A Simple Collaborative Method in Web Proxy Access Control for Supporting Complex Authentication Mechanisms

Shingo Takada*, Akira Sato*, Yasushi Shinjo*, Hisashi Nakai†, Koichi Sakurai* and Kozo Itano*
*Graduate School of Systems and Information Engineering
University of Tsukuba,
Tsukuba, Japan
Email: {takada@softlab.cs, akira@cc, yas@cs, sakura@softlab.cs, k3itano@cs}.tsukuba.ac.jp
†Graduate School of Library, Information and Media Studies
University of Tsukuba,
Tsukuba, Japan
Email: nakai@slis.tsukuba.ac.jp

***Abstract*—Modern authentication mechanisms, including Shibboleth and OAuth, provide user attributes such as affiliations and e-mail addresses. Conventional collaborative methods have problems using such attributes in egress access control for the Web. This paper proposes a new collaborative method using Web browsers, proxy servers, and authentication servers. The proposed method simplifies communications among these elements by using a trusted shared repository that stores user attributes. A new authentication mechanism can be added to the system by deploying an authentication server of the new authentication mechanism. This authentication server is a Web application and stores user attributes in a shared repository associated with the user identifiers. When proxy servers receive requests from Web browsers, the proxy servers retrieve user attributes from the shared repository and the proxy servers decide whether or not to allow access to external Web pages in accordance with the URLs and relevant user attributes. Unlike in a standard such as the Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO), neither Web browsers nor proxy servers are required to include extensions for authentication mechanisms. On the basis of the simple collaborative method, the authors have implemented an egress access control system for the Web that performs user authentication with Shibboleth and Facebook. The access control system has been operational in a university library for more than a year.**

***Index Terms*—Web proxy servers, access control, user authentication, Shibboleth, OAuth**

I. Introduction

In public spaces such as libraries and hotels, network administrators often perform egress access control. Egress access control is access control that allows or disallows internal users to access external networks according to rules. An example of this is a captive portal performing user authentication before internal users access the Internet. In a captive portal, network administrators can write access control rules with IP addresses and port numbers.

In some organizations, network administrators are required to write access control rules with URLs. We call this *URL-level access control*. For example, network administrators in a library may wish to allow visitors to access external Web sites such as an Online Public Access Catalog (OPAC) and e-journals without user authentication. They may also wish to allow visitors to access any external Web sites with user authentication. Parental control of Web sites is another example of URL-level access control. In this paper, we focus on URL-level egress access control performed by Web proxy servers.

In typical URL-level egress access control systems, the following three types of elements work together: 1) Web browsers that are operated by users, 2) Web proxy servers that handle requests sent from Web browsers, and 3) authentication servers that perform user authentication. In conventional collaborative methods, these elements collaborate with one another as follows.

1) Web browsers send authentication factors (typically user names and passwords) to authentication servers.
2) The authentication servers verify these factors and send credentials back to the Web browsers. Here, a *credential* is proof of an authentication result and is often represented as a cookie, token, or ticket.
3) The Web browsers send URLs of external Web pages to Web proxy servers along with the credentials. The Web proxy servers verify the credentials and decide whether or not to allow access to the Web pages associated with the URLs according to the credentials and rules.

In Step 3, Web browsers and proxy servers communicate in accordance with standards including Basic Authentication, Digest Authentication, and the Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO) [8][11][15]. GSSAPI stands for the Generic Security Service Application Program Interface[18].

These conventional collaborative methods have problems with modern complex authentication mechanisms because these mechanisms use identity servers that provide *user at-*

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States

DOI 10.4108/icst.collaboratecom.2012.250446

tributes, i.e., supplemental information other than user identifiers. For example, the identity servers of Facebook provide names, groups, and e-mail addresses to connected Web applications. Facebook uses OAuth [6] to transfer such user attributes to Web applications. In Shibboleth [1], a single-sign-on system, Web applications can obtain user attributes such as display names, affiliations, and e-mail addresses from identity servers. Network administrators wish to use these user attributes for URL-level egress access control. However, this is no trivial matter because most standards [8][11][15] require authentication mechanism-specific communication between Web browsers and Web proxy servers. Adding a new modern authentication mechanism requires developing both a Web browser extension and a Web proxy extension. We will describe the details of these problems in Section III.

In this paper, we propose a simple collaborative method for Web proxy access control that enables the use of advanced user attributes of modern complex authentication mechanisms. Our method simplifies communications among Web browsers, proxy servers, and authentication servers by using a trusted shared repository that stores user attributes. These elements work together as follows:

1) Web browsers send authentication factors to authentication servers along with authentication mechanism-independent user identifiers.
2) The authentication servers verify authentication factors, obtain user attributes from external identity servers, and store the attributes on a trusted shared repository associated with the user identifiers.
3) The Web browsers send the URLs of external Web pages to Web proxy servers along with user identifiers. The Web proxy servers then retrieve user attributes from the trusted shared repository with the user identifiers and decide whether or not to allow access to Web pages with the URLs according to the user attributes and rules.

In our simple collaborative method, a new authentication mechanism can be added to the system by deploying an authentication server. Since an authentication server is a regular Web application, we can implement such a Web application with widely available libraries and tools. Unlike in SPNEGO, neither Web browsers nor proxy servers are required to include extensions for a new authentication mechanism.

On the basis of our simple collaborative method, we have implemented a Web proxy access control system that performs user authentication with Shibboleth and Facebook. The Shibboleth implementation has been fully operational in our university library since more than a year ago.

II. Related Work

A. Access Control with User Attributes for Base Web Servers

The Apache HTTP server allows writing access control rules with not only user identifiers but also groups of users belonging to Basic Authentication or Digest Authentication [9]. In addition to simple groups, many access control mechanisms with advanced user attributes have been proposed and implemented for base Web servers. The method in [22] enables us to use the role of a users in Role-Based Access Control (RBAC). The methods in [16][7][5] enable us to use attributes in X.509 certificates. The method in [4] enables us to use proof based on proof-carrying authorization. All these mechanisms perform ingress access control on base Web servers. Our method performs egress access control on Web proxy servers.

B. Parental Control and Content Filtering for Web Pages

Parental control enables parents to restrict children's use of computers. This includes restricting Web access. This type of control can be provided via operating systems [3], extensions of Web browsers [10], and proxy servers [21]. These methods use the authentication mechanisms of operating systems and simple password-based authentication mechanisms. Our method performs egress access control using advanced user attributes in modern complex authentication mechanisms.

C. Egress Network Access Control

Captive portals are often used to perform egress network access control in public spaces such as libraries and hotels [2][27]. A captive portal intercepts access to external servers and enforces user authentication before allowing access. A captive portal usually performs egress access control at the IP address level in routers. The method in [25] performs egress access control at the name level in DNS servers and routers. None of these systems can perform egress access control at the URL level.

D. Web Proxy Authentication and SPNEGO

RFC 2616 [8] and RFC 2617 [11] include standards of user authentication between Web browsers and proxy servers. In these standards, Web proxy servers ask Web browsers to send credentials with `Proxy-Authenticate` headers in HTTP responses, and the Web browsers comply by sending the credentials to Web proxy servers with `Proxy-Authorization` headers in HTTP requests. These standards enable Basic Authentication and Digest Authentication in proxy servers.

The Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO) [29] is an authentication mechanism that supports negotiation between Web browsers and base Web servers. GSSAPI stands for Generic Security Service Application Program Interface [18]. SPNEGO is designed to perform user authentication with Kerberos [24] and Windows NT LAN Manager (NTLM) [13].

These standards can be used together. i.e., we can use SPNEGO in `Proxy-Authorization`[1]. However, using SPNEGO in proxy servers does pose some problems, which we discuss in detail in Section III.

[1] http://sourceforge.net/projects/squidkerbauth/
http://wiki.squid-cache.org/Features/NegotiateAuthentication

E. Shibbolizing Web Proxy Servers

Komura et al. have achieved Shibboleth authentication in Web proxy servers that can perform URL-level egress access control [26]. In their method, Web proxy servers identify users by cookies [17]. When a Web browser sends a URL to the Web proxy server to access an external Web page, it is redirected to a Shibboleth authentication server that is running on the same host as the proxy server. The Shibboleth authentication server performs access control with user attributes provided by Shibboleth and redirects to a fake URL called a *phantom URL*. This phantom URL has the same domain value of the external Web page. The proxy inserts a session cookie into the phantom URL and redirects it to the external Web page. The Web browser then sends the URL to the Web proxy again, along with the session cookie. The proxy allows access if the session cookie is valid. This cookie insertion is repeated for each domain of the external Web pages.

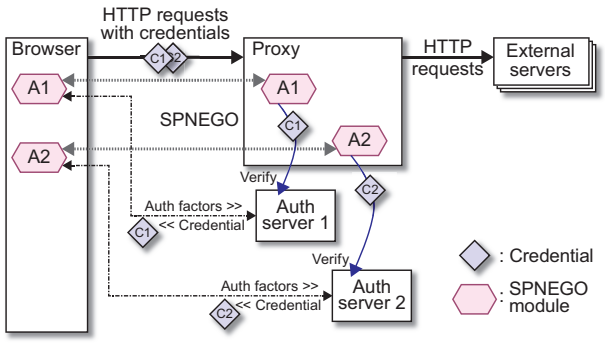
This method, however, has a couple of problems. First, it is very complex, requiring fake URLs and complex redirections. Second, it is specific to Shibboleth. Our proposed method, in contrast, is a simple and generic method for URL-level egress access control.

III. Simple Collaborative Method for Web Proxy Access Control

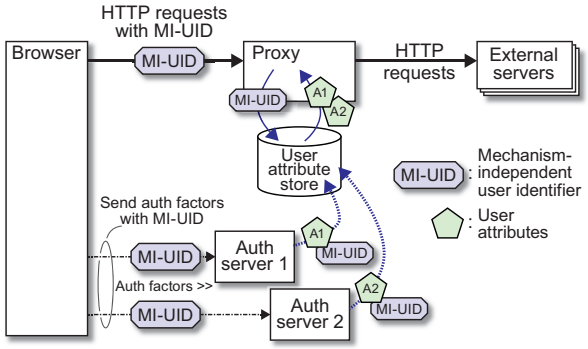
In this section, we describe our simple collaborative method for Web proxy access control.

The collaboration process in a conventional method and the proposed method is shown in Fig. 1. A conventional Web proxy access control system typically consists of a Web browser, a Web proxy server, and two authentication servers (Fig. 1(a)). First, the Web browser sends authentication factors, usually user names and passwords, to the authentication servers. In a public key infrastructure (PKI), authentication factors are X.509 certificates. Second, the authentication servers perform user authentication with the received authentication factors. If these user authentication actions are successful, the authentication servers send the credentials back to the Web browser. (Here, a "credential" is proof of the fact that the user has been authenticated.) Third, the Web browser sends a URL with credentials to a Web proxy server over SPNEGO. Fourth, the Web proxy server verifies the credentials and performs access control in accordance with the credentials and the URL.

Conventional collaborative methods using SPNEGO work well for Kerberos. However, they face difficulties when used with modern complex authentication mechanisms such as Shibboleth and OAuth. First, in SPNEGO, Web browsers and proxy servers communicate in an authentication mechanism-specific way, so adding a new authentication mechanism requires defining a new protocol. To add Shibboleth authentication, we have to define how user attributes are delivered in Security Assertion Markup Language (SAML). To add OAuth authentication, we have to define how access tokens are delivered from Web browsers to proxy servers.



(a) Collaboration among a browser, authentication servers, and a proxy in SPNEGO.



(b) Collaboration among a browser, authentication servers, and a proxy in proposed method.

Fig. 1. Collaboration among Web browser, proxy server, and two authentication servers.

The second problem is that adding a new modern authentication mechanism requires developing both a Web browser extension and a Web proxy extension. The Web browser in Fig. 1(a) includes two extensions, and the proxy server also has two extensions. Needless to say, developing such extensions requires a lot of effort. For example, to add Shibboleth authentication, we have to develop Web browser extensions for Firefox, Internet Explorer, Safari, Opera, Google Chrome, etc., as well as a proxy server extension. Instead of using the HTTP headers of SPNEGO, we could use cookies to transfer user attributes and access tokens from Web browsers to proxy servers. However, using cookies has its own set of problems, which we describe in Section IV-A.

We address these problems with a simple collaborative method, shown in Fig. 1(b). The Web proxy access control system based on our method consists of a Web browser, a Web proxy server, authentication servers and a *user attribute store*. First, the Web browser sends authentication factors to the authentication servers along with an *authentication mechanism-independent user identifier*. Second, the authentication servers perform user authentication with the received authentication factors. If these user authentication actions are successful, the authentication servers store the user attributes associated with the user identifier in the user attribute store. Third, the Web browser sends a URL along with the user identifier to a Web proxy server. Fourth, the Web proxy server retrieves these

attributes and performs access control in accordance with the user attributes and the URL.

Our simple collaborative method has four distinct advantages over conventional methods.

First, we do not have to define any protocols to deliver user attributes or access tokens from Web browsers to proxy servers over SPNEGO. This also means that we do not have to implement SPNEGO modules in the Web browsers and proxy servers.

Second, we can easily add a new authentication mechanism to the system by deploying an authentication server of the new authentication mechanism. This authentication server is a regular Web application that is protected by the new authentication mechanism. We can implement such applications with widely available libraries and tools. In addition, we can reuse existing authentication modules for base Web servers. We will discuss concrete examples of Web applications in Section IV-E.

Third, the verification of attributes in proxy servers can be done quite simply. User attributes are delivered only through trusted components, authentication servers, and the user attributes store. Proxy servers do not have to verify user attributes that are thus obtained from the trusted user attribute store.

Fourth, our method can easily allow a single user to log in with multiple authentication mechanisms at the same time. For example, a user can log in to both Shibboleth and Facebook simultaneously. This user can then access external Web pages allowed by both the Shibboleth and Facebook authentications.

Our method does have an implementation issue, in that we have to realize authentication mechanism-independent user identifiers. We discuss the requirements of user identifiers in the following subsection.

A. Authentication mechanism-independent user identifiers

Our collaboration method for Web proxy access control systems requires authentication mechanism-independent user identifiers. These user identifiers are the key to enabling collaboration among Web browsers, authentication servers, proxy servers, and the attribute store.

In the proposed method, user identifiers should have the following characteristics:

- Be unique within a local area network.
- Be authentication mechanism-independent.
- Must not change after user authentication.
- Must be strong against spoofing attacks.
- Must be delivered from a Web browser to a proxy server in every HTTP request.

B. The User Attribute Store

The user attribute store is a trusted key-value store for storing user attributes associated with user identifiers. The user attribute store should be *isolated* from Web browsers.

The user attribute store manages the following entries:

```
(<user identifier>, [<attribute_1>, ...])
```

Each entry consists of a user identifier and a list of user attributes. When a user logs in, an entry is created by an authentication server. The entry of a user should expire after the user logs out.

The user attribute store provides following procedures:

`append_attr(uid, attr)`
: This procedure adds attributes (`attr`) to the entry of the user identifier (`uid`). If there is no entry, the user attribute store creates a new one. This procedure is executed by authentication servers.

`get_attr(uid)`
: This procedure returns the list of attributes that are associated with the user identifier (`uid`). It is executed by proxy servers.

`remove_attr(uid)`
: This procedure deletes the entry of the user identifier (`uid`). It is executed by authentication servers.

C. Proxy servers

Proxy servers are Web proxy servers that perform egress access control at the URL level. Proxy servers run on the borders between the local area network to external networks and relay HTTP requests from Web browsers running in the local area network and Web servers running in external networks. Network administrators describe a policy and install it on proxy servers.

In the proposed method, network administrators can describe a policy by using advanced user attributes of modern complex authentication mechanisms. For example, network administrators in a library can allow a "staff" of Shibboleth to access any external servers while they allow a "student" to access only external e-journals.

While network administrators can use advanced user attributes, they can describe a policy in a uniform and authentication mechanism-independent way. When we add an authentication mechanism, we do not have to add extensions to proxy servers. Network administrators can write mechanism-independent rules as well as mechanism-specific rules. For example, administrators can write a rule that allows users to access external Web pages if the users provide their e-mail addresses. The user attribute e-mail address can be obtained from identity servers connected to OAuth, Shibboleth, and other authentication mechanisms.

D. Authentication Servers

Authentication servers perform user authentication and save user attributes in the user attribute store. Since a modern complex authentication mechanism often requires multiple servers to complete a single user authentication process, we allow that in our method user authentication can be done by multiple servers. For example, the user authentication in Shibboleth is done by two servers: a Shibboleth service provider and a Shibboleth identity provider.

Our method does not mediate communication between Web browsers and individual authentication servers. This improves the modularity of authentication servers. Each authentication server can obtain user attributes from an identity server in an authentication mechanisms-specific way. For example, an

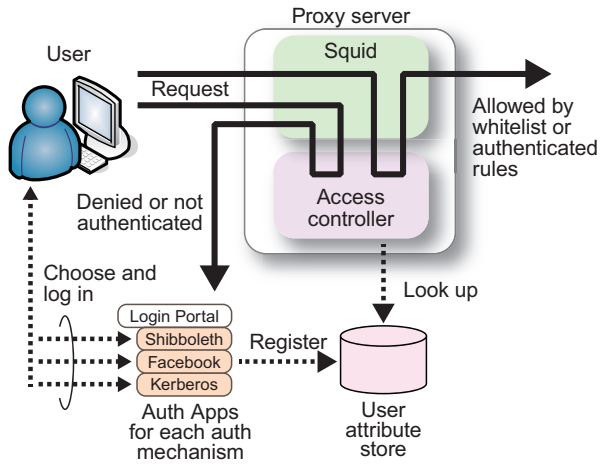


Fig. 2. Implementation overview.

authentication server of Shibboleth can obtain user attributes from a token in Security Assertion Markup Language (SAML) [20]. The authentication server receives this token from a Web browser through a redirection. An authentication server connected to Facebook obtains user attributes from external identity servers with a remote procedure call. After obtaining user attributes, authentication servers save them into the user attribute store. Proxy servers can thus retrieve various user attributes in a uniform way.

In Section IV-E, we describe the implementations of authentication servers of Shibboleth and Facebook.

IV. Implementation of Egress Access Control System for the Web

In Section III, we described our simple collaborative method for URL-level egress access control. In this section, we describe the implementation of a URL-level egress access control system based on our simple collaboration method. This implementation supports modern user authentication mechanisms of Shibboleth and Facebook OAuth as well as a legacy one of Kerberos.

An overview of the implemented system is shown in Fig. 2. The system consists of Web browsers, the proxy server, the user attribute store, a **Login Portal**, and **Auth Apps**. The proxy server consists of two parts: a Squid proxy server and an extension module called the **Access Controller**. The Squid proxy server invokes the Access Controller every time it receives an HTTP request. The Access Controller returns true if the access is granted. Otherwise, it returns false.

The Login Portal is a portal page for choosing authentication mechanisms. On this page, users choose the necessary authentication mechanism to access external Web resources. Each authentication mechanism-specific procedure is realized by an Auth App that is a Web application.

A. Implementation of authentication mechanism-independent user identifiers

We discussed the requirements of authentication mechanism-independent user identifiers in Section III-A. In this subsection, we discuss three implementations of user identifiers.

The first implementation is an ideal one and uses HTTP headers. We introduce two new HTTP headers:

```
Proxy-Set-Cookie: <set-cookie-string>
Proxy-Cookie: <cookie-string>
```

These headers enable cookies within a proxy server. The first header is similar to `Set-Cookie` in [17] but differs in that it is used in a reply message delivered from a proxy server to a Web browser. The second header is similar to `Cookie` but differs in that a Web browser sends a user identifier to both an authentication server and a proxy server.

The second implementation is also an ideal one and uses HTTP headers. Instead of introducing new headers, we use an existing one:

```
Proxy-Authorization: Basic <rnd_uid>:<rnd_pw>
```

This header is similar to `Proxy-Authorization: Basic` in [11] but differs in that it sends a random uid and a random password to identify the user. During a single login session, a Web browser must send the same uid and password.

These two implementations require changing existing Web browsers and proxy servers.

The final implementation is a practical one and does not use any HTTP headers. We use the IP addresses of client computers that run Web browsers as authentication mechanism-independent user identifiers. The IP addresses satisfy the requirements discussed in Section III-A. A computer's IP address is unique in a local area network and is authentication mechanism-independent. The IP address of a computer can be fixed while a user logs in. If we use intelligent network switches, we can fix the IP addresses of a client computer and prevent IP address spoofing[2]. Some routers with password user authentication use source IP addresses as user identifiers [2][27]. In our target local area network, IP addresses were sufficient for URL-level egress access control.

Using IP addresses as user identifiers makes implementations simple. We do not have to modify Web browsers. We can obtain the IP addresses of computers that run Web browsers in proxy servers and authentication servers.

On the other hand, using IP addresses as user identifiers has limitations. First, we cannot deal with the sharing of IP addresses. For example, we cannot allow the use of Network Address Translation (NAT) in a local area network. Second, we must defend against IP address spoofing with intelligent switches or other techniques. Finally, we should turn off Privacy Extensions for Stateless Address Autoconfiguration

[2]Cisco Catalyst 2960 has Dynamic Host Configuration Protocol (DHCP) Snooping, Dynamic Address Resolution Protocol (ARP) Inspection, and IP Source Guard capabilities that can identify the IP address of each computer and prevent malicious users from carrying out spoofing attacks.

in IPv6 [19]. We can enable this option on the proxy server hosts to prevent external servers from tracking users with IP addresses.

B. User Attribute Store

The user attribute store is a trusted key-value store for storing user attributes associated with user identifiers. We have implemented the user attribute store by using a tuple space of Linda, a coordination language [12]. Since we implemented the user attribute store in Ruby, we used Rinda [23] which realizes tuple spaces for Ruby.

An example of the tuple space is as follows.

```
[ MAGIC_attr, uid1, attr1_shib  ]
[ MAGIC_attr, uid1, attr1_oauth ]
[ MAGIC_attr, uid2, attr2_shib  ]
[ MAGIC_attr, uid3, attr3_oauth ]
```

The first item, `MAGIC_attr`, is a magic string indicating that the tuple contains user attributes. The second items are user identifiers, and the third items are user attributes. In this example, the user associated with `uid1` has two user attributes: `attr1_shib` and `attr1_oauth`.

A tuple is created by an authentication server, as follows.

```
space.write([MAGIC_attr, uid, attr])
```

The procedure `write()` creates a tuple in the brackets "[" and "]" in the tuple space. An authentication server creates a tuple for each user with this procedure after user authentication. Multiple tuples can be created for a single user by multiple authentication servers.

Tuples are retrieved by proxy servers as follows.

```
attr_list = space.read_all(
              [MAGIC_attr, uid, nil])
attr_list.each {|tuple|
   attr = tuple[2]
   use( attr )
}
```

The procedure `read_all()` returns a list of tuples that matches the pattern in the argument. In this code, `nil` means a wild card.

As discussed in Section III-B, the tuple space server should be isolated from Web browsers. In the current implementation, we run the tuple space server, the proxy server, the Login Portal and Auth Apps on a single host. The tuple space server accepts connections only from these servers. Since the tuple space server supports SSL, we can protect it with SSL certificates.

C. Web Proxy and Access Controller

We implemented the proxy server by using Squid[3], an open source Web proxy server. Squid has an extension mechanism to rewrite URLs in HTTP requests. This extension program is called a *URL rewrite program*. We have realized the Access Controller as a URL rewrite program of Squid.

When Squid is executed, it spawns the process of the Access Controller. Each time Squid receives an HTTP request from a Web browser, it passes the requested URL, the IP address of the Web browser, and the request method (GET, POST, etc.) to the Access Controller through a pipe that connects the two. The Access Controller returns the same URL through another pipe if the access is granted. Squid relays the request to the external server and the response from the external server to the Web browser. Otherwise, the Access Controller returns the URL of the Login Portal and Squid sends a redirection message to the Web browser.

[3]http://www.squid-cache.org/

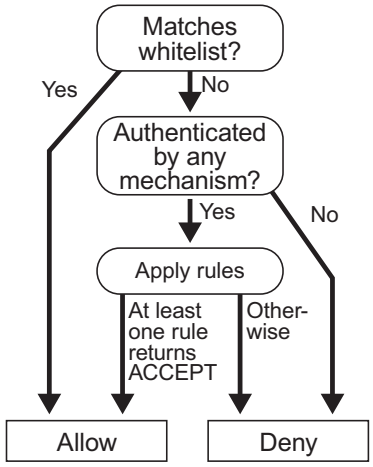


Fig. 3. Access control decision flow.

When the Access Controller receives a URL, an IP address, and a request method, it decides whether or not to allow access to the Web page, as shown in Fig. 3. First, the Access Controller matches the URL with patterns in a whitelist that includes regular expressions and is maintained by network administrators. If the URL matches a pattern in the whitelist, the Access Controller returns the same URL and the request is allowed. Next, it tries to obtain user attributes from the user attribute store with the IP address. If no user attribute entry is found, it means the user is not authenticated and the Access Controller returns the URL of the Login Portal to cause a redirection to it.

If any user attributes are found in the user attribute store, the Access Controller applies rules. We will describe the details of these rules in Section IV-D. If at least one rule allows access, the Access Controller returns the same URL and the request is allowed. If not, the Access Controller returns the URL of the Login Portal.

The Access Controller caches user attributes for performance. It maintains the consistency of the cache, by using Rinda's update notification mechanism.

D. Access Control Rules

Network administrators describe an access control policy as a list of rules, and save these rules in a configuration file belonging to the Access Controller. A single rule consists of a head and a body. The head describes the subjects (users), and the body includes URL patterns and actions.

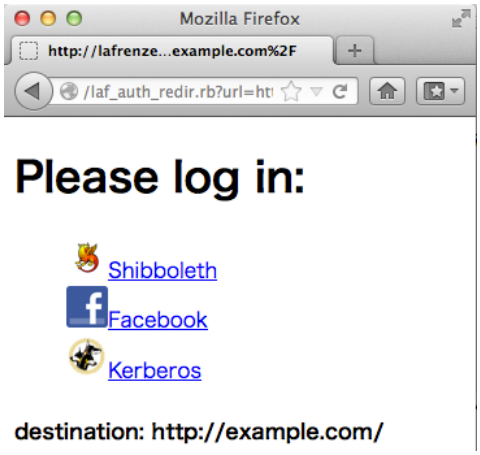


Fig. 4. Screenshot of Login Portal.

The Access Controller evaluates the rules as follows. First, it evaluates the head part with user attributes. If the condition is satisfied, the body part is evaluated. This body part is a list of a URL pattern, a request method (GET, POST, PUT, etc.), and an action. A URL pattern is a regular expression, and an action is either ACCEPT or REJECT.

An example of a rule is:

```
- cond:
    affiliation: "student@.+"
  do:
    - url:     "http://example\.com"
      action:  REJECT
    - url:     "http://bbs\.example\.net"
      method:  POST
      action:  REJECT
  default_policy: ACCEPT
```

This rule is applied to users whose affiliation attribute is "student". If the user sends a request to example.com, it is denied. If the user sends an HTTP POST request to bbs.example.net, it is also denied. Otherwise, the request is allowed because the default policy is ACCEPT.

Another example rule is:

```
- cond:
    group:     "important_visitor"
    email:     ".+@.+$"
  do:
    - url:     "http://.*example\.org/"
      action:   ACCEPT
  default_policy: REJECT
```

This rule permits access to services on example.org if the user provides an email address and belongs to the group "important_visitor".

E. Login Portal and Auth Apps

Login Portal is a portal Web page where a user can choose an authentication mechanism. A screenshot of the Login Portal is shown in Fig. 4. On this page, a user can choose one of three authentication mechanisms: Shibboleth, Facebook, or Kerberos. Clicking any of these links executes the relevant Auth App.

```
#!/usr/bin/env ruby
require 'rinda/tuplespace'

uri   = "http://attr-store-host:port/"
space = DRbObject.new_with_uri(uri)
uid   = ENV["REMOTE_ADDR"];
attrs = {"mechanism"  =>"shibboleth",
         "email"      =>ENV['mail'],
         "affiliation"=>ENV['affiliation'],
         "persist_id" =>ENV['persistent_id']}
space.write(["MAGIC_attr",uid,attrs ])
```

Fig. 5. Auth App for Shibboleth authentication.

```
AuthType shibboleth
AuthName "Shibboleth User Only"
ShibRequireSession On
ShibUseHeaders On
require valid-user
```

Fig. 6. The .htaccess file to protect the Auth App for Shibboleth authentication.

As stated previously, an Auth App is a Web application that mediates an authentication mechanism and the user attribute store. It performs user authentication in an authentication mechanism manner and then stores user attributes into the user attribute store.

Fig. 5 shows the main part of the Auth App for Shibboleth. This is a Common Gateway Interface (CGI) program running on the Apache HTTP server and is protected with the access control description shown in Fig. 6. This access control description activates the Apache module (mod_shib [28]) that allows the CGI program to access Shibboleth user attributes through environment variables.

When a Web browser accesses this Auth App for the first time, it is redirected to a Shibboleth identity provider. This redirection is automatically caused by the file shown in Fig. 6. In the identity provider, the user inputs his/her user name and password, and after the identity provider verifies these, it redirects the Web browser back to the Auth App. The second time, the Auth App (Fig. 5) is granted by the file shown in Fig. 6. When the Auth App is executed, it obtains two Shibboleth attributes from environment variables. One is an affiliation, which is the affiliation of the user (such as "staff" or "student"). The other is a persistent_id, which is an anonymized identifier of the user. These attributes are packed into a single hash table, which the Auth App saves in the user attribute store. The total size of this Auth App for Shibboleth was 14 lines of code.

Fig. 7 shows the main part of the Auth App for Facebook. This is a Web application of Facebook. It is similar to the Shibboleth Auth App in Fig. 5, but there are two differences. First, the Facebook one has to determine if the user has been authenticated by itself. Second, it obtains user attributes not from environment variables but from the class library FacebookOAuth::Client.

The Facebook Auth App checks the parameter "code", which is an access token in OAuth. If no access token is given, the Auth App redirects to a Facebook identity server. After

```
#!/usr/bin/env ruby
require 'cgi'
require 'rinda/tuplespace'
require 'oauth'
require 'facebook_oauth'

uri   = "http://attr-store-host:port/"
space = DRbObject.new_with_uri(uri)
uid   = ENV["REMOTE_ADDR"];

if cgi.params["code"].empty?
  mode = :redir_for_auth
else
  mode  = :auth_done
  token = cgi.params["code"][0]
end

client = FacebookOAuth::Client.new(
          :application_id => APP_ID,
          :application_secret => APP_SECRET,
          :callback => callback_url)

if mode == :redir_for_auth
  auth_url = client.authorize_url(
              :scope => 'user_groups,email' )
  print cgi.header(:location => auth_url)
elsif mode == :auth_done
  client.authorize(:code => token)
  fggroups = get_fbgroup(client)
  group = fggroups.includes?
           IMPORTANT_VISITORS :
           "important_visitors":""
  attrs = { "mechanism" => "facebook",
            "id" => client.info["id"],
            "email" => client.info["email"],
            "group" => group}
  space.write(["MAGIC_attr",uid,attrs ])
end
```

Fig. 7. Auth App for Facebook authentication.

user authentication, the Auth App is executed again with an access token. If an access token is given, the Auth App uses it to obtain the user attributes. In the example shown in Fig. 7, the Auth App obtains three attributes from a Facebook identity server: the user identifier (ID) on Facebook, the user's e-mail address, and a list of groups that the user belongs to. The Auth App then packs these attributes into a single hash table and saves it in the user attribute store. The total size of this Auth App for Facebook was 33 lines of code.

F. Reusing Legacy Authentication Mechanisms for Base Web Servers

In our egress access control system for the Web, we can reuse legacy authentication mechanisms for base Web servers. Fig. 8 shows the main part of the Auth App for Kerberos. This is a CGI program running on Apache and is protected with the access control description shown in Fig. 9. This program and the description of Kerberos are similar to those of Shibboleth in Figs. 5 and 6. The access control description in Fig. 9 actives the module mod_auth_kerb[4]. In Fig. 8, the Auth App obtains the principal name of Kerberos from an environment variable and then uses it to create an e-mail address. It then puts the

[4]http://modauthkerb.sourceforge.net/

```
#!/usr/bin/env ruby
require 'rinda/tuplespace'

uri   = "http://attr-store-host:port/"
space = DRbObject.new_with_uri(uri)
uid   = ENV["REMOTE_ADDR"];
princ = ENV["REMOTE_USER"];
princ_name = princ.split("@")[0]
email = "#{princ_name}@example.edu"
attrs = {"mechanism" =>"kerberos",
         "email"     => email,
         "principal" => princ}
space.write(["MAGIC_attr",uid,attrs ])
```

Fig. 8. Auth App for Kerberos authentication.

```
AuthType Kerberos
AuthName "Kerberos User Only"
KrbMethodNegotiate On
KrbMethodK5Passwd Off
KrbAuthRealms REALM.EXAMPLE.EDU
Krb5KeyTab /etc/httpd/conf/keytab
require valid-user
```

Fig. 9. The .htaccess file to protect the Auth App for Kerberos authentication.

e-mail address and the principal name into a hash table and saves it in the user attribute store.

We can reuse other legacy authentication mechanisms in the same manner as Kerberos, including simple password files, the Lightweight Directory Access Protocol (LDAP), and SSL client certificates.

V. Evaluation

In this section, first, we evaluate our method of providing collaboration among Web browsers, Web proxy servers, and authentication servers that realizes egress access control for the Web. Next, we describe an actual application of the proposed system in our university library. Finally, we discuss the scalability of our egress access control system for the Web.

A. The Simple Collaborative Method Among Web Browsers, Proxy Servers, and Authentication Servers

As described in Section III, conventional collaborative methods have several problems with modern complex authentication mechanisms such as Shibboleth and OAuth. To recap: in SPNEGO, Web browsers and proxy servers must communicate in an authentication mechanism-specific way, which means defining a new protocol is required if we want to add a new authentication mechanism. This requires developing both a Web browser extension and a Web proxy extension, which of course requires a lot of effort.

Our collaborative method solves these problems. It has several advantages over conventional methods, as we described in Section III, and briefly recap here. First, since we use a generic method, it is not necessary to define a protocol to deliver user attributes or access tokens for each authentication mechanism. This also means that we do not have to implement SPNEGO modules in both Web browsers and proxy servers. Second, we can add a new authentication mechanism to the system simply by deploying an authentication server of the

new authentication mechanism. Third, it is simple to verify user attributes in proxy servers. Finally, our method can easily allow a single user to log in with multiple authentication mechanisms at the same time.

Our method requires the use of authentication mechanism-independent user identifiers. In Sections III-A and IV-A, we discussed the requirements and implementations of such user identifiers. The current implementation running in our university library uses IP addresses as user identifiers. As discussed in Section IV-A, using IP addresses has several limitations. We also show ideal implementation methods of authentication mechanism-independent user identifiers by using HTTP headers.

As discussed in Section III-B, we have to isolate the user attribute store from Web browsers. We can perform this isolation with regular ways such as using packet filters and SSL certificates.

B. Application in our university library

We applied our method to an egress access control system for the Web in our university library. This library has 182 kiosk terminals that automatically run Web browsers in Windows. With kiosk terminals, visitors are allowed to access OPAC Web sites, library information pages, bus timetables, e-journals, and so on without user authentication. University students and staff are allowed to access any external page with user authentication through Shibboleth.

The egress access control system consists of a proxy server, a user attribute store, and an authentication server of Shibboleth. Web browsers running on the kiosk terminals are configured to use the proxy server. The authentication server is connected to the common identity provider of our university. Since this common identity provider is shared with other major services, including the e-learning system Moodle, students enjoy the single-sign-on feature of Shibboleth.

This egress access control system has been operational since May 2011. In this study, we analyzed the system's access logs. Fig. 10 shows the request rates on January 12, 2012, the day the maximum number of accesses per second was recorded. The X-axis is the time of day and the Y-axis is the number of accesses per second.

We ran the proxy server, the tuple space server of the user attribute store, and the authentication server in a single physical machine. This machine had a Xeon X5570 processor, 2 GB of memory, and a 1000Base-T network card. Its operating system was Linux CentOS 5.8. All the kiosk terminals and the server machine were connected to a gigabit network.

The system's performance was efficient to serve 182 kiosk terminals with a single physical machine. The peak access was 381 requests per second at 13:09:33. The total number of accesses on the day was 587,325, 76% of which were accesses with user authentication. The rest were accesses that matched patterns on the whitelist. The number of user authentication processes performed was 575. The whitelist had 712 patterns and was maintained by network administrators of the university library. A total 8,800 people used this access control system over the course of one year.

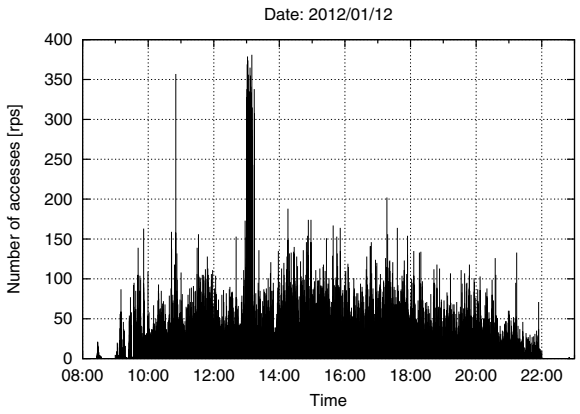


Fig. 10. Number of accesses per second on January 12, 2012. The peak was 381 requests per second.

C. Scalability of Our Egress Access Control System for the Web

In our collaboration method, we use a shared repository (the user attribute store) to share user attributes among proxy servers and authentication servers. This collaboration method allows the running of multiple proxy servers and multiple authentication servers as demand increases.

The scalability of the access control system is determined by the user attribute store. In the current implementation, we used Rinda, the tuple space implementation for the Ruby language. We also realized user attribute caching in the proxy server by using Rinda's update notification mechanism. This implementation could serve 381 requests per second (as described in Section V-B) with a single physical machine.

To serve more requests per second, we need a better key-value store than Rinda. CouchDB[5] and memcached[6] are replacement candidates. Since these key-value stores provide the append procedure, it would be possible to implement the `append_attr` procedure described in Section III-B.

VI. Conclusion

In this paper, we have described a new collaborative method for egress access control in Web proxy servers that enables us to use advanced user attributes of modern complex authentication mechanisms such as Shibboleth and OAuth. Our collaborative method simplifies communications among Web browsers, Web proxy servers, and authentication servers by using a trusted shared repository that stores user attributes. In our method, a new authentication mechanism can be added to the system by deploying an authentication server of the new authentication mechanism. Since an authentication server is a regular Web application, we can implement the applications with widely available libraries and tools. We have

[5]http://couchdb.apache.org/
[6]http://memcached.org/

implemented the authentication servers for Shibboleth and Facebook OAuth. They were simple CGI programs running on the Apache HTTP server. Their code sizes were 14 and 33 lines of code, respectively. Unlike in SPNEGO, neither Web browsers nor proxy servers are required to include extensions for modern complex authentication mechanisms.

Our method uses authentication mechanism-independent user identifiers that function as keys to access user attributes in the trusted shared repository. Web browsers send requests to authentication servers and proxy servers along with user identifiers. We have shown two ideal implementations that use HTTP headers and one practical implementation that uses IP addresses.

On the basis of our collaborative method, we have implemented the egress access control system for the Web in our university library. This system supports Shibboleth and has been fully operational for more than a year. The system's performance was good enough to serve 182 kiosk terminals with a single physical machine.

In future, we intend to implement better authentication mechanism-independent user identifiers, specifically, with user tracking techniques [14]. For example, if we rewrite HTML content in response messages and embed Web bugs in that content, we can identify individual users. We also plan to apply our method to a large system with a scalable key-value store.

REFERENCES

[1] "Internet2 Middleware Architecture Committee for Education(MACE) Directory Working Group," *http://middleware.internet2.edu/dir/*.

[2] G. Appenzeller, M. Roussopoulos, and M. Baker, "User-friendly access control for public network ports," in *IN IEEE INFOCOM*, 1998, pp. 699–707.

[3] Apple Inc., "Mac OS X v10.5, 10.6: About the Parental Controls Internet content filter," http://support.apple.com/kb/HT2900.

[4] L. Bauer, M. A. Schneider, and E. W. Felten, "A general and flexible access-control system for the web," in *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 93–108.

[5] D. W. Chadwick, A. Otenko, and E. Ball, "Role-based access control with x.509 attribute certificates," *IEEE Internet Computing*, vol. 7, no. 2, pp. 62–69, Mar. 2003.

[6] E. E. Hammer-Lahav, "The OAuth 1.0 Protocol," RFC 5849, 2010.

[7] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A role-based access control model and reference implementation within a corporate intranet," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, pp. 34–64, Feb. 1999.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, 1999.

[9] R. T. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, pp. 88–90, 1997.

[10] M. Foundation, "Block and unblock websites with parental controls," http://support.mozilla.org/en-US/kb/block-and-unblock-websites-with-parental-controls.

[11] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617, 1999.

[12] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.

[13] E. Glass, "The NTLM Authentication Protocol and Security Support Provider," 2003, http://davenport.sourceforge.net/ntlm.html.

[14] W. T. Harding, A. J. Reed, and R. L. Gray, "Cookies and web bugs: What they are and how they work together." *Information Systems Management*, vol. 18, no. 3, p. 17, 2001.

[15] K. Jaganathan, L. Zhu, and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows," RFC 4559, 2006.

[16] O. Kornievskaia, P. Honeyman, B. Doster, and K. Coffman, "Kerberized credential translation: a solution to web access control," in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01, 2001.

[17] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC 2109, 1997.

[18] J. Linn, "Generic Security Service Application Program Interface," RFC 2743, 2000.

[19] T. Narten and R. Draves, "Privacy Extensions for Stateless Address Autoconfiguration in IPv6," RFC 3041, 2001.

[20] OASIS Standard, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0," 2005, http://docs.oasisopen.org/security/saml/v2.0/saml-core-2.0-os.pdf Accessed: 2012/06/18.

[21] P. P. Pal and M. Atighetchi, "Supporting safe content-inspection of web traffic," in *The Journal of Defense Software Engineering*, 2008, pp. 19–23.

[22] J. S. Park, R. Sandhu, and G.-J. Ahn, "Role-based access control on the web," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 1, pp. 37–71, Feb. 2001.

[23] M. Seki, "dRuby and Rinda: Implementation and Application of Distributed Ruby and its Parallel Coordination Mechanism," *International Journal of Parallel Programming*, vol. 37, no. 1, pp. 37–57, 2009.

[24] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *proceedings of the Winter 1988 Usenix Conference*, 1988.

[25] S. Suzuki, Y. Shinjo, T. Hirotsu, K. Kato, and K. Itano, "Name-level approach for egress network access control," in *Networking - ICN*, 2005, vol. 3421, pp. 284–296.

[26] K. Takaaki, S. Hiroaki, D. Noritoshi, and M. Ken, "Design and Implementation of Web Forward Proxy with Shibboleth Authentication," in *Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on*, july 2011, pp. 321–326.

[27] K. Watanabe, M. Otani, S. Tadaki, and Y. Watanabe, "Opengate on cloud," *International Conference on Advanced Information Networking and Applications Workshops*, pp. 1027–1030, 2012.

[28] W. Xu, D. W. Chadwick, and S. Otenko, "Development of a Flexible PERMIS Authorisation Module for Shibboleth and Apache Server." in *EuroPKI*, ser. Lecture Notes in Computer Science, vol. 3545. Springer, 2005, pp. 162–179.

[29] L. Zhu, P. Leach, K. Jaganathan, and W. Ingersoll, "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism," RFC 4178, 2005.