FAME-PERMIS Project Output
WORKPACKAGE 1 – Deliverable D1

Report on the Design of
FAME Architecture and Components

Aleksandra Nenadić
Ning Zhang
Alan Rector
Carole Goble

School of Computer Science
University of Manchester

December 2005

School of Computer Science • University of Manchester • Manchester M13 9PL
Table of Contents

1. Introduction ........................................................................................................... 3
2. Shibboleth .............................................................................................................. 4
   2.1. Shibboleth Architecture ............................................................................... 5
   2.2. Interactions among Shibboleth Components ............................................... 6
3. FAME, PERMIS, and Shibboleth .......................................................................... 8
4. FAME .................................................................................................................. 8
   4.1. FAME Architecture .................................................................................... 9
   4.2. Deriving Loa ............................................................................................... 11
   4.3. FAME Logic ................................................................................................. 12
       4.3.1. FAME Tokens .................................................................................... 12
       4.3.2. FAME Authentication Scenarios ....................................................... 13
       4.3.3. FAME Component Logic and Flow ..................................................... 14
       4.3.4. FAME Keys ....................................................................................... 17
       4.3.5. FAME Token Structure .................................................................... 17
       4.3.6. FAME Token Verification ................................................................. 19
       4.3.7. FAME Timeouts ............................................................................... 20
5. FAME Configuration .............................................................................................. 21
   5.1. Apache Configuration .................................................................................. 21
   5.2. Database Configuration ............................................................................... 25
   5.3. Authentication Server(s) Configuration ...................................................... 26
6. FAME Installation ................................................................................................. 27
   6.1. Prerequisites ............................................................................................... 27
   6.2. FAME Installation Kit ................................................................................. 27
   6.3. Installation Steps ......................................................................................... 28
       6.3.1. Setting up the FAME Apache Module .............................................. 28
       6.3.2. Setting up the FAME Database ......................................................... 29
       6.3.3. Setting up the AS ............................................................................. 30
FAME: Architecture and Components  
(Version 1.0)

1 Introduction

This document describes the design and implementation of FAME (Flexible Access Middleware Extension) architecture and components. The work is part of the FAME-PERMIS project aimed at providing a multi-level user identification/authentication service in the Shibboleth\(^1\) infrastructure that has been endorsed by the JISC (Joint Information Systems Committee) as the next generation authentication and authorisation infrastructure for the UK Higher Education community. The service derives authentication assurance levels based upon the strength of the authentication token and protocol used by a user and feeds it to the PERMIS\(^2\) authorisation decision engine for fine-grained access control.

The FAME component of the project is designed to protect Web-based applications (including the Shibboleth) with multi-level authentication services. It can be easily integrated with any authentication services with a Web-based front-end (e.g. Kerberos, NIS, and authentication systems that use LDAP or Mysql, etc.), and supports the use of various authentication tokens including username/password pairs, soft certificate tokens, hard certificate tokens (stored in a smart-card) and IP addresses. Based upon the token used in an authentication instance, FAME derives a level of (authentication) assurance (LoA) and passes it to an authorisation decision engine at a target site over the Shibboleth infrastructure. The decision engine, in this case PERMIS, can then achieve a more fine-grained access control using attributes including LoA. In addition, FAME has built-in support for single-sign-on (SSO), a facility which enables a user to authenticate once and gain access to resources provided by multiple sites without re-authentication.

Shibboleth technology define open-source protocols and key entities (including Identity Providers (IdPs), Service Providers (SPs), and Where Are You From (WAYF) service) to support locally authenticated users to access remote resources provided by various institutions (SPs) in a federation. Shibboleth itself does not provide authentication and authorisation services. It leaves the task and the means of identifying a user (i.e. user authentication) to the user’s home institution (IdP). Instead, it defines a set of protocols for the secure assertion of the user’s identification information and attributes between the IdP and a target institution (SP). Shibboleth assumes that the IdPs and SPs in a federation have pre-defined trust relationships, and a SP trusts an IdP to authenticate its users properly. By using the user’s attributes from the assertion made by the IdP, the SP makes an authorisation decision. In addition, Shibboleth allows a user (or his/her home institution) to choose what attributes can be released to a specific SP thus achieving user privacy.

Different authentication methods, tokens, and protocols provide different levels of authentication assurance\(^3\). Different data items (or resources) may have varying levels

\(^2\) The PERMIS project, [http://www.permis.org](http://www.permis.org).
of sensitivity (e.g. library catalogue vs. highly confidential patients’ medical records). To provide a fine-grained access control to resources, there is a need to link access privileges to the authentication assurance level of the method/token used to identify a user. For instance, IP address-based authentication and authorisation services would grant the access privilege to anybody who has access to a machine with a correct IP address. Authentication via username/password establishes the identity of a user through proving the knowledge of an authorised username/password pair. A smart-card based authentication method authenticates a user provided that the user possesses a hard cryptographic token and also can demonstrate the knowledge of a secret (or a PIN) used to lock/unlock this token. Clearly, the IP-based authentication method provides the weakest, whereas the smart-card based method provides the strongest level of authentication assurance among the three methods. Access to data with different levels of sensitivity should be controlled with proper authentication methods/tokens. The FAME-PERMIS project is aimed at realising this vision.

In the Shibboleth infrastructure, FAME plays the role of an institution’s local authentication system (on the Identity Provider or IdP in Shibboleth terminology). It is responsible for offering users a range of authentication methods supported by the institution, and deriving an authentication assurance level (called Level of Assurance, or LoA, hereafter) based on the strength of the selected authentication method and authentication token. The derived LoA is then passed to the resource provider (an SP) via a Shibboleth protocol, along with other information about the user (in the form of attributes) that the institution/user is willing to release using the Shibboleth security assertion protocol. The SP subsequently feeds the LoA into an authorisation decision engine, namely PERMIS, to achieve fine-grained access control over its resources. As part of the project, PERMIS will be extended to include LoA in its decision making process, so that its authorisation decisions will now be made based on the following tuple: \{Subject, Target, Action, LoA\}, where Subject is the user accessing the resource, Target is what the user is accessing, and Action describes what action subject is allowed to perform on the target.

FAME and PERMIS components are both integrated into the Shibboleth infrastructure. The remaining part of this report will describe this integration, the detailed designs of the FAME-PERMIS architecture, the FAME component, and how the FAME component interoperates with PERMIS through the Shibboleth infrastructure. In detail, Section 2 describes the architecture and protocols of Shibboleth, and Section 3 explains how the two set of components, FAME and PERMIS, fit inside the Shibboleth framework. Finally, Section 4 is devoted to the detailed presentation of the FAME design, architecture and components.

2 Shibboleth

Shibboleth is an infrastructure for enabling inter-institutional sharing of Web resources that are subject to authentication and access control. It defines two types of architectural entities, origin sites (also referred to as identity providers or IdPs) and target sites (resource/service providers or SPs). The origin and target sites form a federation, and, in the Shibboleth infrastructure, different sites of the same federation are assumed to trust each other. In other words, a target trusts an origin to identify its users and provide correct attributes about the users. Based on the attributes that are
asserted by the origin, the target makes a decision as whether the user should be granted the access to the requested resource.

A user’s home institution (i.e. the user’s origin site) is responsible for identifying and authenticating the user, and providing attribute assertions about the user to a target site from which the user has requested resources. Shibboleth does not mandate how user authentication should be performed at the origin – the only requirement is that the origin has a Web-based authentication mechanism. Authorisation is performed at the target (e.g. by using an access control decision engine such as PERMIS) and is based on the attributes asserted by the origin. The access control mechanism matches the obtained attributes against policy rules associated with the requested resource in order to determine whether the requester is permitted to access it. Similarly, Shibboleth does not mandate which authorisation engine should be used by the target site. In other words, in the Shibboleth infrastructure, the authentication and authentication decisions are left to (and managed locally at) the origin and the target, respectively. The only requirement is that the origin and the target agree and understand the meaning of the attributes and implement the Shibboleth protocol for their request and release. In this way, Shibboleth detaches the management of users at cooperating institutions so that only IdPs are involved with verifying the identity of users belonging to their institution.

Shibboleth has an embedded mechanism to protect users’ identity privacy, i.e. to prevent the revelation of the user’s identity to the SP: the attributes released from an IdP to an SP need not contain information about the user’s identity, but rather a user’s pseudonym, i.e. an assertion that the user belongs to a certain group, such as student or member of staff. In this way, the user’s identity privacy is preserved and, furthermore, the user has the option of specifying exactly which of their attributes should be released to a specific SP.

2.1 Shibboleth Architecture

Shibboleth architectural components are illustrated in Figure 1. On the origin side (IdP), Shibboleth has four architectural components: the Attribute Authority (AA), the Handle Service (HS), a directory service (such as LDAP) containing user attributes and a Local Authentication System (LAS). The latter two components are not provided by Shibboleth. The HS is responsible for determining the identity of the user by interacting with the Local Authentication System (LAS) and creating an opaque handle (i.e. a reference number) for the user. The word ‘opaque’ means that no entity other that the AA and the HS should learn anything about the user from examining the handle alone. The handle will be used as an index number for the Shibboleth Attribute Requester (SHAR) from the target site to later fetch the user’s attributes from the AA. Shibboleth does not specify how the HS knows the identity of the user, but it is assumed that the user is logged to the LAS prior to being allowed to hit the HS. The AA manages all the users’ attributes at the origin site.

On the target side (SP), Shibboleth has three components: the Shibboleth Indexical Reference Establisher (SHIRE), the Shibboleth Attribute Requester (SHAR), and the Resource Manager (RM). SHIRE monitors the users’ requests for the resource on the target: if the user has not been authenticated yet by the IdP (as determined by the absence of the handle), the user is asked to go back to his IdP and do so. Once the user is authenticated and returns back to the SP with the handle obtained from his IdP,
SHAR uses the handle to ask for the user’s attributes from the IdP. RM makes access control decisions by interacting with some sort of an authorisation engine, once user’s attributes arrive from the IdP.

In addition, there is a WAYF (Where Are You From) service in the architecture. This service is used by an SP to find out the address of an IdP for a particular user. The WAYF service may determine the address of an IdP by allowing the user to select his/her origin institution from a list of institutions that have been registered with the federation. The WAYF service may be provided by a third party or directly by a SP.

Figure 1. Shibboleth architecture and components

### 2.2 Interactions among Shibboleth Components

A user initiates a request through his/her browser for a Shibboleth-protected resource provided by a SP (e.g. www.target-site/resource as shown in Figure 1). The SHIRE component at the SP intercepts the request to ask for a handle for the user (if there is none already contained in the request). For doing so, the SHIRE creates an Attribute Query Handle (AQH) message which the user’s browser carries to the HS located on the user’s IdP. To determine the address of the user’s HS, the WAYF service can be used. This service may allow the user to select its origin institution (containing the address of the HS) from a list of institutions that have been registered with the federation. Alternatively, if the SHIRE already knows the address of the user’s origin HS, the WAYF service can be skipped over. The AQH message also contains the SHIRE’s handle acceptance URL (i.e. the address to which the HS is to return the handle) and the originally requested target URL that should be re-requested upon successful authentication.

The HS at the origin is protected by the origin’s LAS (Local Authentication System), so before the HS is invoked by the browser carrying the AQH message, the user needs to authenticate him/herself to the LAS. Upon successful authentication, the HS creates
a handle (i.e. a reference number) for the user. The handle package (called AQH presentation) is passed back to the SHIRE via the HTTP POST method (by including it in a form that gets posted to the SHIRE’s handle acceptance URL). The handle package contains the handle for the user and the address of the AA that should be queried further for the user’s attributes. The actual handle is in the SAML format and is digitally signed (i.e. asserted) by the HS. Now when the user hits the SHIRE again, the SHIRE detects that the user browser contains the handle, and upon the verification of the authenticity of the handle (i.e. the verification of the HS’s signature in AQH presentation), the SHIRE passes it to the SHAR.

Upon the receipt of the handle, the SHAR sends an Attribute Query Message (AQM) containing the handle to the user origin’s AA to ask for the user’s attributes. The AA determines the user’s identity using the reference number contained in the handle, retrieves the user’s attributes related to the requested target from an attribute directory (e.g. LDAP), and constructs a reply, called an Attribute Response Message (ARM) containing these attributes. Both AQM and ARM are SAML messages. Based on the user’s attributes in the assertions, the decision is then made by the RM as whether to return the requested resource or an (HTTP) error to the user.

Figure 2 (originally from the document “Shibboleth Architecture: Protocol and Profiles”) depicts the interactions between the origin (identity provider) and the target (resource/service provider). The dashed lines and boxes represent optional interactions.

---

4 SAML (Security Assertions Markup Language) is an emerging OASIS standard used by Shibboleth for its attribute assertions (see http://www.oasis-open.org).
3 FAME, PERMIS and Shibboleth

This Section explains how FAME and PERMIS components fit into the Shibboleth infrastructure (see Figure 3 for illustrations).

PERMIS is integrated within Shibboleth infrastructure to provide the target sites with a policy-based access control decision engine to enable controlled use of their resources. The attributes obtained by the target’s SHAR from the origin’s AA are fed into PERMIS to make access-control decisions. One of the attributes passed from the origin to the target is LoA, which is derived by the FAME component upon the authentication method/token used by the user in the identification process.

FAME, on the other hand, is a part of the Shibboleth origin, and plays the role of the Local Authentication System (LAS) as shown in Figure 3. It is designed to offer users a variety of authentication methods supported by the origin, calculate the LoA based upon the authentication method/token used, and include this LoA together with the user’s other attributes ready to be fetched by Shibboleth targets.

![Figure 3. FAME-Shibboleth-PERMIS integration](image-url)

4 FAME

The purpose of the FAME component is to guard access to the Shibboleth’s HS that is run by an origin. As Shibboleth does not provide any authentication per se, FAME must ensure that only authenticated users are passed to the HS. The functionalities of the FAME can be summarised as follows:

1. Facilitate controlled access to the Shibboleth’s HS;
2. Allow a user to choose among a list of authentication methods (i.e. Authentication Servers) supported/provided by the origin;
Invoke the appropriate Authentication Server (AS) based upon the user’s choice;
(4) Calculate the LoA based on the cryptographic strength of the selected authentication method;
(5) Include the current user’s LoA among other user’s attributes to be passed back to the Shibboleth target;
(6) Provide the Single Sign-On (SSO) functionality. Instead of invoking an authentication process each time when the user requests the access to the Shibboleth’s HS, FAME only authenticates the user for the first time. After successfully authenticating the user, FAME issues him/her with a “SSO token” for subsequent accesses. The token can be reused every time when an access request is made by the user to the HS, until the token expires (or the user’s current SSO session ends). In other words, re-authentication of the user (provided that the user does not wish to do so) is not necessary within the validity period of a SSO token.

4.1 FAME Architecture

The FAME system architecture consists of the following (as shown on Figure 4): a User-Agent (UA), two FAME internal components and Shibboleth’s HS running inside an Apache Web Server, and a set of Authentication Services/servers (ASs) run by the origin. The User-Agent (UA) is a Web browser that supports the use of cookies. The two FAME components are:

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

The two FAME components jointly provide the SSO facility to the users. The Authentication Servers are Web-based authentication services external to the FAME components. These include any existing authentication services that can be invoked via a Web-based interface. Examples of such services are Kerberos, NIS, authentication via LDAP directory and MySQL database, SSL authentication with PKI soft and hard tokens. FAME does not re-implement all these existing authentication services, but rather integrates itself with them and invokes them as necessary. Furthermore, FAME is designed in such a way that it will support the use of any Web-based authentication system (including existing standard or custom built, as well as emerging ones).

Figure 4. FAME architecture
FAME SSO Checker (F-SSO)

The FAME SSO Checker (F-SSO) plays the role of a gate-keeper by controlling the access to the Shibboleth’s HS. If a user has not been authenticated yet in the current session (as determined by the absence of an sso cookie in the user’s Web browser) the user will be redirected to the FAME Login Server (F-LS) and forced to go through an authentication process. Otherwise, if the user has been previously authenticated in this session and issued with the sso cookie by the F-LS, the access to the HS will be granted by the F-SSO and the current Shibboleth session continues without re-authentication.

FAME Login Server (F-LS)

The role of the FAME Login Server (F-LS) is to receive users directed from the F-SSO and redirect them to an Authentication Server (AS) supported by the site. If a site has two or more ASs supported (determined by the origin’s authentication policy and needs), then the FAME interface window will allow a user to choose one from the list of the supported ASs, and the F-LS will redirect the user to the AS of his/her choice. Once the authentication is performed and successful, the user is returned back to the F-LS from the AS. The AS will also pass back the userID to the F-LS. Note that the userID varies with different authentication methods used. For instance, with username-password based authentication, the userID is the user’s username; with Kerberos, the userID is the principal’s name in the form of <Fully Qualified Domain Name>@<Realm Name>; for LDAP authentication, the userID is the user’s DN (Distinguished Name); for certificate based authentication, the userID a combination of C=<Country>, ST=<State>, L=<Locality>, O=<Organisation>, OU=<Organisational Unit>, CN=<Common Name> extracted from the user’s certificate; etc. Also note that one user can have multiple userID as he/she can be authenticated using various authentication methods on the site. However, at any one time, the user can only choose one authentication method.

Once a user has been successfully authenticated by, and re-directed from, the AS, the user is issued with an sso cookie by the F-LS. The cookie is scoped for the use by the F-SSO and contains the userID received from the AS and the LoA derived by F-LS based upon the authentication method/token used by the user. The SSO cookie also contains other information; detailed descriptions of the contents of the SSO cookie are given in Section 4.2.1. After the sso cookie is generated, the F-LS redirects the user back to his originally requested URL (i.e. the url of the Shibboleth’s HS). This time, when the user tries to access the HS, the request will again be intercepted by the F-SSO. However, as now the user’s request contains the sso cookie created by the F-LS, the F-SSO will let the user through to the HS. In addition, the F-SSO will pass to the HS the userID (contained in the sso cookie) through the environment variable REMOTE_USER in order to supply the HS with the identity of the browser user. This information is further used by the HS to obtain a handle for the user. On the other hand, if the authentication with the AS is not successful, it is up to the AS to display the error message to the user and the user will not be redirected to the HS.

All the interactions between the UA and the F-SSO and between the UA and the F-LS are SSL-protected to prevent cookies from being disclosed to an eavesdropper that could use the cookies to impersonate the user and to gain unauthorised access to the
HS. All the interactions between the UA and the AS are SSL-protected to prevent disclosure of passwords or other sensitive credentials. In the following, we give a more in-depth coverage of the FAME logic and authentication scenarios.

4.2 Deriving LoA

According to NIST Draft Recommendation for Electronic Authentication (NIST Publication 800-63), there are four levels of authentication - Level 1 to Level 4. Level 1 provides the lowest level of assurance and Level 4 the highest. These levels are defined in terms of the likely consequences of an authentication error. As these errors become more serious, the required level of assurance increases. A short summary of the technical requirements for each of the four levels is provided below.

Level 1 – Successful authentication requires the claimant to prove that he/she controls the authentication token through a secure authentication protocol. Any of the authentication tokens of Level 2, 3, or 4 are allowed, as well as PINs. Plaintext tokens (e.g. passwords) are never transmitted across the network. However, this level does not require the use of cryptographic protocols that block off-line attacks. For example, password challenge-response protocols are allowed at this level, and an eavesdropper, having intercepted such a protocol exchange, can launch an off-line dictionary attack in order to discover the password. Therefore, there is no requirement at this level to use FIPS (Federal Information Processing Standards) approved cryptographic techniques.

Level 2 - Successful authentication requires the claimant to prove that he/she controls the authentication token through a secure authentication protocol. Any of the authentication tokens of Level 3, or 4 can be used, as well as passwords. The use of cryptographic protocols that can prevent off-line, replay and on-line guessing attacks is required. FIPS approved cryptographic techniques are required.

Level 3 - Successful authentication requires the claimant to prove that he/she controls the authentication token (i.e. to prove a possession of a key or a password) through a secure authentication protocol. Three kinds of tokens may be used: cryptographic (soft and hard) tokens, one-time password device tokens and password tokens used in zero-knowledge password protocols. The use of cryptographic protocols that can prevent off-line, replay, on-line guessing, verifier impersonation and man-in-the-middle attacks is required. FIPS approved cryptographic techniques are required. All sensitive data transferred is cryptographically authenticated and, optionally, encrypted under keys derived from the authentication process.

Level 4 - Successful authentication requires the claimant to prove that he/she controls the authentication token (i.e. to prove a possession of a key or a password) through a secure authentication protocol. This level is similar to Level 3, except that only hard cryptographic tokens are allowed. These tokens are hardware (physical) devices that cannot be easily copied and which must be unlocked with a password or a piece of biometric, and, thus, provide two-factor authentication. Either public or symmetric key technology may be used and FIPS approved cryptographic techniques are required.

Table 1 shows relations between authentication tokens and the corresponding LoAs.
Table 1. Authentication tokens vs. LoAs

<table>
<thead>
<tr>
<th></th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard token</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Soft token</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Zero-knowledge password</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>One-time password</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Strong password</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the FAME system, it is the responsibility of the Authentication Server (AS) to provide the FAME administrator with the correct LoA, based on the cryptographic protocols and tokens used in the authentication process provided by the AS. This LoA is then hard-coded into FAME by the FAME administrator, i.e. inserted into the database that holds configuration information of all ASs that FAME component interacts with.

### 4.3 FAME Logic

The FAME authentication service makes use of several tokens and techniques to achieve SSO. The ideas are based on the SSO solutions described in the “Eagle Book”\(^6\), PubCookie\(^7\), WebAuth\(^8\) and Apache-AuthTicket\(^9\).

#### 4.3.1 FAME Tokens

To store and exchange information between different system components, to allow the components to mutually authenticate each other and to achieve SSO functionality, the FAME solution makes use of several tokens, as outlined in Table 2. These tokens are transferred between components either as cookies or URL parameters. Cookies are only used for passing information between two FAME internal components (namely, F-SSO and F-LS), while URL parameters are used to pass information between FAME internal component F-LS and external ASs. A more detailed explanation of the contents of the tokens is given in Section 4.2.4.

---

\(^6\) The “Apache Modules with Perl and C” book by Lincoln Stein and Doug MacEachern.


\(^9\) Apache-AuthTicket, [http://www.annocpan.org/~MSCHOUT/Apache-AuthTicket-0.40/lib/Apache/AuthTicket.pm](http://www.annocpan.org/~MSCHOUT/Apache-AuthTicket-0.40/lib/Apache/AuthTicket.pm)
Table 2. FAME Tokens

<table>
<thead>
<tr>
<th>Token name</th>
<th>Passed From → To</th>
<th>Token protection</th>
<th>Token description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request-url cookie</td>
<td>F-SSO → F-LS</td>
<td>The token is passed only over an SSL-protected channel.</td>
<td>Generated by the F-SSO and scoped for the use by the F-LS. It contains the address of the originally requested url (of the Shibboleth’s HS), which was intercepted by the F-SSO.</td>
</tr>
<tr>
<td>sso cookie</td>
<td>F-LS → F-SSO</td>
<td>The token is passed only over an SSL-protected channel. It also contains a “cryptographic signature” created as a keyed hash of the token data with a secret key known only to the F-SSO and F-LS.</td>
<td>Generated by the F-LS and scoped for the use by the F-SSO after the user has successfully authenticated in order to implement SSO for subsequent resource accesses by the user via FAME. This token is not used by any external component.</td>
</tr>
<tr>
<td>auth-request token</td>
<td>F-LS → AS</td>
<td>Encrypted with the secret key shared between the F-LS and the AS and is only passed via an SSL-protected channel.</td>
<td>Generated by the F-LS and sent to the AS when requesting user authentication with the AS. It contains a random challenge $RC$ that is used to authenticate the AS to the F-LS when response from the AS is received in the form of auth-reply token.</td>
</tr>
<tr>
<td>auth-reply token</td>
<td>AS → F-LS</td>
<td>Encrypted with the secret key shared between the F-LS and the AS, and only passed via an SSL-protected channel.</td>
<td>Generated by the AS once the user has been successfully authenticated and before the user is directed back to the F-LS. It contains a random challenge (that equals to the random challenge received from the auth-request token incremented by 1, i.e. $RC + 1$) and the userID of the authenticated user.</td>
</tr>
<tr>
<td>auth-control cookie</td>
<td>F-LS → F-LS</td>
<td>The token is passed only over an SSL-protected channel. It also contains a “cryptographic signature” created as a keyed hash of the token data with a secret key known only to the F-SSO and F-LS.</td>
<td>Generated by the F-LS and scoped for the use of the F-LS only. It contains the same random number sent to the AS in auth-request token and is used to verify the authenticity of the AS’s auth-reply token.</td>
</tr>
</tbody>
</table>

4.3.2 FAME Authentication Scenarios

In the following, we use use-case scenarios to illustrate the working of the FAME system. Steps of authentication procedures are presented together with illustrating diagrams where redirection between the components is represented using dashed lines.

**Scenario 1: Access to the Shibboleth’s HS without sso cookie (initial sign-on)**

A user initiates a new session to access the FAME-protected Shibboleth’s HS. In this case, no sso cookie has been generated by the F-LS yet. In detail,

1. A user makes a request for the url of the Shibboleth HS (running inside an Apache Server at the origin) that uses FAME for authentication.
2. The F-SSO intercepts the request and verifies that it is not associated with a current valid authenticated session for the requested resource. This is achieved by checking whether the request contains a valid sso cookie that has been previously issued by the F-LS. Consequently, the F-SSO generates the request-url cookie that contains the address of the originally requested
resource (i.e. that of the HS) and redirects the user to the F-LS to perform the authentication.

3. The F-LS determines a list of the available Authentication Servers (ASs) each with an associated LoA, and sends a form page with the list back to the user.

4. The user is presented with the page containing the ASs choices and chooses his/her preferred method of authentication.

5. The user sends his choice back to the F-LS by submitting the form.

6. The F-LS redirects the user to the selected AS after generating the auth-control cookie (containing the random challenge RC) and auth-request token (also containing the random challenge that is passed to the AS, i.e. RC).

7. The user is presented with the login page by the AS where they enter their credentials, if authentication methods such as username/password-based ones are chosen. Other authentication methods, such as SSL client certificate authentication, do not require the user to enter any credentials. Authentication in these cases is performed without further interactions between the user and the AS, and the user’s authentication credentials are sent to the AS automatically by the user’s browser.

8. The user’s credentials are sent by the browser to the AS. If the authentication is not successful, the user is prompted to re-enter their credentials until the authentication is completed successfully.

9. Otherwise, if the authentication is successful, the AS generates an auth-reply token containing the userID of the authenticated user and the random challenge incremented by one, i.e. RC+1, and re-directs the user back to the F-LS.

10. The F-LS verifies the auth-reply token by confirming that the random number contained inside it (RC+1) is equal the random number from the auth-control cookie incremented by 1. This verification confirms the authenticity of the AS to the F-LS as the tokens exchanged between them (i.e. auth-request and auth-
reply tokens) are encrypted with the secret key shared by these two components. If the verification is positive, the F-LS generates a **sso** cookie for the use by the F-SSO. The F-LS then redirects the user to the originally requested url (which was received through the **request-url** cookie).

11. The redirect will cause the browser to re-request the original resource. This time, when the request is intercepted by the F-SSO, it contains the **sso** cookie issued by the F-LS.

12. The F-SSO verifies the **sso** cookie, and if the verification is successful the F-SSO lets the user through to the originally requested resource, i.e. the HS.

**Scenario 2: Access to the Shibboleth’s HS with sso cookie (Single Sign-On)**

In this scenario, a user requests access to the HS and the user has already got a valid **sso** cookie for this session (because he/she has previously visited the HS and has been successfully authenticated to the F-LS).

![Figure 6. Application access with an sso cookie](image)

1. A user makes a request to an F-AS for an URL tied to an application (i.e. the Shibboleth HS) that uses FAME for authentication.

2. The F-AS detects a valid **sso** cookie (which the user has obtained earlier in the session) and grants the user the access. If the **sso** cookie has expired or is invalid, the user will be redirected to the F-LS for re-authentication.

### 4.3.3 FAME Component Logic and Flow

**F-SSO Logic**

- Does the request have a valid **sso** cookie?
  - Yes: Satisfy the request and let the user through to the requested Shibboleth’s HS.
  - No: Redirect the user to the F-LS.

**F-LS Logic**

- Does the request have a valid **request-url** cookie?
  - Yes: Send a list of available ASs to the user. Has the user submitted his preferred AS?
    - Yes: (1) F-LS generates an **auth-control** cookie scoped for the F-LS itself that will keep a random RC. (2) F-LS generates an **auth-request** token scoped for the AS containing the same random challenge RC. In addition, it also contains the return address of the F-LS, to which the user will be re-directed upon successful authentication. (3) F-LS redirects the user to the
selected AS. (4) Has the user been returned from the AS with a correct **auth-reply** token?

- Yes: F-LS sets the sso cookie, empties request-url and auth-control cookies and redirects the user to the originally requested address.
- No: The AS will display the error message to the user.
  - No: The user cannot continue unless an AS is selected.
  - No: Generate an error page, as this was not a redirect from the F-SSO.

**AS Logic**

- Does the request have a valid **auth-request** token?
  - Yes: Authenticate the user. Has the user authenticated correctly?
    - Yes: Decrypt the auth-request token and retrieve the random challenge \( RC \) and the return address of the F-LS. Generate the encrypted auth-reply token containing the response to the random challenge and the authenticated user’s userID and redirect the user back to the F-LS.
    - No: Ask user to re-authenticate.
  - No: Generate an error page.

The following diagram depicts the flow between the FAME system components.

---

**Figure 7. FAME flow diagram**
4.3.4 FAME Keys

FAME system components use secret keys for the protection of tokens passed between them. The internal components, namely F-SSO and F-LS, share and use the same secret key, called FLS_KEY, to protect cookies passed between them (i.e. request-url, auth-control and sso cookies). They use the same key as they are essentially part of the same Apache module. This key is used to protect the authenticity and integrity of the cookies by generating a “cryptographic signature” that is a hash value over the secret key and the data in the cookie. That is, the cryptographic signature of a cookie is generated as follows:

\[ H = \text{md5}(\text{FLS}_\text{KEY}, \text{md5}(\text{FLS}_\text{KEY}, <\text{cookie}_\text{data}>)) \].

As cookies are only passed over an SSL-protected connection, this ensures that the cookies are also protected against eavesdropping and there is no need for double encryption of cookies’ data.

Each FLS_KEY also has a version number associated with it. This allows the site administrator to issue a new secret FLS_KEY periodically without invalidating the current valid cookies. For example, the site administrator might periodically insert a new secret key into the secret key database, and flush secret keys that are more than 2 days old. Since the cookie issued to the user contains the secret version, the authentication process will still accept the cookies as long as the corresponding secret key exists in the secret key database. Apache Server should be restarted after the key is updated, to allow the latest key to be used by the F-LS and F-SSO.

In addition, the F-LS component also shares a secret key, called FLS_AS_KEY, with each of the external Authentication Servers that it supports. This key is used to encrypt auth-request and auth-reply tokens passed between the two components. The tokens are passed as URL parameters and are visible in Web browser’s address bar in order to achieve mutual authentication between the F-LS and the AS. To prevent eavesdropping their contents, the tokens are encrypted. It is the responsibility of the F-LS and AS administrator(s) to make sure that the correct FLS_AS_KEY is agreed and is in place.

4.3.5 FAME Token Structure

<table>
<thead>
<tr>
<th>Request-url cookie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role: The role of this cookie is to pass the originally requested address of the Shibboleth’s HS from the F-SSO that guards the access to the HS to the F-LS that is responsible for authentication. Upon successful authentication the F-LS redirects the user to this address.</td>
</tr>
<tr>
<td><strong>Field</strong></td>
</tr>
<tr>
<td>creation_time</td>
</tr>
<tr>
<td>request_url</td>
</tr>
<tr>
<td>remote_user_ip</td>
</tr>
<tr>
<td>key_version</td>
</tr>
</tbody>
</table>
| hash | Hash of the cookie’s concatenated fields above using secret key, FLS_KEY: \[
hash = \text{md5}(\text{FLS}_\text{KEY}, \text{md5}(\text{FLS}_\text{KEY}, \text{creation}_\text{time}, \text{request}_\text{url}, \\
\text{remote}_\text{user}_\text{ip}, \text{key}_\text{version})).
\]
**Sso cookie**

Role: This cookie helps with providing the SSO facility. It is created by the F-LS upon user’s successful authentication and return from the AS. It also contains the remote user’s ID that is later passed to the Shibboleth’s HS, as well as his/her LoA that will be used later as one of the user’s attributes passed to PERMIS via Shibboleth. The cookie also contains the user’s IP address, in order to make it more difficult for an attacker to forge the cookie. A potential attacker would have to guess both the secret key and spoof the user’s IP address. The cookie is created by the F-LS and used by the F-SSO to grant the access to the Shibboleth’s HS.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creation_time</td>
<td>Timestamp of cookie creation.</td>
</tr>
<tr>
<td>remote_user</td>
<td>UserID used when user authenticated with AS.</td>
</tr>
<tr>
<td>remote_user_loa</td>
<td>LoA of the AS selected by the user for authentication.</td>
</tr>
<tr>
<td>remote_user_ip</td>
<td>User’s IP address.</td>
</tr>
<tr>
<td>key_version</td>
<td>Version of the key used to create the hash of this cookie.</td>
</tr>
<tr>
<td>hash</td>
<td>Hash of the cookie’s concatenated fields above using secret key, FLS_KEY:</td>
</tr>
<tr>
<td></td>
<td>hash = md5(FLS_KEY, md5(FLS_KEY, creation_time, remote_user, remote_user_loa, remote_user_ip, key_version)).</td>
</tr>
</tbody>
</table>

**Auth-request token**

Role: This token, passed from the F-LS to the AS as a url parameter, carries the random number challenge for the AS and the address of the F-LS where the user should be returned upon successful authentication.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random_no</td>
<td>Random number challenge (used for authenticating the AS).</td>
</tr>
<tr>
<td>fls_address</td>
<td>Address of the F-LS to return the user upon authentication.</td>
</tr>
</tbody>
</table>

**Auth-control cookie**

Role: This is a control cookie that holds random number challenge that is also passed from the F-LS to the AS via the auth-request token. The random number is later compared to the random number received from the AS in the auth-reply token. The cookie also holds the address of the AS, the corresponding LoA, and remote user’s IP address in order to make the cookie more difficult to forge. A potential attacker would have to both guess the secret key and spoof the user’s IP address. The cookie is created and used only by the F-LS, and emptied when the user is returned to his originally requested url.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creation_time</td>
<td>Timestamp of cookie’s creation.</td>
</tr>
<tr>
<td>random_no</td>
<td>Random number challenge RC (used for authenticating the AS to the F-LS). This random number has the same value as the one contained in the auth-request token passed to the AS.</td>
</tr>
<tr>
<td>as_url</td>
<td>Address of the AS selected by the user for authentication.</td>
</tr>
<tr>
<td>remote_user_loa</td>
<td>LoA of the AS selected by the user for authentication.</td>
</tr>
<tr>
<td>remote_user_ip</td>
<td>User’s IP address.</td>
</tr>
</tbody>
</table>

---

10 UserID can be username with username-password based authentication, DN (Distinguished name) with certificate-based authentication, Kerberos username, etc. depending on the authentication method.
<table>
<thead>
<tr>
<th>key_version</th>
<th>Version of the key used to create the hash of this cookie.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>Hash of the cookie’s concatenated fields above using secret key, FLS_KEY: hash = md5(FLS_KEY, md5(FLS_KEY, creation_time, random_no, as_url, remote_user_loa, remote_user_ip, key_version)).</td>
</tr>
</tbody>
</table>

**Auth-reply token**

Role: This token, passed from the AS to the F-LS as a url parameter, carries the response to the random challenge received in the auth-request token and the userID of the authenticated user.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random_no</td>
<td>Random number from the auth-request token incremented by 1 (used for authenticating the AS to the F-LS).</td>
</tr>
<tr>
<td>remote_user</td>
<td>UserID extracted from the user’s credentials (e.g. username) when the user is being authenticated with the AS.</td>
</tr>
</tbody>
</table>

### 4.3.6 FAME Token Verification

**Verifying an request-url cookie (performed by the F-LS)**

- The cookie has correct format (i.e. no missing fields);
- The cookie is within its expiration time (set to 1 minute by default):
  
  \[
  \text{request-url cookie.creation_time} < \text{current_time} + 1.
  \]
- The hash contained in the cookie is verified positively with the secret key FLS_KEY shared between the FAME internal components F-SSO and F-LS, to confirm the authenticity and integrity of the cookie:

  \[
  \text{request-url cookie.hash} = \text{md5}(\text{FLS_KEY}, \text{md5}(\text{FLS_KEY}, \text{request-url cookie.creation_time}, \text{request-url cookie.request_url}, \text{connection.remote_ip}, \text{request-url cookie.key_version})).
  \]

**Verifying an sso cookie (performed by the F-SSO)**

- The cookie has correct format (i.e. no missing fields);
- The cookie is within the SSO Timeout (set to 8 hours by default):

  \[
  \text{sso cookie.creation_time} < \text{current_time} + 480.
  \]
- The hash contained in the cookie is verified positively with the secret key FLS_KEY shared between the FAME internal components F-SSO and F-LS, to confirm the authenticity and integrity of the cookie:

  \[
  \text{sso cookie.hash} = \text{md5}(\text{FLS_KEY}, \text{md5}(\text{FLS_KEY}, \text{sso cookie.creation_time}, \text{sso cookie.remote_user}, \text{sso cookie.remote_user_loa}, \text{connection.remote_ip}, \text{sso cookie.key_version})).
  \]
Verifying an auth-request token (performed by the AS)

- The AS can decrypt the token with the secret key shared between the AS and the F-LS and the obtained token has correct format (i.e. no missing fields). This token is sent from the F-LS to the AS and we do not require any additional verification here as the AS should perform authentication whenever it receives an authentication request from the F-LS.

Verifying an auth-control cookie (performed by the F-LS)

- The cookie has correct format (i.e. no missing fields);
- The cookie is received within the Authentication Timeout (set to 1 minute by default);

\[
\text{auth-control cookie.creation_time} < \text{current_time} + 1.
\]

- The hash contained in the cookie is verified positively with the secret key FLS_KEY used for its creation by the F-LS, to confirm the authenticity and integrity of the cookie:

\[
\text{auth-control cookie.hash} = \text{md5}(\text{FLS_KEY}, \text{md5}(\text{FLS_KEY, auth-control cookie.creation_time, auth-control cookie.random_no, auth-control cookie.as_url, auth-control cookie.remote_user_loa, connection.remote_ip, auth-control cookie.key_version)}).
\]

Verifying an auth-reply token (performed by the F-LS)

- The AS can decrypt the token with the secret key shared between the AS and the F-LS and the obtained token has correct format (i.e. no missing fields). The token is received within the specified authentication timeout, i.e. within the time that is left to the user to perform the authentication and return to the F-LS from the AS (which is set to 1 minute by default);

\[
\text{auth-reply cookie.creation_time} < \text{current_time} + 1.
\]

- The random number contained in the token is equal to the random number from the auth-control cookie incremented by 1.

\[
\text{auth-reply cookie.random_no} = \text{auth-control cookie.random_no} + 1.
\]

4.3.7 FAME Timeouts

- The Authentication Timeout indicates the time period during which the auth-control cookie and auth-request token are created by the F-LS, the user is redirected to the AS for authentication, and the user is returned back from the AS to the F-LS after the authentication. In other words, it is the time duration given to the user for her/him to successfully complete the authentication. It is calculated as the difference of the auth-control cookie’s creation_time and the time when the auth-reply token is received. The default setting is 1 minute, but this can be changed via FameAuthTimeout directive (see Section 5.1 for details).

- The SSO Timeout specifies the time interval upon the expiration of which the user will be prompted for re-authentication. It is the time interval within which the SSO cookie is considered as non-expired. It is computed using the
creation_time field of the sso cookie. The default setting is 8 hours, but this can be changed via FameSSOTimeout directive (see Section 5.1 for details).

5 FAME Configuration

The FAME system internal components are implemented as an Apache mod_perl module called Fame.pm. The module is created under the MyApache2:: namespace, thus its full name is actually MyApache2::Fame.pm. The system also keeps configuration data and secret keys in a database, and makes use of locally supported (by the IdP) authentication systems. Thus, there are three things that have to be done in order to configure the FAME system:

1. Configure mod_perl Apache2 Server,
2. Create the required database for storing secret keys and information about ASs and their LoAs, and
3. Set up the Authentication Server(s) to work with FAME.

5.1 Apache Configuration

The module has been developed for and tested with Apache2. There are two blocks that have to be entered into Apache2’s configuration file, httpd.conf. The first block is to specify Apache’s access control to the Shibboleth HS, which is presumably already configured to run with the Apache Server, and plays the role of the F-SSO. The following gives an exemplar setting of this block:

```xml
<Location /shibboleth/HS>
  AuthType MyApache2::Fame
  AuthName Fame
  PerlAuthenHandler Apache2::Fame::sso_checker
  require valid-user
</Location>
```

The second block specifies the location and settings for the F-LS. It should look similar to this:

```xml
<Location /fls>
  PerlResponseHandler Apache2::Fame::login_server
</Location>
```

A synopsis of the FAME module’s configuration that should be entered in the Apache’s httpd.conf is given as follows.

```plaintext
# in httpd.conf
# Load Fame module
PerlModule MyApache2::Fame
# Fame module configuration items
PerlSetVar FameLoginServer /fls
PerlSetVar FameTopPage /var/www/localhost/htdocs/fame/top.html
```
PerlSetVar FameBottomPage /var/www/localhost/htdocs/fame/bottom.html
PerlSetVar FameErrorPage /var/www/localhgost/htdocs/fame/error.html
PerlSetVar FameDB dbi:mysql:database=fls;host=localhost;port=3306
PerlSetVar FameDBuser flsuser
PerlSetVar FameDBPassword flspassword
PerlSetVar FameSecretsTable secrets:secret_key:date_time
PerlSetVar FameASTable authservers:url:auth_type:loa:secret_key
PerlSetVar FameAuthTimeout 1
PerlSetVar FameSSOTimeout 480

# Shibboleth’s HS protected by the F-SSO
<Location /shibboleth/HS>
  PerlAuthenHandler Apache2::Famie::sso_checker
  AuthType MyApache2::Fame
  AuthName Fame
  require valid-user
</Location>

# Fame Login Server
<Location /fls>
  PerlResponseHandler Apache2::Fame::login_server
</Location>

Valid configuration items are:

**FameLoginServer**

This directive is mandatory. It specifies the relative url of the Fame Login Server (F-LS).

**FameTopPage**

This directive is optional. It is used to customise the look of the HTML page that shows a list of Authentication Servers available to the users when they hit the F-LS after being redirected to it by the F-SSO (see Figure 8). It goes in pair with the **FameBottomPage** directive and only will be used if both directives are specified. The page that lists the Authentication Servers is created dynamically and consists of three parts. The first part is static and is specified by **FameTopPage**. This is actually an absolute path to a file that contains the top part of the HTML page. The second part of the page is the list of the Authentication Servers that is contained in an HTML table generated dynamically by the F-LS (the available Authentication Servers and their addresses are read from the database). This HTML code is appended to the top part of the page (read from the **FameTopPage** file). Finally, the contents of the file specified by the **FameBottomPage** are appended after the table with ASs, to finalise the page generation.

If not specified, the F-LS will use FAME’s default outlook for this page.

Default:

/fame/top.html (note that this is relative to the Apache’s DocumentRoot).
Figure 8. The F-LS page snapshot

**FameBottomPage**

This directive specifies the absolute path to an HTML file containing the bottom of the page with a list of available Authentication Servers. It goes in pair with FameTopPage directive.

Default:

/`fame/bottom.html` (note that this is relative to the Apache’s DocumentRoot).

**FameErrorPage**

This directive specifies the absolute path to a custom error page to display to the user when SERVER_ERROR occurs (this is done via Apache’s custom response mechanism). If not specified, the FAME’s default error page will be used.

Default:

/`fame/error.html` (note that this is relative to the Apache’s DocumentRoot).

**FameDB**

This directive specifies the Perl DBI URL string to use when connecting to the database.
Format:
<perl_database_interface>:<database_type>:database=<database_name>;host
=_host_name_:port=<port_number>.

Default:
dbi:mysql:database=fls;host=localhost;port=3306.
If the host is ‘localhost’ and the port is ‘3306’, they can be omitted from the string.

FameDBUser
This directive specifies the username to use when connecting to the database.
Default: flsuser.

FameDBPassword
This directive specifies the password to use when connecting to the database.
Default: flspassword.

FameSecretsTable
This directive specifies the name of a table containing secret keys used by the F-LS and F-SSO, as well as the names of the two columns of this table. The columns are the secret key itself and its version. We usually use a timestamp when the key was inserted in the database as the version number of the key (for details on database configuration, see Section 5.2).

Format:
<table_name>:<secret_key_column_name>:<secret_key_version_column_name>.

Default:
secrets:secret_key:date_time.

TicketASTable
This directive specifies the name for the table containing information about Authentication Servers, as well as the names of the four columns of this table. Columns are the url of the Authentication Server, the authentication type the Authentication Server provides, the LoA of the Authentication Server and the secret key shared between the AS and F-LS.

Format:
<table_name>:<url_column_name>:<auth_type_column_name>:<loa-column_name>:<secret_key_column_name>.

Example:

FameAuthTimeout
This directive specifies the number of minutes that the user is left to successfully complete authentication. It is computed as a difference between the creation time of an auth-control cookie and the receipt of the auth-reply token. If this timeout is expired, the user will be forced to log in (re-authenticate) again.

Default: 1 minute.

**FameSSOTimeout**

This directive specifies the number of minutes before the sso cookie is considered as invalid. If this timeout is expired, the user will be forced to re-authenticate.

Default: 480 minutes (8 hours).

### 5.2 Database Configuration

The FAME database is used by the F-SSO and F-LS components only and consists of two tables: the Secrets table and the Authentication Servers table. A database user with read-only access to this database is also required for the use by the two FAME components. An exemplar database called fls and a database user flsuser can be created as follows.

#### Example:

```sql
CREATE DATABASE fls;
GRANT SELECT on fls.*
TO flsuser IDENTIFIED BY flspassword;
```

The names used for the database, the Secrets and Authentication Servers tables, database user and password can be respectively set up for the FAME module in `httpd.conf` configuration file using directives **FameDB**, **FameSecretsTable**, **TicketASTable**, **FameDBUser** and **FameDBPassword**.

#### Secrets table

This table stores the secret keys used by the F-LS and the F-SSO for creation of cookies passed between them (namely, the sso cookie and the request-url cookie). Each key has a version number associated with it, which is implemented as the timestamp when the key was inserted into the database. The key version allows periodical updates of the secret key without invalidating current valid cookies created with previous keys. This table is configured by the **FameSecretsTable** directive and can be created as follows.

#### Example:

```sql
CREATE TABLE secrets (  
    secret_key text NOT NULL,  
    date_time datetime NOT NULL default '0000-00-00 00:00:00'  
);
```
Authentication Servers table

This table stores information about configured/supported Authentication Server(s). Each row includes the url at which the AS is running, the authentication type (exemplar settings are Username/password, Kerberos, Browser certificate, Smart-card certificate) and are shown on the main F-LS page as an option for the user to choose, the LoA provided by the AS, the secret key shared between the AS and the F-LS (which is used for encryption and decryption of the `auth-request` and `auth-reply` tokens), and an optional timestamp. The last (i.e. timestamp) column in the table is optional, as it is not used by the FAME module. This table is configured by the `FameASTable` directive and can be created as follows.

Example:

```sql
CREATE TABLE authservers (  
  url varchar(100) NOT NULL default '',  
  auth_type varchar(100) NOT NULL default '',  
  loa smallint(6) NOT NULL default '1',  
  secret_key text NOT NULL,  
  date_time datetime NOT NULL default '0000-00-00 00:00:00',  
  PRIMARY KEY (url)  
);
```

5.3 Authentication Server(s) Configuration

The FAME system attempts to use existing authentication systems and to integrate with them with minimum modifications. We distinguish two cases based on how the Authentication Server is implemented. (1) If an Authentication Server is a standard Apache module for protecting Web resources (such as Kerberos system using Apache module `mod_auth_kerb` or an LDAP-based authentication system using `mod_ldap`), then it is only required to install a script provided in the FAME installation kit (see Section 6) in order to make the FAME interoperate with the AS and no alterations to the AS are required. (2) If an AS is a custom-built system, then some modifications are required on the AS’s side, in order to accommodate requests passed by the F-LS component of the FAME system and return the necessary information back to the F-LS upon authentication. In case (2), it is necessary to understand how information is passed between the F-LS and the AS in order to make the necessary modifications to the AS. In case (1) when the AS is a standard Apache module, this functionality is implemented through a script provided within the FAME installation kit. Information on how to configure the script is given in Section 6.3.

The AS receives the authentication request from the F-LS in the form of:

```text
http://<address_of_the_AS>?AuthRequestToken=<encrypted_auth_request_token>
```

In other words, `auth-request` token containing two parameters is first encrypted by the F-LS with the symmetric key shared between the F-LS and AS before it is passed to the AS via URL. The AS has to decrypt the token and extract the two parameters contained in it: random challenge (`RC`) and the return address of the F-LS (to where
to redirect the user upon successful authentication), which are delimited by a comma (",".). The format of the auth-request token is:

\(<auth_request_token> = <random_challenge>,<address_of_the_FLS>\)

Upon successful authentication, the AS is required to redirect the user back to the F-LS and pass the auth-reply token via URL, which now looks like the following:

http://<address_of_the_FLS>/?AuthReplyToken=<encrypted_auth_reply_token>

Auth-reply token is again encrypted (now by the AS) with the symmetric key shared between the F-LS and AS. It contains a response to the random challenge from the auth-request token and the name of the authenticated user that has to be passed to Shibboleth (refer to Section 4.1 for information on what is considered as the user name), delimited by a colon (":".). The AS computes the response to the F-LS’s random challenge (i.e. RC) as RC + 1. The auth-reply token has the following format:

\(<auth_reply_token> = <random_challenge_response>:<user_name>\)

6  FAME Installation

The following installation instructions are given with reference to the Linux/Unix environment. The FAME system has been developed and tested under Gentoo Linux 2.6.11-r3. The FAME module (called Fame.pm) is an Apache module written in Perl that protects the access to the Shibboleth’s HS running within the Apache Web Server. Before you start to install FAME, you will have to have the following prerequisite components installed.

6.1 Prerequisites

(1) Apache Web Server (the FAME module was developed and tested with version 2.0 and it is suggested that you use the same version);
(2) Perl interpreter (version 5.8.6 was used for developing and testing the FAME module);
(3) mod_perl – an Apache module for providing a persistent Perl interpreter embedded in the Web Server for creating Apache Perl modules;
(4) Shibboleth version 1.2 origin and target should be installed and interoperating (you may also use the test Shibboleth target provided by Internet2\(^\text{11}\));
(5) Mysql database version 4.1 or above, or a similar database system (myqsl was used for developing and testing the FAME module).

6.2 FAME Installation Kit

The FAME installation kit comprises the following items:

(1) The source code for the FAME’s Apache module (written in Perl) – Fame.pm;
(2) A set of three HTML documents: top.html, bottom.html and error.html required by the Fame.pm to create user-friendly pages;
(3) The database set-up script – fame.sql;

\(^{11}\)http://shibboleth.internet2.edu/.
(4) A script for the AS (written in Perl) – AS.pm.
(5) A report on the Design of FAME Architecture and Components (i.e. this document), which contains the installation instructions.

### 6.3 Installation Steps

Installation should proceed according to the following steps.

#### 6.3.1 Setting up the FAME Apache Module

Firstly, create a directory where the Fame module will be located. For example, create a `perl` directory within the server root (on our system `/etc/apache2/perl`) where you will keep all your Perl modules (e.g. `/etc/apache2/perl`). As the module is created under the `MyApache2::` namespace, create the `MyApache2` subdirectory within the `perl` directory. The `MyApache2` subdirectory is where the `Fame.pm` module will be located.

```
$ mkdir /etc/apache2/perl
$ mkdir /etc/apache2/perl/MyApache2
$ cp Fame.pm /etc/apache2/perl/MyApache2
```

Next, you need to tell Apache where to look for this module. Apache’s `mod_perl` can be configured to invoke a start-up file with a set of Perl commands each time the server is launched and restarted. This is where we place a `use lib` statement that will tell Apache where to look for the `Fame.pm` module. Here is an example start-up file (the command to load the `Fame.pm` module is at the bottom, highlighted in green).

Place the file in the directory containing all the configuration files for the Apache modules.

**File: /etc/apache2/modules.d/apache2-mod_perl-startup.pl**

```perl
use lib qw(/home/httpd/perl);

# enable if the mod_perl 1.0 compatibility is needed
use Apache2::compat ();

use ModPerl::Util (); # for CORE::GLOBAL::exit

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::RequestUtil ();

use Apache2::ServerRec ();
use Apache2::ServerUtil ();
use Apache2::Connection ();
use Apache2::Log ();

use APR::Table ();
use ModPerl::Registry ();

use Apache2::Const -compile => ':common';
use APR::Const -compile => ':common';

# Enable Apache to find Fame module
```

- 28 -
use lib '/etc/apache2/perl';
1;

Finally, create the directory `fame` in your document root (on our system the document root is `/var/www/localhost/htdocs`). Copy the three HTML files that the `Fame.pm` module requires for creating pages to this directory.

```
mkdir /var/www/localhost/htdocs/fame
$ cp top.htm /var/www/localhost/htdocs/fame
$ cp Bottom.htm /var/www/localhost/htdocs/fame
$ cp error.htm /var/www/localhost/htdocs/fame
```

### 6.3.2 Setting up the FAME Database

Assuming that you have already installed the `mysql` database, the only thing you have to do now is to proceed to execute the `fame.sql` script to create the necessary database called `fls` and the default database user for the `Fame.pm` module. If you want to use a different username/password for the database user or different names for the database, table or column names (other than those used in the script), modify the script to reflect this. To execute the script, go to the directory where your `fame.sql` script is located and type in the following.

```
$ mysql -u root -p < fame.sql
Enter password:
```

To verify that `fame` database has been created properly, connect to mysql database:

```
$ mysql -u root -p
Enter password:
```

Type in the administrator’s password and you should receive the mysql prompt.

```
mysql>
```

Type in the following command to see the list of databases:

```
mysql> show databases;
```

You should be seeing something like the following:
### 6.3.3 Setting up the AS

In the case you are using a standard Apache authentication module (such as `mod_auth_kerb` for Kerberos authentication system), you should use the script provided (`AS.pm`) to get your AS interoperate with the F-LS component of FAME. `AS.pm` script is within the same namespace as `Fame.pm` module (i.e. Apache2:: namespace), thus copy `AS.pm` to the same directory where `Fame.pm` is located (in this installation guide it is `/etc/apache2/perl/MyApache2`).

Let us suppose you are running Kerberos authentication system using Apache module `mod_auth_kerb`. You should create a `<Location>` in your `httpd.conf` that is tied to our `AS.pm` script and protected by Kerberos. Such a section in your `httpd.conf` may look something like the following:

```plaintext
# # Location protected by Kerberos #
<Location /kerb-auth>
  SSLOptions +StrictRequire
  SSLRequireSSL
  SetHandler perl-script
  PerlResponseHandler MyApache2::AS
  AuthType KerberosV5
  AuthName "Kerberos Login"
  KrbAuthRealms CS.MAN.AC.UK
  Krb5Keytab /etc/apache2/apache2_kerb.keytab
  KrbMethodR5Passwd on
  KrbServiceName HTTP
  KrbVerifyKDC on
  require valid-user
</Location>
```

The above code defines a `<Location>` with your Web server, accessible from a Web browser as `http://<your_server>/kerb-auth` only by a user successfully authenticated by Kerberos. The `<Location>` (http://<your_server>/kerb-auth) actually plays the role of your AS. The `AS.pm` script serving this `<Location>` will receive requests from the F-LS (but only if the user has been authenticated previously), perform the necessary tasks (explained in the Section 5.3), and redirect the user back to the F-LS.

The `AS.pm` script can be reused for any authentication system other than Kerberos. The only thing that needs to be configured inside the script is the path to the file containing the Base64-encoded secret key shared between the F-LS and AS (look for variable `$KEY_FH`).