mit iCampus

Author:	
Last Modified:	2005-01-22T22:13:00Z
By:	Jud Harward

Table of Contents

Introduction

Authentication and Authorization

Authentication and authorization are complementary services that govern what operations a user can perform within a particular computing system. Authentication determines who the user *is*. An authenticated user is a user whose identity has been determined to the satisfaction of the system. Such a user is often referred to as a *principal*. In the rest of this document we shall use principal to refer to an authenticated identity and user to mean a valid user of an *iLab* system with an account identified by a **`userID`**.

Authorization determines what privileges a user possesses. In many systems these checks are performed at runtime as the user tries to access specific resources. That is the strategy we have adopted for *iLab*. Whenever the user attempts to execute a Service Broker method that requires a special permission, the Service Broker must first call the **`CheckAuthorization()`** method with a number of arguments including the **`userID`**, which represents the Service Broker's view of the authenticated user's identity. Obviously, authorization will only work properly if authentication has first correctly identified the user.

Native vs. External Authentication

An authentication API, such as the one described below, often has two tasks:

1. it must determine a user's identity, when the user accesses the system; and
2. it must allow system administrators to manage user identities, by creating new accounts, changing forgotten passwords, etc.

There is increasing interest today in providing organization-wide (aka, enterprise-scale) authentication. In the past, separate systems within one organization have often possessed their own authentication schemes, so that a single user might correspond to multiple principals across an organization's network. The trend today is toward allowing an organization to authenticate its members using central services, and then to assign the verified principal different authorizations in each system the user accesses. The *iLab* architecture supports this pattern through a simple interface to identify a principal (the **`Authenticate()`** method). The interface can be implemented in multiple ways at a single institution. Some of these implementations may depend on backend enterprise-scale authentication. At MIT, this corresponds to campus-wide Kerberos IDs. We refer to this as *external authentication*. The standard Service Broker implementation will provide a simple ASP.NET password based-authentication scheme as a reference implementation. We will refer to this reference implementation as *native authentication*. It is conceivable that at MIT we may need to employ both for a particular class. For instance, some of the students accessing a lab through an MIT Service Broker may not be MIT-registered students and, therefore, may not possess Kerberos IDs. They would be forced to use native authentication.

It is important to realize that in the case of enterprise-scale authentication the Service Broker Authentication API will remain a true API that only defines an interface through which the user can access the appropriate enterprise (e.g., Kerberos) services. Furthermore, the API will not provide methods to create or alter enterprise-scale IDs. The methods that perform those functions against the simple native authentication implementation cannot be applied to enterprise-scale IDs.

In order to establish the equivalence of an externally defined principal with a native principal, the *Authentication* and *Administrative APIs* separate the act of *creating* a principal from that of *registering a pre-existing* principal. If external authentication is used, then the principal is created outside of the iLab system. If native authentication is used, then an administrator with the appropriate privileges creates the new principal using the **`CreateNativePrincipal()`** method of this API. In both cases it will then be necessary to link the principal to an iLab user record via the Administrative API's **`AddUser()`** method. Note that iLab users are identified most directly by an integer (**`userID`**) although they also possess a string **`userName`** that is unique across a namespace shared with iLab **`Group`** instances. This has the further advantage that the string identifying the principal and the **`userName`** of the iLab user record need not be the same. The string **`principalID`** must be unique within an authentication domain. But consider the following conflict and solution. Suppose an iLab administrator created a native account with the principal and userName **`jsmith`**. A later class decided to use Kerberos IDs, and it included a second student with the Kerberos ID **`jsmith`**. The second student can continue to use the Kerberos ID because it is unique in the Kerberos domain, but it must be mapped to a different iLab **`userName`**, e.g. **`jsmith2`**, via the administrative **`AddUser()`** method.

In summary, Service Broker authentication will eventually proceed along two tracks:

Native Authentication

The principal is created via **`CreatePrincipal()`**. Before the user can execute any privileged methods, he or she must also possess a user record created via the administrative **`AddUser()`** method, which links the **`principalID`** to the new and **`userName`**.

External Authentication

The user possesses a principal maintained by the overall organization. It is not necessary to create a principal, but this preexisting principal must be registered at the same time that the appropriate user record is created via the administrative **`AddUser()`** method.

Methods

General Authentication Methods

Authenticate

Purpose:

/* Performs whatever actions including GUI interaction to identify and authenticate the user who has initiated the current session. */

Arguments:

`int userID`
/* The ID of the user. */

`string password`
/* The user’s password. */

Returns:

`bool success`
/* **`true`** if the user has been authenticated; **`false`** otherwise. */

Native Authentication Methods

CreateNativePrincipal

Purpose:

/* Creates a new principal in the native authentication system. */

Arguments:

`string userName`
/* The requested **`userName`**. */

Returns:

`long userID`
/* the new **`userID`** (**`>0)`** if the native principal was successfully created, **`-1`** otherwise. */

Note:

Not currently implemented.

RemoveNativePrincipals

Purpose:

/* Removes principals from the native authentication system. */

Arguments:

`int[] userIDs`
/* The IDs of the principals to be removed. */

Returns:

`int[] unRemovedUsers`
/* An array of the users who were not successfully removed, i.e., the ones for which the operation failed. */

Note:

Not currently implemented.

ListNativePrincipals

Purpose:

/* Enumerate all known native principals. */

Arguments:

`none`

Returns:

`int[] userIDs`
/* An array of all known native principals. */

Note:

Not currently implemented.

SetNativePassword

Purpose:

/* Sets the password of the specified native principal. */

Arguments:

`int userID`
/* The ID of the native principal whose password is to be changed. */

`string password`
/* The new password. */

Returns:

`bool success`
/* **`true`** if the change was successful; **`false`** if the new password was of inappropriate form or the native **`userID`** unknown. */