication protocols and their levels of assurance.

	Level 1	Level 2	Level 3	Level 4
Private key proof-of-possession protocol	×	×	×	×
Symmetric key proof-of-possession protocol	×	×	×	
Zero-knowledge password protocol	×	×		
Tunnelled password protocol (e.g. password over SSL)	×	×		
Challenge-response password protocol	×			

such as a cryptographic key stored in a file. Our FAME-PERMIS system is aimed at integrating all of the authenticators shown in Table I and protocols from Table II.

4. FAME-PERMIS AND SHIBBOLETH INTEGRATION

4.1. FAME-PERMIS overview

The FAME-PERMIS system consists of two major components: the FAME component and the PERMIS component. FAME is a multi-level and multi-factor authentication framework designed to support the use of a whole range of authenticators and to integrate various ASs with a Web-based front-end. In addition, based on the authenticator and the authentication protocol used in an authentication instance, FAME derives the authentication strength, or LoA, and feeds it to a remote authorization decision engine, i.e. PERMIS, that uses the LoA as part of its access-control decision making. In the implementation presented in this paper, the two components are integrated into the Shibboleth infrastructure, so that the LoA is passed using the Shibboleth inter-site message-passing protocol.

In detail, the FAME system is designed to have the following functionalities.

(1) Facilitate controlled access to distributed resources.
(2) Allow a user to choose among a list of supported ASs (or servers).
(3) Invoke an appropriate AS based on the user's choice and assign the user a LoA.

DOI: 10.1002/cpe

(4) Pass the LoA via Shibboleth to the remote PERMIS authorization engine.
(5) Provide SSO functionality, such that if a user hits the FAME system again in the same session after the initial authentication, the user does not need to present his/her credentials again in order to re-authenticate. In other words, if a user has gone through an initial authentication process successfully, the user will be issued with a valid authentication token by FAME. This token will then enable the user to access the same service multiple times or different distributed services in the current session assuming that (i) the session has not expired and (ii) the user's authentication LoA is not lower than that required by the sites involved.

Owing to the modular nature of the design, both FAME and PERMIS can work alongside each other, or independently, in supporting controlled access to Web-based services. Both can also be independently integrated into the Shibboleth infrastructure to allow institutions to authenticate their users with a variety of methods, and to finely control access to their resources via an appropriate RBAC policy.

4.2. The PERMIS infrastructure

The PERMIS infrastructure has been constructed in a modular fashion as shown in Figure 1. This shows how different components of PERMIS can be used in a Grid environment to add authorization capabilities to both the sending and receiving sites. Prior to any computer interactions taking place, step 0 requires the various managers (or sources of authority (SOA)) to negotiate amongst themselves and agree the various policies that will control access to their pooled resources. Once this has been agreed, they can use the PERMIS policy editor to construct their policies in XML [16]. Writing XML is beyond the ability of most managers, therefore the policy editor provides an easy-to-use GUI that allows non-experts to intuitively construct their policies [17]. A PERMIS policy comprises two parts, a role assignment policy (RAP) that says who is trusted to assign which attributes to whom, and a target access policy (TAP) that says which attributes are needed to access which resources under what conditions. (Some of these conditions will subsequently be used to evaluate the LoA produced by the FAME system.) The credential validation service (CVS) evaluates all received credentials against the RAP (step 8), discards untrusted ones and passes all validated attributes to the policy enforcement point (PEP) (step 9). The PEP in turn passes these to the PDP (step 11), along with the user's access request (step 7) and any environmental parameters (step 10). The PDP makes an access-control decision based on the TAP, and passes its response back to the PEP (step 12).

At the sending side, there might also be a PEP that intercepts each user's outgoing request (step 1), and asks the local PDP to decide whether this is allowed (step 3). The PDP makes its decision (step 4), and may optionally inform the outgoing PEP of some obligations that should be placed on the outgoing request. These obligations could contain details of the correct set of credentials that should accompany the request, in order for the user to be granted access to the target resource. This of course will be known by the subject SOA from the initial negotiations at step 0. The PEP can then request the appropriate credentials from either a credential-issuing service (if they do not already exist), or from an attribute repository (if they already exist). Several credential-issuing services already exist today. In Shibboleth (see later) it is termed the Attribute Authority (AA), in PERMIS there is the Delegation Issuing Service (DIS) and in some Grid systems there is the VO Membership Service (VOMS) [18]. Once the user's request has the appropriate credentials, they can all be sent to the remote site.

DOI: 10.1002/cpe

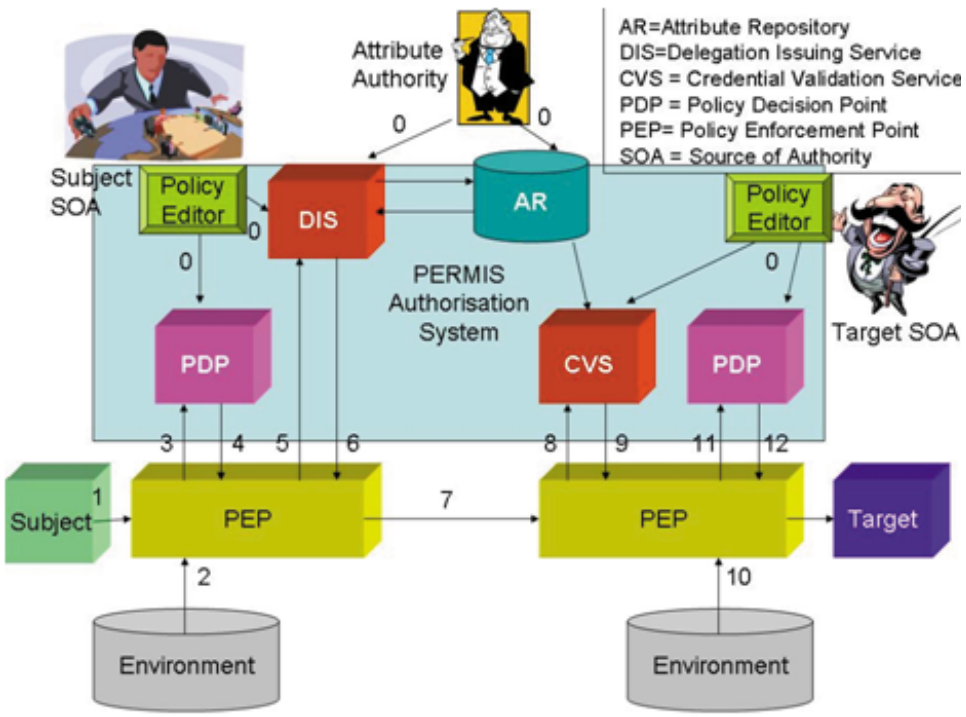


Figure 1. The PERMIS authorization infrastructure.

In traditional Grid systems, the credentials are inserted into the proxy certificate. In Shibboleth, the remote site asks for the user's credentials, rather than requiring the user to send them with each request.

There are additional features supported by PERMIS that are not described here, for example, the ability to push or pull credentials, and to digitally sign policies and store them in a Lightweight Directory Access Protocol (LDAP) directory. These are described more fully in [11].

4.3. The Shibboleth infrastructure

The Shibboleth infrastructure provides a secure means for asserting and passing information about their users across multiple institutions in a VO. An inter-site interaction occurs when a user from an origin site attempts to gain access to a resource or service on a target site through his/her Web browser. On the origin side, Shibboleth has two architectural components: the Handle Service (HS) and the AA. The HS is an application responsible for creating a handle for the user, and is protected by a local authentication service used by the origin site. The handle, a reference number pointing to the user, will be used by the AA to identify the attributes of the user, which are stored in a local repository,

DOI: 10.1002/cpe

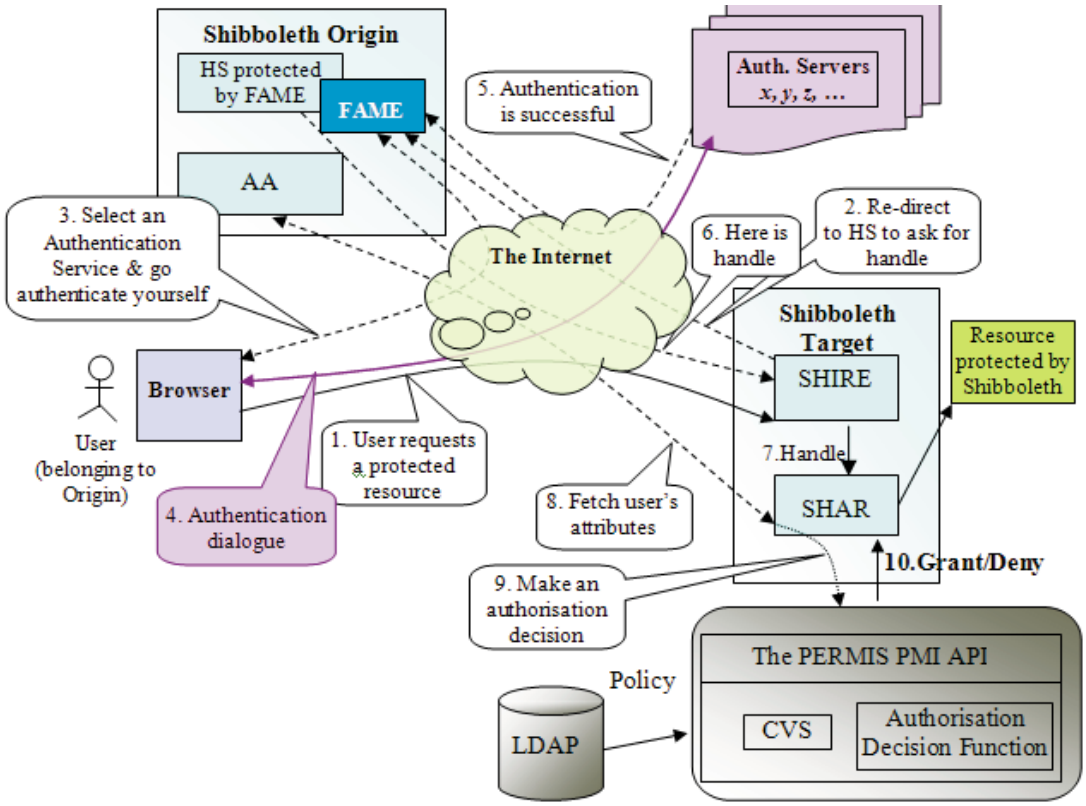


Figure 2. The Shibboleth architecture.

normally a LDAP directory. As shown in Figure 2, FAME can be plugged into the Shibboleth origin stack acting as the local authentication service to protect the access to the HS.

On the target side, Shibboleth has three main components: the Shibboleth Indexical Reference Establisher (SHIRE), the Shibboleth Attribute Requester (SHAR) and the basic Shibboleth authorization function called ShibAuthz (not shown). The SHIRE monitors users' requests for resources on the target. If the user has yet to be authenticated by his origin site (as determined by the absence of a handle), he/she will be directed back to the origin site for authentication. Once the user is authenticated and returned back to the target with the handle attached to a cookie, the SHAR uses the handle to fetch the user's attributes from the AA at the origin. When the attributes are obtained, they are passed to the PERMIS authorization decision engine for validation and an access-control decision based on the user's valid attributes and the local access-control policy. As illustrated in Figure 2, inter-component interactions are further explained below.

DOI: 10.1002/cpe

- *Step 1.* User accesses a resource on the target using his/her Web browser.
- *Step 2.* The SHIRE component intercepts this request and detects that the user does not carry a handle inside the cookie, so it redirects the user to his/her origin site's HS.
- *Step 3.* The HS component at the origin is protected by a local authentication service, thus the user has to authenticate before being allowed to access the HS. Alternatively, if the origin has an SSO authentication system and the user has been authenticated previously in the same session, the user will be permitted to access the HS straight away. If FAME is used, then the user is presented with a list of available authentication services for the user to choose from. Once the choice is made, the user will be re-directed to the chosen AS.
- *Step 4.* The user performs authentication with the chosen AS.
- *Step 5.* On successful authentication, the user is redirected back to FAME.
- *Step 6.* This time, the user is allowed through to the HS that creates a handle for the user and redirects the user back to the target with the handle contained in the cookie. The SHAR at the target uses the handle[‡] as the reference number to fetch the user's attributes from the AA at the origin site.
- *Step 7.* When the user hits the SHIRE again, the SHIRE detects the handle and passes it to the SHAR.
- *Step 8.* The SHAR uses the handle to fetch the user's attributes from the AA, which may be signed and packed inside a SAML message, but by default Shibboleth uses the Secure Socket Layer (SSL) to protect the transfer of the attributes.
- *Step 9.* The SHAR passes the attributes to the PERMIS engine, which validates that the sending site is trusted to issue these attributes. It then makes an access-control decision based on the user's valid attributes and the target's access-control policy. If the decision is 'allow', then the user's access request is granted.

As can be seen from Figure 2, the integration of FAME into the Shibboleth architecture is through the Shibboleth HS, while the integration of PERMIS is through the Shibboleth SHAR component. Thus, both components can be independently integrated. FAME controls access to the HS, PERMIS controls access to the target resource. Only FAME-authenticated users are allowed to go through to the HS. Only PERMIS authorized users are allowed to go through to the resource. In addition, FAME passes the 'name' of an authenticated user back to HS via the REMOTE_USER environment variable for further use by the HS in creating the handle. This 'name', for instance, can be the user's username, Distinguished Name, Kerberos principal name, etc., depending on the specific authentication method used.

4.4. The FAME design

FAME, as shown in Figure 3, consists of two internal components and is compatible with any AS with a Web front-end. The two internal components are:

(1) FAME SSO Checker (F-SSO); and
(2) FAME Login Server (F-LS).

[‡]The reason for Shibboleth to pass a handle for an authenticated user instead of passing the user's real identity to a target site is to preserve the privacy of the user's identity.

DOI: 10.1002/cpe

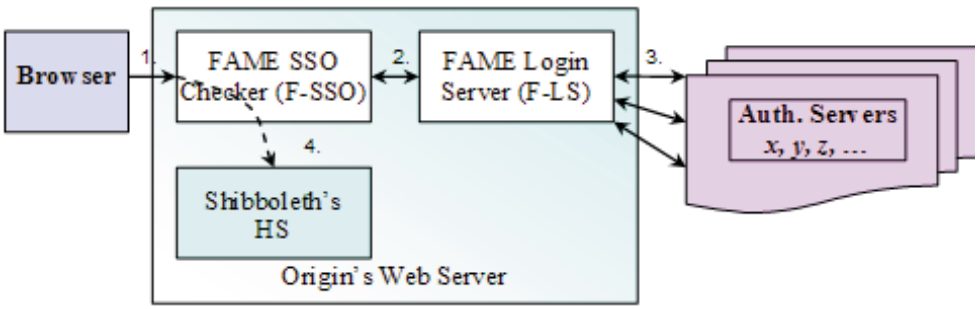


Figure 3. The FAME system.

F-SSO plays the role of an authentication enforcer: it protects the Shibboleth HS making sure that all of the users are authenticated before being allowed to access the HS. In addition, it provides the SSO functionality by issuing SSO cookies to users who have been successfully authenticated in the current session. When a user accesses the HS, the F-SSO component intercepts the request to check whether there is a SSO cookie in it. If the request contains a SSO cookie, which means that the user has already been authenticated previously in the current session, then the access will be granted and the session continues. Otherwise, the user will be redirected to the F-LS for authentication. On a successful authentication, F-SSO issues an SSO cookie to the user, and passes the name of the user to the HS via a REMOTE_USER environment variable. The HS can then create a handle for the named user. Next time when the same user hits the HS again in the same session, the SSO cookie will be detected, and the user will be let through without re-authentication. The default validity period of a SSO cookie is set to 8 hours.

F-LS is responsible for redirecting a user coming from F-SSO to his/her preferred AS. It presents users with a page where they can choose their preferred methods of authentication. Once a user makes the choice, the F-LS redirects the user to the selected AS. In addition, after the authentication is performed, F-LS calculates an LoA based on the authentication token and the protocol used in the authentication instance. F-LS maintains a list of configured authentication servers against their LoAs. The calculated LoA for the user needs to be passed through Shibboleth to the target site, and the PERMIS decision engine will use the LoA value along with other attributes of the user to make an access-control decision.

FAME is compatible with any AS as long as it has a Web-based front-end. Examples of such ASs are Kerberos, Network Information Service (NIS), LDAP, MySQL-based services, SSL authentication with PKI soft and hard tokens, etc. The supported ASs can be local or remote. The design also makes sure that it can work with emerging or future authentication systems. FAME can also be used independently from Shibboleth to protect any Web-based resources protected with authentication facilities.

DOI: 10.1002/cpe

4.5. FAME integration with Shibboleth

F-SSO and F-LS are both implemented as Apache Web Server modules. An Apache Web Server together with a Tomcat servlet container hosts the Shibboleth HS that is implemented as a Java servlet. For clarity, let us assume that the HS is hosted at the URL address https://my-host/shibboleth/HS. To integrate FAME with Shibboleth, the `myhost` Apache's configuration file should have an entry similar to the following:

```
<Location/shibboleth/HS>
          AuthType Fame
          SetHandler perl-script
          PerlResponseHandler MyApache2::Fame::sso_checker
             ...
<Location/>
```

To set up F-LS, a similar set of code to that below should be inserted into the Apache Web Server's configuration file:

```
<Location /fls>
         SetHandler perl-script
         PerlResponseHandler MyApache2::Fame::login_server
         ...
</Location>
```

All of the requests and responses sent among the user's Web browser, F-SSO, F-LS and authentication servers are SSL-protected to prevent FAME SSO cookies and users' credentials from being eavesdropped by perpetrators who could then use them to impersonate authorized users. F-LS and F-SSO share a secret key, `FLS_SSO_KEY`, used for authenticating cookies passed between these two components. F-LS shares a unique key, `FLS_AS_KEY`, with each of the ASs for encrypting tokens passed between them as URL parameters.

The ASs that have been integrated with FAME so far include Apache's Basic Authentication using username/password pairs, Kerberos (using Apache module `mod_auth_kerb`), LDAP (using Apache's `mod_auth_ldap`) and SSL authentication using client certificates (with Apache's `mod_ssl`). To demonstrate the flexibility and efficacy of the FAME design, we have developed a custom-built Java Card Web Authentication Service (JC-WAS) and integrated it with FAME. Section 5 details the design of this Java-Card-based authentication service and its integration with FAME-PERMIS.

4.6. PERMIS integration with Shibboleth

Integration of PERMIS and Shibboleth is performed via the Apache Web Server. The Shibboleth's SHIRE, SHAR and the basic default authorization capability ShibAuthz are all (logically) packaged inside a single Apache module called `mod_shib`. Apache must be configured to call `mod_shib` during the authentication and authorization phases of handling a user's request to access a Web-based resource. The PERMIS infrastructure provides a similar module called `mod_permis`, which logically integrates the PERMIS authorization decision engine into Apache. Apache must be configured to call

DOI: 10.1002/cpe

`mod_permis` prior to `mod_shib` during the authorization phase only. In this way PERMIS can make an authorization decision prior to and instead of `mod_shib`. Each Apache module must return either OK, Declined or an Error Code when called by Apache. In the first and last cases, Apache knows that the phase has been completed, and therefore does not call any further modules in its list for that phase. In the Declined case, Apache knows that the phase has not yet been completed, and so it calls the next module in its configured list. A full description of the integration of PERMIS and Shibboleth can be found in [12].

4.7. FAME integration with PERMIS

FAME computes a LoA, and this must be carried by Shibboleth to PERMIS, in order for the access-control decision to vary according to the LoA. There are a number of ways of achieving this transfer, each with their own advantages and disadvantages. In the first approach, the F-LS may pass the LoA to the target site as a cookie along with the user handle. Whilst this is simple to engineer at the user's home site, it is difficult to engineer at the target site since the LoA has to be merged with the other attributes passed by Shibboleth. The SHAR component would have to be modified to take the LoA from the cookie and pass it to the PERMIS decision engine.

In the second approach, the F-LS stores the user's LoA in the user's LDAP entry. Then when the target site's SHAR asks the origin's AA for the user's attributes, the LoA is returned along with the user's other attributes.

In the third approach, a custom data connector (CDC) can be built between the F-LS and the Shibboleth AA. The Shibboleth AA can be configured to release multiple attributes from multiple data connectors, so that when a Shibboleth target SHAR requests the attributes of a user from the origin's AA, the AA sees what attributes should be released and looks up which data connectors are needed for this. The AA contacts each of these data connectors in turn to release the necessary attributes. It then merges all of the attributes together prior to returning them to the SHAR. The CDC facility is a standard Shibboleth extensibility feature for retrieving attributes from a data provider that does not have a standard data-connector implementation (i.e. LDAP, SQL database, etc., which come with the Shibboleth software). By building a CDC to the F-LS we will be able to return the LoA to PERMIS via Shibboleth, without needing to store it in the user's LDAP directory entry.

There are several advantages of the use of the CDC approach. Firstly, a user can have several connections open simultaneously to different Shibboleth target sites, using different authentication methods for each, and each will get the correct LoA. With the LDAP model, only one LoA attribute can be created in the user's LDAP entry, so in order to cater for multiple connections, the LoA attribute values would have to be specially encoded with a combination of the handle and the LoA, e.g. as `handle:LoA`. This is rather messy, although not unusual for LDAP applications. The second advantage is that, with this approach, the LoA is automatically garbage collected (i.e. it is forgotten by the F-LS) when the user logs off or the timeout expires. With the LDAP model, however, there is no clean way of removing the LoA attribute from the user's LDAP entry once the user logs off. With the CDC approach, we simply leave it there and overwrite it when the user logs on again. Thirdly, this approach offers a better performance than the LDAP approach. Network operations are required in order to store the LoA in the LDAP directory. CDCs, on the other hand, are direct calls from one process to another.

DOI: 10.1002/cpe

The CDC approach, however, does have the following two disadvantages. Firstly, it will not work if the F-LS server and the AA are on different machines, since it requires direct calls between the processes. The LDAP model will work in this configuration, since it assumes a distributed system and a network-accessible LDAP server. Secondly, it will not work if the LoA is to be provided by a non-Shibboleth attribute retrieval mechanism, e.g. by the CVS operating in pull mode and fetching them itself.

Both mechanisms require the PERMIS TAP to be constructed in such a way that the LoA is treated as a user attribute just like any other attribute, such as role, affiliation, department, etc. Furthermore, in order to allow a user with a higher LoA, say 3, to have more privileges than a user with a lower LoA, say 1 or 2, the LoA attribute needs to be defined as a hierarchical attribute. In just the same way as other attributes/roles, such as doctor > healthcare professional, can be defined according to the hierarchical RBAC model, so can LoAs, for example, LoA 4 > LoA 3 > LoA 2 > LoA 1. In this way, a policy can be written such as 'subjects with the following attributes: affiliation = St Mary's Hospital, LoA = 4, role = doctor may write to Patient Records; and subjects with the following attributes: affiliation = St Mary's Hospital, LoA = 1, role = healthcare professional, may read Patient Records'. Then a doctor with an LoA of 2 would be able to read a patient's records, but not be able to write to it.

To evaluate the efficacy of our FAME-PERMIS framework, we have prototyped an AS with the LoA value of 4, the Java Card authentication service, and integrated it along with other custom ASs into the framework. The next section describes our efforts in developing this high-level assurance AS and its integration with FAME-PERMIS.

5. LINKING FAME WITH JC-WAS

5.1. JC-WAS overview

A Java Card is a special smart card that is embedded with a microprocessor and a memory chip and can execute on-card applications written in Java. These on-card applications are called Java Card applets. A Java Card can be loaded with multiple applets, which can safely reside and execute on the same card without interfering with each other's code or data.

Smart cards are a good way of storing secrets and private keys and are increasingly used for user authentication. When used in conjunction with public key cryptography, the key pair is generated inside a Java Card, and the public key is exported to a Certification Authority (CA) for key certification. The private key never needs to come out of the card. Once the certificate is generated and signed, it is loaded back into the card. The Java 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, hash operations and digital signature generation and verification. Smart cards are generally tamper resistant and, in order to impersonate the card's owner, an attacker must get hold of his smart card as well as work out the PIN used to lock the card. For this reason, smart cards provide the highest LoA according to the NIST Electronic Authentication Guideline [15]. The Java Card used for this development is the Schlumberger Cyberflex e-gate smart card with 32K EEROM and a cryptographic co-processor, together with the Schlumberger USB smart-card reader.

The card users are assumed to have subscribed to some credential service provider (CSP), such as VeriSign, and their identities verified by a registration authority (RA) that may be the CSP or a separate

DOI: 10.1002/cpe

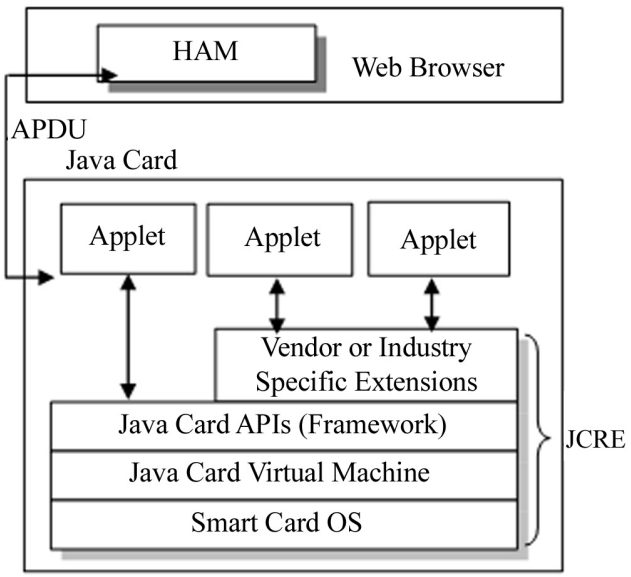


Figure 4. The Java Card architecture.

organization trusted by the CSP. On successful identity verification, the users are each issued with a Java Card by the CSP.

The JC-WAS AS consists of a set of client components and a JC-WAS servlet.

5.2. The client components

Java Cards have a well-defined software architecture [19], as shown in Figure 4. The card's operating system (OS) is situated in the bottom layer of the stack. 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 applications, or Java Card applets, are located on the top-most par of the architecture. More than one applet can be executed on one card, and each applet is uniquely identified by an application identity (AID) [19]. All of the applets must extend the Applet Abstract Base class that defines the methods to be used by the JCRE to control an applet lifecycle. The communication between an off-card application and an on-card application is achieved through the use of *command* and *response* Application Protocol Data Units (APDUs) [20].

DOI: 10.1002/cpe

Three Java applications have been developed for the purpose of JC-WAS on the client side: the *Certificate Issuance Application* (CIA) module, the Host Authentication Module (HAM), and the *Java Card PKI Authentication Application* (PKIA^2) module.

5.2.1. CIA module

The CIA module is an off-card standalone application run on the client's host machine. It performs all of the functions needed for public key certification. The public key of a Java Card has to be certified before the card can be used with the JC-WAS system. To certify a public key, the following steps are necessary.

(1) A public and private key pair is first generated inside the card. This key generation is commanded by CIA. The private key never needs to come out of the card.
(2) After the key generation, the public key is retrieved from the card. To retrieve the key, a method is needed to separate the public key object into two components, the Exponent and the Modulus. Then these components are retrieved separately from the card. Once out of the card, another method is needed to reconstruct the components back to the public key object before step (3) can be performed. The CIA module includes all of these methods, and the key retrieval operation is performed by the CIA through its interaction with the PKIA^2 module.
(3) Then CIA forms a 'CertificationRequestInfo' object that contains the public key just retrieved and reconstructed along with other attributes of the card, including the subject's (i.e. the Java Card's) distinguished name and the certificate expiry date.
(4) CIA sends the 'CertificationRequestInfo' object back to the card for it to be signed by the card with the corresponding private key.
(5) CIA retrieves the signature from the card, and generates a 'CertificationRequest'. The 'CertificationRequest' consists of the 'CertificationRequestInfo' object, the signature on the object, and the signature algorithm identifier.
(6) CIA dispatches this 'CertificationRequest' to a remote (or local) CA that certifies the enclosed public key by generating a proper X.509 certificate for the key.

As indicated above, there are two ways of certifying a public key. One is to use a remote CA, and the other is to certify it locally using some third-party software such as the Java Cryptography Provider [21]. This second key certification method is more suitable when a security domain wishes to locally issue cards equipped with certificates to its users.

5.2.2. HAM

The HAM module is a Java applet executed inside a Web browser on the client machine. It is responsible for user's local authentication with the Java Card locked with a PIN and, in addition, acts as a gateway passing data from the Java Card to the JC-WAS servlet, and *vice versa*. In general, applets have less privileges than Java applications and they cannot access local resources, e.g. read and write files on the client file system, or access the smart-card reader without special privileges. Thus, in order to access the card, HAM needs to be digitally signed by its creator and the client's Web browser needs to be set to trust the applet's signer, as the digital signature provides a means of verifying the originator and the integrity of the applet.

DOI: 10.1002/cpe

The communication between the HAM and the Java Card was initially implemented using the vendor's (i.e. Schlumberger's) middleware API, called Axalto Middleware API [22]. Later on, to make the solution vendor-independent, this API was replaced by the Open Card Framework (OCF) API [23], although the OCF was discontinued after version 1.2 was published in 2000.

5.2.3. Java Card PKIA^2 module

The PKIA^2 module, a Java Card applet executed inside the Java Card, interacts with and is commanded by the CIA module to perform the public key certification function mentioned above. In addition, it interacts with and is commanded by the HAM module to perform user-to-card and card-to-server authentication functions. For example, the four methods invoked by the CIA module for the purpose of public key certification are: `PIN_Validate()` for ensuring that the CIA application can only invoke the methods after successful user-to-card authentication; `Retrieve_PublicKey()` for retrieving the public key from the Java Card; `Build_CertificateReq()` for constructing a certificate request; and, finally, `Import_Certificate()` for putting the signed certificate back into the card.

The interaction between an on-card application, i.e. the PKIA^2 applet, and an off-card application, i.e. CIA or HAM, is accomplished through sending ADPU commands to the card by the off-card application. For example, for the public key retrieval operation, the CIA module sends an APDU command to the PKIA^2 applet invoking the `Retrieve_PublicKey()` method of the PKIA^2 module, which, on receiving the command, will execute the method and send the public key back to the CIA module using an APDU response message. A problem experienced with this particular card protocol type was that, when we were to download a certificate back to the card, the actual size of the certificate was 527 bytes, i.e. much longer than the APDU buffer size of 255 bytes. For this reason, multiple APDU commands are necessary to import a single certificate to the card, and the exact offsets of the certificate array have to be calculated in order to correctly import the certificate. Other Java Card protocol types may not have this problem.

5.3. The JC-WAS authentication servlet

The JC-WAS servlet runs on an AS. It communicates with the client-side component, HAM, and implements the server-side authentication logic using the public-key cryptographic challenge–response protocol. It is developed using the Java servlet [24] and Java Server Page (JSP) [25] technologies. As shown in Figure 5, to perform a card-to-server authentication, the servlet generates a challenge and sends it to the card via the HAM module. When the response is returned, it verifies the signature using the public key certified in the certificate. The JC-WAS server uses standard HTTP and SSL protocols to communicate with the HAM module run in the Web browser.

5.4. Observations

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 an effort in the standardization of smart/Java card application development in 1997, when computer manufacturers, solution providers, card manufacturers and card-reading device manufacturers organized a consortium to work on an open Java framework for smart-card access.

DOI: 10.1002/cpe

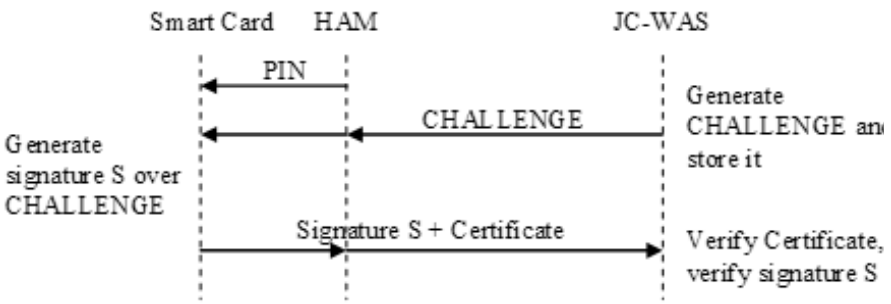


Figure 5. Authentication protocol using public-key cryptography.

The resulting framework was called the OCF [23]. 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 [20] devices on different target platforms such as Windows, UNIX workstations, etc. They produced an API that allows a developer to register cards, look for cards in readers and optionally have Java agents start up when cards are inserted into the reader. However, since 2000 the OCF is no longer active. Although there was no official announcement, the OCF API development has stopped since then; the last version released was 1.2. The trend seems, at the moment, to be to use vendors' middleware APIs.

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 on-line Web service provided by Verisign [22]. However, this method is only available to users who use Microsoft CryptoAPI [26] or RSA Cryptoki [27]. In addition, due to the Java Card firewall facility, a certificate written into a card using this method cannot be access by other applets. This means that, unless you use Microsoft CrytoAPI or RSA Cryptoki, this on-line certification service is not available.

5.5. JC-WAS and FAME integration

The integration of JC-WAS and FAME is done as follows. Firstly, the FAME-Apache assembly and the Tomcat Server hosting the JS-WAS servlet must be installed and configured to interoperate. The JS-WAS servlet plays the role of one of the authentication servers supported by the FAME system (as previously illustrated in Figures 2 and 3). Next, the FAME system has to be configured to include JS-WAS as one of its linked authentication servers. This is done by inserting the necessary information that FAME needs to know about JC-WAS into the FAME database. The table structure where the record is inserted and its exemplar values are given in Table III.

In detail, FAME needs to know the URL of JC-WAS, the authentication type of JC-WAS, the LoA it provides and a secret key, `FLS_AS_KEY`, shared between FAME and JS-WAS. After these initial settings, FAME is ready to support JC-WAS and will re-direct users to it if the users choose to use it.

DOI: 10.1002/cpe

Table III. Information about JC-WAS needed by FAME.

Authentication Server's URL	AS's authentication type	Level of assurance provided by AS	Base 64-encoded shared secret (`FLS_AS_KEY`)
https://my-host/jc-was	smart card	4	bGtoa2RzaGZrc...eXBa

The secret key, `FLS_AS_KEY`, is used for JS-WAS to authenticate itself to FAME and for the secure passing of authentication-related parameters between them. On successful authentication, JS-WAS will re-direct the user back to the FAME's F-LS component, which, in turn, will return the user back to the resource the user was requesting, i.e. the Shibboleth HS. The flow of interactions between FAME and JC-WAS are summarized in Figure 6.

6. CONCLUSIONS AND FUTURE WORK

In this paper, our work on the design of a multi-level and multi-factor authentication framework to achieve fine-grained access control, FAME-PERMIS, has been presented. Its implementation in the context of Shibboleth has also been discussed, and we have shown how access-control decisions can be based on the users' authentication assurance level as well as their other attributes. As a proof of concept of the FAME authentication framework, a Java-Card-based AS has been prototyped and the prototype has been integrated with the authentication framework to achieve the highest level of authentication assurance. The work has demonstrated the feasibility of using such an authentication framework, and has also 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.

In our future work, we will continue to integrate other authentication systems and services into our FAME system and to fulfil our vision of adaptive authentication and linking authentication strengths to authorization decisions and fine-grained access control for Grids. In addition, we will also be looking at taking into account other emerging authentication methods in our FAME system design.

Concerning the PERMIS authorization infrastructure, we are currently integrating the concept of obligations, so that the sending PEP can be told which credentials need to be included along with a user's request. We are also adding more sophisticated access-control features such as separation of duties and we are standardizing the PEP–PDP interface so that it can use the XACML request context for passing the authorization decision query. Finally, as sensor networks become more prevalent, we will need a mechanism for transferring context information, such as the user's location, from the sending site to the target site, so that authorization decisions can be based on this, in much the same way as we have used the LoA in the current work.

Finally, further research is needed to find an ideal way of passing LoAs from FAME to PERMIS. We have described three alternative solutions in this paper, but none of these is ideal. At the time of writing, we found out that the next version of Shibboleth will be based on SAMLv2.

DOI: 10.1002/cpe

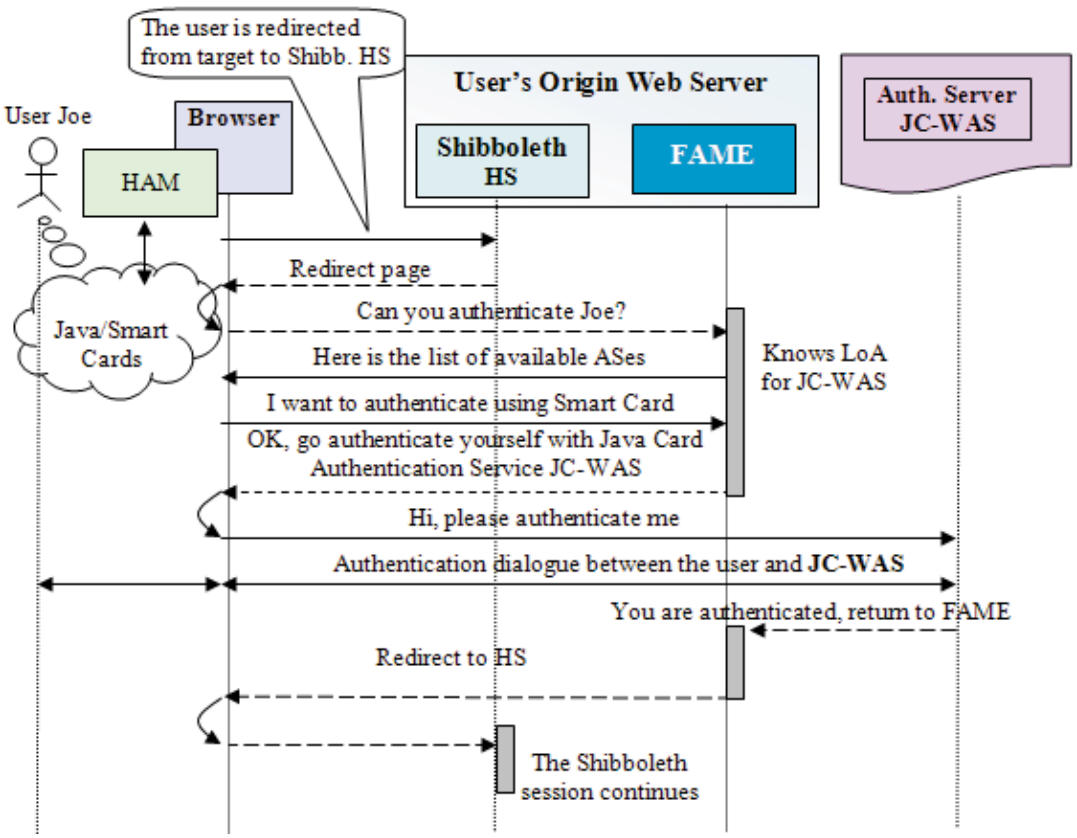


Figure 6. Interactions between FAME and JC-WAS.

As part of this work, a service provider (i.e. target) will be able to ask an identity provider (IdP) at an origin site to authenticate a user to a particular LoA level, using the SAMLv2 protocol elements. The IdP can then return the LoA in the response also using the SAMLv2 protocol. We will closely follow this development. If SAMLv2 is implemented in Shibboleth, we propose to use this standard way of passing the LoA between the service provider and user origin sites.

ACKNOWLEDGEMENT

The work presented in this paper is sponsored by the Joint Information Systems Committee (JISC) as part of the FAME-PERMIS Project.

DOI: 10.1002/cpe

REFERENCES

1. Foster I, Kesselman C (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Francisco, CA, 1998.
2. DOE Science Grid. http://www.doesciencegrid.org/ [October 2005].
3. The DataGrid Project. http://www.eu-datagrid.org/ [October 2005].
4. GriPhyN—Grid Physics Network. http://www.griphyn.org/ [October 2005].
5. Amin K, Laszewski GV, Mikler AR. Grid computing for the masses: An overview. *Proceedings of the Grid Computing Conference (GCC 2003)* (*Lecture Notes in Computer Science*, vol. 3033). Springer: Berlin, 2004; 464–473.
6. Helken H *et al.* De-identification framework. *White Paper*, IBM Haifa Labs, Israel, August 2004.
7. Foster I, Kesselman C, Tsudic G, Tuecke S. A security architecture for computational Grids. *Proceedings of the 5th ACM Conference on Computer and Communication Security*. ACM Press: New York, 1998; 83–92.
8. Butler R *et al.* A national-scale authentication infrastructure. *IEEE Computer* 2000; **33**(12):60–66.
9. Shibboleth. http://shibboleth.internet2.edu/draft-internet2-shibboleth-arch-v05.html [August 2004].
10. OASIS. Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML), 19 April 2002. http://www.oasis-open.org/committees/security/.
11. Chadwick DW, Otenko A. The PERMIS X.509 role based privilege management infrastructure. *Future Generation Computer Systems* 2002; **936**:1–13.
12. Xu W, Chadwick D, Otenko S. Development of a flexible PERMIS authorisation module for Shibboleth and Apache server. *Proceedings of 2nd EuroPKI Workshop*, University of Kent, July 2005 (*Lecture Notes in Computer Science*, vol. 3545). Springer: Berlin, 2005.
13. Chadwick DW, Otenko S, Welch V. Using SAML to link the GLOBUS toolkit to the PERMIS authorisation infrastructure. *Proceedings of 8th Annual IFIP TC-6 TC-11 Conference on Communications and Multimedia Security*, Windermere, U.K., 15–18 September 2004 (*IFIP International Federation for Information Processing Series*, vol. 175). Springer: Berlin, 2005; 251–261.
14. O'Gorman L. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE* 2003; **91**(12):2019–2040.
15. Burr WE *et al.* Electronic authentication guideline. *NIST Special Publication 800-63*, NIST, Gaithersburg, MD, September 2004.
16. Chadwick DW, Otenko A. RBAC policies in XML for X.509 based privilege management. *Proceedings of Security in the Information Society: Visions and Perspectives—IFIP TC11 17th International Conference on Information Security (SEC2002)*, Cairo, Egypt, 7–9 May 2002, Ghonaimy MA, El-Hadidi MT, Aslan HK (eds.). Kluwer: Dordrecht, 2003; 39–53.
17. Brostoff S, Sasse MA, Chadwick D, Cunningham J, Mbanaso U, Otenko S. 'R-What?' Development of a role-based access control (RBAC) policy-writing tool for e-scientists. *Software: Practice and Experience* 2005; **35**(9):835–856.
18. Alfieri R *et al.* VOMS: An authorization system for virtual organizations. *Proceedings of the 1st European Across Grids Conference*, Santiago de Compostela, Spain, 13–14 February 2003. Springer: Berlin, 2003. Available at: http://grid-auth.infn.it/docs/VOMS-Santiago.pdf.
19. Ortiz E. An introduction to Java Card technology—Part 1, 2003. http://developers.sun.com/techtopics/mobility/javacard/articles/ [August 2004].
20. International Standard ISO. Smart Card Standard Part 4: Interindustry Commands for Interchange, ISO 7816-4, 2004. Available at: http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4.aspx [August 2004].
21. Sun Microsystems Inc. Java Cryptography Extension (JCE) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4. http://java.sun.com/j2se/1.4.2/docs/.
22. Schlumberger. Cyberflex Access Software Development Kit 4.5, Guide to Schlumberger smart card middleware. http://www.cyberflex.com/Support/Documentation/MiddlewareGuid4-5.pdf [August 2004].
23. OCF. http://www.opencard.org [August 2004].
24. Sun Microsystems Inc. The Java Servlet Specification Version 2.4. http://java.sun.com/products/servlet/.
25. Sun Microsystems Inc. The Java Server Pages Specification Version 2.0. http://java.sun.com/products/jsp/.
26. Microsoft 'cryptography'. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/seccrypto/security/cryptography_portal.asp [20 July 2004].
27. RSA Laboratories. PKCS #10 v1.7: Certification Request Syntax Standard, 2000. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-10/pkcs-10v1_7.pdf [August 2004].

DOI: 10.1002/cpe