WebISO
Internet2 Draft
Expires: April 26, 2004

Dors
University of Washington

October 27, 2003

# WebISO: Target-Side Integration Models

## draft-internet2-webiso-target-side-models-01

## Status of this Memo



## Abstract

This document describes the issues related to target-side integration of web-based applications with "WebISO" systems: systems that provide organization-wide web-based sign-on services. This document describes the origin of the application integration problem and compares the two common target-side models used to integrate applications to use WebISO systems for authentication.

The primary audience of this document is vendors, developers, and deployers of web-based applications, but WebISO system implementors may also find insights for making their systems more friendly to application integration.

---

## 1. Introduction

### 1.1 Background

This document assumes a casual familiarity with the terminology, common components, characteristics, and services of WebISO systems, as introduced and developed in the **WebISO: Service Model and Component Capabilities** document.

### 1.2 History: Rooted in Infrastructure Poverty

Developers of web-based applications find themselves at the crossroads where user authentication is concerned. Historically, they couldn't rely on any authentication services being

available to their applications, so they did what anyone would do and built user management and authentication sub-systems into their applications. This is understandable. It's about the only way an application can be deployed in an infrastructure-poor environment.

Today, many universities and other large organizations are deploying WebISO systems to support initial sign-on, and very often single sign-on (SSO), services to web-based applications throughout the enterprise. These WebISO systems very often leverage, and therefore imply the pressence of, existing enterprise authentication services within the organization, such as those built on Kerberos or LDAP. With these forms of "middleware" in place, developers finally have sufficient reason to decouple authentication from their applications and support external means of user authentication.

### 1.3 The Problem: Too Much of a Good Thing

Organizations that have deployed, or are in the process of deploying, WebISO services are eager for a return on their investments. They evangelize these services to internal and external application developers alike: "Please use our WebISO system for authenticating users to your application."

Software vendors and external content providers are particularly prone to these integration requests. And anyone on the receiving end of more than a couple of them soon learns of the variety of existing WebISO packages (e.g. A-Select, Cosign, Pubcookie, Yale/CAS), each with its own peculiar name and way of doing the same basic thing: cross web-server, single sign-on user authentication.

The challenge, therefore, for developers of web-based applications marketed to organizations with deployed WebISO systems is:

- making sense of the basic target-side WebISO integration models,
- understanding the implications of each model on application design and integration,
- and then making sound technological, as well as business, decisions about which authentication methods and mechanisms to support based on the objectives and specific requirements of a given application and its market.

The intent of this document, then, is to help developers confront this challange by revealing the overlap that exists among different WebISO systems with regard to application integration. There are, in fact, just two common target-side integration models, described in detail below, that WebISO systems use to interface with applications: the container-based model and the code-library model.

### 1.4 Scope

Fortunately for application developers, what one WebISO system does one way, another most likely will do in a very similar way, particularly on the target side. However, an effective examination of this commonality has to consider a few limits on what WebISO systems do and do not provide.

First, WebISO systems are oriented toward authentication. That is, they help to establish the

identity of users, and then provide authentication information about a user to target applications. A simple userid is the most common form of identifier, which may or may not be tied to a specific realm (see "" below). Some WebISO systems, however, may represent authentication information as a set of different values. Others may provide additional user attributes for authorization purposes. Others may not provide user identity at all, choosing instead to preserve user privacy by delivering a temporary "handle" with which the target can attempt to acquire further information. But, principally, WebISO systems provide authentication services, and that's what developers can expect.

Next, WebISO systems most often provide a one-time service that coincides with initial sign-on to an application. They generally do not provide an interface for making further requests at arbitrary moments during an established application session. Once authentication information has been delivered to the target system, the WebISO system henceforth does very little for the application - perhaps some session management, but that's all.

Similarly, we add that WebISO systems do not provide directory services. They provide enough authentication (and occasionally authorization) information upon which an access control decision can be based. And that's all. Applications should be able to use other services, such as directory services based on LDAP, to fulfill any mid-session needs for user attributes above and beyond what is provided by a WebISO system.

Finally, there are whole target-side integration scenarios not discussed in this document. For example, one might be curious how WebISO systems interact with backend applications within "multi-tier" environments, or how WebISO systems work when they are just one of multiple authentication mechanisms from which a target application can choose. These are interesting scenarios, but their solutions tend to differ case by case and therefore are better covered separately.

### 1.5 Motivations

If each WebISO system does things its own way, why undertake this document?

- Because there is significant overlap.
- Because as a community it makes sense to share common practices.
- Because the similarities among WebISO systems - in terms of common issues as well as target-side integration models - reveals broad but clear strategies for application design.

By describing the overlap among WebISO systems, we hope to supplement the documention already provided by individual systems, and, in doing so, encourage application developers to support more general WebISO-based models of authentication and application sign-on. At the same time, we hope to suggest how WebISO system designers can better accentuate the similarities in their own target-side WebISO modules and libraries.

Ultimately, we'll be able to gauge our success by the increased understanding of the target-side issues and by the number of applications that integrate more easily with a wide number of WebISO systems.

---

## 2. General Target-Side Issues

Before examining how applications invoke the machinery of WebISO systems, lets review a number of general integration issues inherent to all methods.

### 2.1 User Principals

WebISO systems make user principals available to target applications. The format of the principal may be a simple userid (e.g. "sally") or something richer, perhaps with realm information attached (e.g. "sally@example.edu"). Applications that optionally can work with realm information are going to be more integration friendly than those that cannot.

Does an application get to choose the format of the identifier it receives from the WebISO system? Does it get some sort of object with several representations wrapped up together? The answer depends on the WebISO system being used.

### 2.2 Session Management

Although WebISO systems help users to sign on to applications, they do not always establish and maintain a session for subsequent requests by each user to the same target application. Some do. Some don't. It typically depends on the deployment model being used. Therefore, developers must consider which models they want to support, and determine what impact this has on session management, before designing, if necessary, any for the application itself.

Note: Some WebISO systems also provide SSO-creation, SSO-expiration, and password-last-entered times, which can be factored into application session timeouts as well.

### 2.3 Logout

WebISO systems provide single sign-on (SSO) services to potentially many applications across multiple web servers, but they do not usually provide single logout across the same set of applications.

Logout issues are complex. For example, does logging out of one application that uses a WebISO system for initial sign-on also trigger a domino effect of logging out of all the other authenticated applications the user visited during the same browsing session? What do users expect when they logout of an application: local or global effects?

Best practices are still being established for logout and for changing context between different users within the same browsing session. Some WebISO systems support some form of "global" logout: either by tieing application logout to SSO-session logout, so that the user must reauthenticate in order to sign on to any previously unvisited applications; or by terminating simultaneously all the current application sessions obtained through that SSO session, plus the SSO session itself. Policies may differ from site to site, so developers should design for flexibility with regard to logout.

### 2.4 Application Control

Application developers naturally want precise control over application behavior, including initial sign-on, session management, etc., and they often view control as essential to the deployability

of their applications.

People who set up, maintain, and manage policy for an organization's WebISO system and infrastructure often view things differently, ostensibly siding with "what's best for users" when they try to withhold control and flexibility from applications in the name of consistency, uniformity, and some base level of security, all of which contribute to their more wholistic views of deployability.

Application control is at the heart of the remaining sections of this document, which focus on the specific models and interfaces that applications use to integrate with WebISO systems.

---

## 3. The Container Model

This section describes the "container" model of target-side integration between an application and a WebISO system.

### 3.1 Introduction

The "container" model turns the web server itself (or servlet container, web service, etc.) into a kind of authentication filter or container. In this model, applications are said to be "protected by" the authentication container. Therefore, they can rely on the container to invoke the WebISO system on their behalf and to deliver the resulting authentication information thru whatever mechanisms and interfaces are typical for that web server environment; for example, using environment or server variables.

### 3.2 Defining Characteristics

The defining characteristics of the container model are:

- The application must trust the container for authentication information.
- Authentication is implicit and required. The application will not be called unless authentication succeeds.
- Any options provided by the WebISO system to affect authentication behavior are configured at the container level (e.g. .htaccess in Apache), not at the application level.
- A user principal and any other authentication information is made available to an application thru the web server environment.

### 3.3 Interface

The container model most often exposes information thru the web server environment itself, using environment or server variables or whatever the environment supports for this purpose. Minimally, this interface provides the following variables:

- remote_user: the user principal of the authenticated user.

- auth_type: the type of authentication required by the container to access the contained application.

Some WebISO systems offer additional information, but this practice isn't as common and the variable names may differ from one system to another. They include:

- remote_realm: the realm associated with this particular user principal.
- sso_creation_time: the time the user's single sign-on session was created.
- sso_expiration_time: the time the user's single sign-on session expires; authentication information might be considered invalid after expiration.
- passwd_last_entered_time: the time the user last entered his or her password in a username/password verification challenge-response.

There is no standard naming convention for any of these variables, therefore application developers should make them configurable to create maximally deployable applications.

### 3.4 Similarites to HTTP Basic Authentication

Note that some of these variables are identical to those used with HTTP Basic Authentication on many web server environments. Apache, for example, uses the REMOTE_USER and AUTH_TYPE environment variables, as do many WebISO containers. With Tomcat, you can set "tomcatAuthentication=false" to operate behind an authentication container - HTTP Basic or WebISO - allowing applications to use "getRemoteUser()" to access authentication information. One common exception to this rule is the Microsoft IIS web server environment, where it is difficult, though not impossible, to populate the REMOTE_USER server variable. As a result, WebISO containers generally choose a different variable on IIS, such as HTTP_REMOTE_USER.

In the most common case, however, the authenticated environment created by the container model is identical to that created by the use of HTTP Basic Authentication. For the purposes of authentication, an application that works with HTTP Basic Authentication should also work, without modification, with the container-based WebISO integration model. This interoperability between authentication methods can simplify development, since it's very easy for a developer to create a test environment that uses HTTP Basic Authentication.

Note, however, that the similarities with HTTP Basic Authentication end with the concept of an authentication container and how it delivers information to an application. Developers shouldn't expect to use the actual HTTP headers associated with Basic Authentication. With most WebISO systems, Basic Authentication is never actually used; the application environment is merely emulated. It is the job of the container-based model to mask the true mechanisms of the WebISO system. As with HTTP Basic Authentication, as long as it delivers the required goods, applications shouldn't care how it happens.

### 3.5 Container Configuration

[Fixme: still under construction here.]

What does the container model mean to application deployment, where the rubber hits the road? That is, for someone with an application in one hand, and a WebISO system in another, what are

the common integration and configuration practices that underlie this model?

- per directory/location configuration of authentication options
- different protected services on a single host [Fixme: meaning?]
- session management and logout?
- require https: connections?

### 3.6 Container Constraints

[Fixme: still under construction here.]

The container model places a number of constraints on applications.

First, an application cannot control the exact timing of authentication. The container enforces authentication ahead of application execution; from the application's perspective, either the user is authenticated already or isn't going to be at all. This may not be a concern when there is a clear division between an application's unauthenticated and authenticated pages or when the entire application is authenticated.

Similarly, if an application has a page (i.e. a specific URI) that may or may not require authentication, then the container model forces a workaround, because a single page inside a WebISO container cannot be both authenticated and unauthenticated. The container is configured during setup, so the behavior it imposes is static. Again, authentication is either on or off. If an application requires run-time decision-making - to interact with the WebISO system mid-session - it may require bouncing the user between different URIs, each configured to get a desired effect. The code library model (see "" below) may be the better approach here.

### 3.7 Authorization

While the container model is well established and common for authentication (that is, most WebISO systems offer an Apache module or an ISAPI filter, and a variety of applications have been tailored to use them) authorization is considerably more interesting. Fundamentally it is not acceptable simply to use the pressence of authentication information provided by a WebISO container as implicit authorization; authorization should be done separately and explicitly, based on the circumstances of the application's deployment.

The primary types of authorization used along side of the container model include:

- Application specific, local authorization. The application uses the authentication information it receives thru the container to make its own inherent authorization decisions; this is perfectly acceptable.
- Container-based authorization. The web server environment provides a second container through which authorization can be configured. For example, various Apache modules can be layered atop a WebISO container for authorization (e.g. mod_auth_ldap, mod_auth_pam). Even simpler, Apache's own core htaccess directives can be used to define a list of authorized users.
- Native, platform-specific authorization. For example, Microsoft IIS can leverage

ACLs defined in the local Windows domain or Active Directory forest for authorization.

---

## 4. The Code Library Model

This section describes the "code library" model of target-side integration between an application and a WebISO system.

### 4.1 Introduction

With the "code library" model, an application developer uses a programmatic interface to a set of functions (objects, methods, etc.), as defined by a separate WebISO code library, to implement within an application all the necessary steps in the WebISO-based authentication process; all of which is contingent upon an implementation of the code library written in the developer's desired language.

As the following sections describe, this approach, which might also be called the "API" model, offers the application developer the most control over the authentication process.

### 4.2 Defining Characteristics

The defining characteristics of the code library model are:

- The WebISO system defines an API for "applications" to use. (Which is to say, static content is something of a challenge - where's the application?)
- The WebISO system provides language-specific bindings for the API in the form of code libraries, classes, objects, etc.
- Applications use the language-specific bindings to implement all the necessary steps defined by the WebISO system's API to properly authenticate users.

### 4.3 Interface

The interfaces provided by WebISO systems that offer a code library differ from one implementation to the next and are influenced by the design and flow of the WebISO system itself, its API, and the way the API is represented in specific language bindings. Developers may expect functions that: initialize or prepare an authentication request, initiate the request, receive and process the response, and, finally, retrieve the authentication information.

### 4.4 Advantages

There are a number of advantages to using the code library model.

Its greater flexibility means it imposes fewer constraints on a developer's control over the flow of an application into and out of authentication.

Applications can easily present different views of the same page, at the same URI, depending on

the type of users involved and their ability to authenticate locally (using the WebISO system), anonymously, as a guest, or not authenticate at all.

Applications can arbitrarily force users to re-authenticate at any time; for example to re-establish presence before allowing a user to access a particularly sensitive part of the application.

Parameters of the WebISO system can be controlled at run-time. Therefore, developers can select authentication types, the sequence in which they are tried, return style, etc., all at run-time, based on an application's own internal logic and the context of the session (e.g., the User-Agent header). The container model can mimic this using mulitple URIs with different configuration options, but this method doesn't scale well as the number of options and permutations grows large.

Developers can access additional details thru the API about the user or the authentication process that would be difficult or impossible to access in the container model. [Fixme: for example?]

Some applications are themselves written as proprietary containers (e.g. Apache modules, ISAPI filter and extensions, etc.) that fit within specific web server frameworks. And they may or may not have pre-written authentication layers compatible with the container-based WebIO model. In these cases, a WebISO system that supports only the container model will not be able to integrate with the proprietary container-based application. On the other hand, if the application supports login or authentication plugins, or if the developers are willing to work with a WebISO code library, a WebISO system that provides an API can still be integrated with the proprietary container-based application.

### 4.5 Disadvantages

There are a number of disadvantages to using the code library model too.

External code is incorporated into applications themselves, which raises concerns with versioning of the WebISO system's API and language bindings, timely incorporation of security fixes into applications, and the general likelihood that changes to the WebISO system will require changes to the application that otherwise would not have been necessary if no external code was being used.

The learning curve is steeper. That is, programmatic APIs are inherently more complex than the declarative methods offered by the container model. Developers have to understand a WebISO system's API and call the proper functions in the proper order to successfully authenticate a user.

Security may be handled inconsistently. Developers may not consider the same threats and use the same countermeasures that are applied consistently and implicitly through the container model.

The application deployer (i.e. the system administrator or webmaster) loses a degree of control over configuration. That is, with the code library model, the application developer controls what's configurable in the application, and this may not be enough to the deployer, who is often more aware of the local configuration requirements. In the container model, configuration remains more in the hands of the application deployer.

Static content is problematic for the code library model: there's no application in this case.

Therefore, static HTML files, collections of images, etc., require use of the container model or the creation of some simple but extra application that implements access controls. (Of course, a simple container can be implemented, with relative ease, from a WebISO's code library, and can then provide the basic, container-style authentication that static content (and some applications) require.

---

## 5. Case Studies

Case studies.

Deploying a commercial/open source application with Apache: IMP

Deploying a commercial/open source application with IIS: footprints

Deploying a commercial/open source servlet application: CHEF

---

## References

---

## Author's Address

Nathan Dors (Editor)
University of Washington
4545 15th Ave. NE, Box 354841
Seattle, WA 98105-4527
US
**Phone:** +1 206 543 0624
**EMail:** dors@washington.edu
**URI:** http://staff.washington.edu/dors/

---

## Appendix A. Acknowledgements

The editor acknowledges the contributions of many members of the Internet2 WebISO working group, in particular Wes Craig, Larry Greenfield, Karsten Huneycutt, Kevin McGowan, and Bob Morgan, who regularly provided guidance, ideas, opinions, and much of the text for this document.