Xaraya
Request for Comments: 0014
Category: Informational

M. Canini
Xaraya Development Group
March 2002

Modular Authentication System

Status of this Memo

This memo provides information for the Xaraya community. It does not specify an Xaraya standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The contents of this RFC contain the literal content of the old plain text version of RFC-0014

When time is a less scarcer good, someone might convert the plain text into structured XML so we can benefit from it.

Table of Contents

1. Re-evaluation of this RFC

Since this RFC was first written, Xaraya has gone through many changes, and thanks to the initial implementations of authsystem and authldap, it's time to re-evaluate some of the strengths and weaknesses of the initial design. Here are a few issues we noted :

External authentication and external user data may be related in the sense that they both come from outside Xaraya, but there are many issues concerning retrieval, synchronisation or overriding of user data that do not apply to the "simple" action of authenticating a user at logon.

Usage scenarios for combining different authentication methods and/or distributed user data are not well defined, with the result that design and implementation are limited to certain basic scenarios.

At the moment (April 2003), only authenticate_user() is in active use, and this only for the login process. There is no link or verification during user registration.

User data from external authentication sources is only used at login, in order to create a local user if one doesn't exist. There are no further queries, updates or synchronisation of external user data besides that at the moment.

Current implementation in the auth* modules does not allow an easy centralised management of the different authentication methods, nor of any adequate handling of external user data.

Some usage scenarios for authentication (to be expanded and evaluated) :

local users, local authentication (obviously)

local users, external authentication : i.e. at first login, a local user is created based on external user information (e.g. LDAP or non-Xaraya DB). Local user registration can be disabled via privileges (?)

local users, multiple authentications (local or external) : potential conflict situations - who gets priority for (unique) username, or check against external auth at registration, or treat username+auth method as unique in the future, ...

external users, external authentication : not supported at the moment - see below

Some usage scenarios for user data (to be expanded and evaluated) :

local user data (obviously)

local user data, created once from external user data (e.g. LDAP or non-Xaraya DB)

local user data, resynchronised from external user data at every login (or some other event)

external user data, requested on demand - note : still a need for local core user fields

external user data gets resynchronised from local user data on user update

bi-directional synchronisation nightmare

multiple sources of user data (local or external), with any or all of the above + possible overlap/overriding of one by the other

To be continued...

Note : add link with DD

2. Introduction

Integration of older systems with new ones is a crucial point for success of systems, especially in the internet era. In main situations we can find heterogeneous services that can't cooperate because of their different authentication methods, and often implemented solutions are only wrappers or frontends to existing systems; in no way a definitive and universal solution. However in these years it's common practice to rely on well defined interfaces or portable libraries for getting a flexible and modular authentication system. To ensure Xaraya will be a high quality CMS we have to modularise authentication of users. This is definitely the purpose of MAS (Modular Authentication System).

3. Modifications to current authentication API

Currently the only official way to authorize user login is done by a set of functions in the Xaraya API. The default behavior of Xaraya will be to keep all registered users in the database. That is the current implementation. Modifications that will occure are:

Migration of current authentication code in an authentication module.

Reimplementation of user related API.

Reimplementation of authorization info gathering function. Notwithstanding the introduction of a decentralised users database, to avoid user IDs duplications and espacially user data limitations, Xaraya will keep an entry for every user in the main database.

4. Authentication Modules

The main idea is to use Xaraya modules API for implementing authentication modules. With the advent of Dynamic User Data, Xaraya will keep in the main database the part of user data that can't be handled by authentication module. A generic authentication module could override the user data present in database with other existing data (Think to LDAP authentication where many data already exists). The word override has to be intended as follow: premising that at current time the static users table coexists with the Dynamic User Data, if an authentication module can provide user data from other sources, the value stored in database will be ignored (overridden) whether it exists (has to in static table context) or not (optional user property in Dynamic User Data). The authentication module could implement an interface to create new users. The authentication module could implement an interface to delete users. The authentication module could override the permissions fetched from the database if it implements another interface.

4.1 Module Interfaces

A generic authentication module must implement at least the Authentication interface. Other interfaces extend the exported module API and add new feautures to this module. Incarnation of interfaces is done by defining and implementing one or more module API functions (If you can't understand what is a module API function you're strongly encouraged to read the xarMDG documentation before proceding). The name of one authentication module API function, for its first part, is submitted to module API naming rules. The rest of the name is an underscore plus the interface method name. In this section are described all authentication interfaces currently supported by Xaraya.

4.1.1 Interface Authentication

4.1.1.1 Definition

```
interface Authentication {
boolean authenticate_user($user_name, $user_password);
boolean has_capability($capability);
}
```

4.1.1.2 Incarnation

A generic authentication module ('authgen') MUST incarnate at least this interface. The incarnation for this interface is done by declaring and implementing the following methods:

function authgen_userapi_authenticate_user($args)

```
This function will receive the following parameters:
$args?'uname': user name
$args?'pass': user password
```

This function MUST return a valid system user id IF and ONLY IF the user credentials are considered valid (the user is authenticated sucessfully) or _XARAUTH_FAILED for invalid credentials or false for bad parameters. Here is important to explain what a valid system user id is. Xaraya keeps a reference to all system users notwithstanding they're authenticated by different authentication modules. The section 5. User IDs mapping explains how to return a valid system user id.

function authgen_userapi_has_capability($args) This function will receive the following parameters:

```
  $args?'capability': the required capability, it's an identifier for another
interface
possible values are: _XARAUTH_USER_DATA_HANDLER
_XARAUTH_PERMISSIONS_OVERRIDER
_XARAUTH_USER_CREATEABLE
_XARAUTH_USER_DELETEABLE
```

```
                _XARAUTH_USER_ENUMERABLE
```

This function MUST return true IF and ONLY IF the authentication module for wich it's written supports the required capability, false otherwise

4.1.2 Interface ?DynamicUserDataHandler_Authentication

4.1.2.1 Definition

```
                          interface ?DynamicUserDataHandler_Authentication extends
Authentication {
                boolean is_valid_variable($variable_name);
                array get_user_variables($user_id);
                string get_user_variable($user_id, $variable_name, $property_id,
$property_dtype);
                boolean set_user_variable($user_id, $variable_name,
$variable_value, $property_id, $property_dtype);
                }
```

4.1.2.2 Incarnation

The incarnation for this interface is done by declaring and implementing the following methods:

function authgen_userapi_is_valid_variable($args)

```
                This function will receive the following parameters:
                $args?'name': variable name
```

This function MUST return true IF and ONLY IF the authentication module can provide read and write operations for that variable, false otherwise.

function authgen_userapi_get_user_variables($args)

```
                This function will receive the following parameters:
                $args?'uid': user id
```

```
                This function MUST return all user variables
                handled by the authentication module as an associative
                array of pairs <variable_name, variable_value>, or
                false if it fails (remeber to set the errmsg session
                variable).
```

function authgen_userapi_get_user_variable($args)

```
                This function will receive the following parameters:
                $args?'uid': user id
                $args?'name': variable name
                $args?'prop_id': variable id in user_property table (probably you
can get rid of that)
                $args?'prop_dtype': variable type as defined by Dynamic User Data
types
```

This function MUST return the value of required user variable handled by the authentication module or false when this variable doesn't exist for that user or when it fails to fetch variable value (remeber to set the errmsg session variable).

function authgen_userapi_set_user_variable($args)

```
                This function will receive the following parameters:
                $args?'uid': user id
                $args?'name': variable name
                $args?'value': new value to set to
                $args?'prop_id': variable id in user_property table (probably you
can get rid of that)
                $args?'prop_dtype': variable type as defined by Dynamic User Data
types
```

This function MUST return true if the oparation was concluded with success, false when it fails in the set operation or for other errors (eg: Bad params, No such connection to backend, ...).

4.1.3 Interface ?PermissionsOverrider_Authentication

4.1.3.1 Definition

```
                interface ?PermissionsOverrider_Authentication extends
Authentication {
                array get_authorization_info($user_id);
                }
```

4.1.3.2 Incarnation

TODO

4.1.4 Interface ?UserEnumerable_Authentication

4.1.4.1 Definition

```
                interface ?UserEnumerable_Authentication extends Authentication {
                array get_all_users();
                array get_user($user_id);
                }
```

4.1.4.2 Incarnation

TODO

4.1.5 Interface ?UserCreateable_Authentication

4.1.5.1 Definition

```
                interface ?UserCreateable_Authentication extends Authentication {
                boolean create_user($user_data);
                }
```

4.1.5.2 Incarnation

TODO

4.1.6 Interface ?UserDeleteable_Authentication

4.1.6.1 Definition

```
                interface ?UserDeleteable_Authentication extends Authentication {
```

```
                boolean delete_user($user_id);
                }
```

4.1.6.2 Incarnation

TODO

4.2 Naming conventions

All authentication modules must respect the following naming conventions:

Module name MUST start with "Auth".

Module directory MUST be all in lowercase and MUST begin with "auth".

Module name SHOULD reflect as best as possible the authentication mechanism on wich it's based.

5. Authentication intersection

The new authentication system will permit to use more than one authentication module. All modified User API will communicate with the correct authentication module.

6. User IDs mapping

The UserIDs module will be created to handle UIDs inconsistencies. This is a utility module for authentication modules. Its API permits to easily find correspondencies between different system UIDs. Basically it will use the database to store all mappings.

7. Changelog

1.0 (April 14, 2003)

Re-evaluation of the RFC

0.9 (April 27, 2002)

Cleaned the architecture, moved most of complexity to core system.

Added Incarnation section to explain the obscure incarnation thing.

Added a naming convention.

0.12 (April 13, 2002)

Added has_capabilty method to Authentication interface.

Its purpose is to communicate to the core whether an auth module supports an interface.

Added ?DeleteableUserData_Authentication for a better fine-grained control on user variables.

Renamed ?PermissionOverrider_Authentication to ?PermissionsOverrider_Authentication.

0.1 (March 28, 2002)

Corrected ?ReadableUserData_Authentication and ?WriteableUserData_Authentication interfaces.

Better description of user data override mechanism (I hope).

Added Naming conventions section.

RFC number set to 14.

Removed some typos (Thanks to Gregor).

Rearranged document layout (Thanks to Gregor).

pre-0.1 (March 27, 2002)

Initial Version by Marco Canini <marco.canini@postnuke.com>