Collaboration by Passing Access Rights for Personal Protected Web Resources

Yasushi Shinjo, Daisuke Kamikawa, Akira Sato
Department of Computer Science
University of Tsukuba
1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan

***Abstract*—This paper describes how users can collaborate through sharing personal protected Web resources. Personal protected Web resources are Web pages and services that are typically password-protected. One example is a personal page on an auction site. This paper introduces capability-based access control to the World Wide Web without modifying existing servers and clients. Access rights for personal protected Web resources are represented as capabilities for the Web resources. When users collaborate, capability-based access control on the Web has two advantages over conventional access-control-list based access control. First, a user can easily pass his/her own capabilities to access Web resources to other users along with delegating tasks. For example, a parent can ask a child to bid on a PC on behalf of the parent by passing the capability to access the parent's auction page but not giving the child the password. Second, restricted capabilities are useful in passing access rights. For example, before a parent passes the capability to bid on a PC to a child, the parent can create a restricted capability that allows biding up to $100 on a PC from the original unlimited capability. The proposed method has been implemented as Web applications called CapaEdit and CapaGate in Java by using the Google Web Toolkit. Using CapaEdit, a user can interactively create a capability to access his/her personal protected Web resources with access control to hyperlinks and form parameters. The receiver of the capability can access the Web resources through CapaGate, which enforces the restrictions. Experimental results show that these Web applications perform well enough for interactive use.**

I. Introduction

While the Internet contains a huge number of publicly accessible Web resources, people have their personal protected Web resources. *Personal protected Web resources* are Web pages and services that can be accessed only by a specific person or people. Examples of such resources include personal pages on auction sites, personal schedules, and personal photos on social network service (SNS) sites. To create personal protected Web pages, most Web servers adopt an access control model based on *access control lists* (ACLs). ACLs are stored in objects and contain subjects (principals) and their permitted operations. In ACL-based access control, before the access control mechanisms work, subjects must be authenticated, typically by using usernames and passwords.

While people use personal protected Web resources, people sometimes want to allow other persons to temporarily access these Web resources. For example, a user sometimes wants to show his/her personal photo albums to friends at home. In an office, a project leader sometimes must show confidential Web pages to a project member who does not have permissions to see these pages. In such cases, it is common practice for a person to access the protected Web resources on his/her PC and show the PC's display to others when they sit together. However, when the resource owner and the others are separated by distance and time, it become harder to allow limited access to the resource while keeping privacy and confidentiality. Specifically, ACL-based access control has the following problems. First, it is hard for a user to delegate his/her access rights to other users. For example, if a project leader can access a confidential Web page but does not own it, the leader cannot pass his/her access rights to other project members. Second, user registration is necessary, so if a temporary worker is not allowed to have an account, the temporary worker cannot get access right to the confidential page. Third, most existing Web servers provide poor access control mechanisms about lengths of time something is accessible and the maximum number of times it can be used. For example, we cannot limit the number of uses to 10. If we wish to allow users to access a Web page within a day, we must change the ACL of that Web pager after one day.

We are tackling these problems of ACL-based access control on the Web. We use capability-based access control to solve these problems because it has an advantage over ACL-based one in terms of access rights delegation [13] [18]. We have implemented capability-based access control in the Web without modifying existing Web servers and browsers in the Web application CaStor [14]. CaStor stands for capability store. In CaStor, a user can create a capability to access his/her password-protected Web pages and send it to another user. A receiver of the capability can access the Web pages without knowing the password. The receiver also can redistribute the capability to the third user. A capability holder can create a restricted capability from the original capability. Restrictions include the maximum number of uses, expiration dates, and allowed URL patterns in regular expressions.

However, CaStor-realized capability-based access control on the Web has the following problems. First, CaStor cannot deal with the Web resources that are created with the PUT method in Hypertext Transfer Protocol (HTTP). Second, CaStor cannot restrict HTTP parameters that are sent on the basis of HTML forms. Third, CaStor provides poor user interface to configure permitted hyperlinks and forms. Because of these problems, we cannot, for example, create a capability to bid on an item on an auction site up to $100.

COLLABORATECOM 2010, October 9-12, Chicago, USA

DOI 10.4108/icst.collaboratecom.2010.50

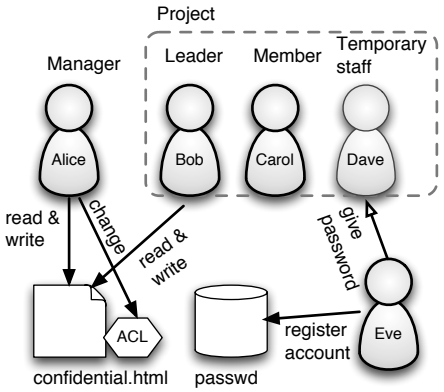

Fig. 1. Colleagues collaborating using a confidential Web resource.

To solve these problems in CaStor, we have implemented the second-generation tool to realize capability-based access control on the Web. The tool consists of two Web applications: CapaEdit and CapaGate. CapaEdit is an authoring tool to make capabilities based on personal Web resources protected with ACLs. CapaGate is a proxy to enforce the restrictions of capabilities. In CapaEdit and CapaGate, we can create a capability to bid on an item on an auction site that uses the POST method with a maximum price limit. Using these programs, we can use advantages of capability-based access control on the Web, and users can collaborate by passing capabilities among themselves.

The rest of the paper is organized as follows. Section II describes capability-based access control on the Web. Section III and Section IV explain CaStor and CapaEdit plus CapaGate, our first and second implementations of the capability-based access control in the Web. Section V describes the performance of CapaEdit and CapaGate. Section VI presents related work, and Section VII summarizes the key points and takes a look at future work.

II. Capability-based access control in the World Wide Web

In this section, we first describe the problems of ACL-based access control in the World Wide Web. Next, we explain the solutions by introducing capability-based access control to the Web.

A. The problems of ACL-based access control in the Web

To protect Web resources, most Web servers adopt an access control model based on ACLs. In ACL-based access control, policies are stored in target objects, that is, Web resources. An access control policy is described as a list of subjects (principals) and their permitted operations. For example, a list can say that Alice is allowed to read and write the resource but Bob is allowed to only read the resource. In many widely used Web servers, such ACLs can be modified only by owners of resources or system administrators. In ACL-based access control, subjects should be authenticated before the access control mechanisms work. To authenticate users, most systems use usernames and passwords, public keys in public key infrastructure (PKI), biometrics, etc.

However, ACL-based access control in the Web has the following problems.

1) Regular users cannot pass their access rights for Web resources to other users. Changing access control policies, which include sensitive information, increases the workload of the owners of Web resources or system administrators.
2) User registration is required for user authentication. If a user is not allowed to have an account on a Web server, the user cannot use Web resources on the Web server.
3) Systems provide inadequate support for describing access control policies that use time constraints.

We describe these problems by using an example in an office. In this example, the following people work together, as shown in Figure 1. Alice is the manager. Bob is the project leader. Carol is a project member. Dave is a temporary member of staff who joins the project for a month. Eve is a system administrator who performs user registration. Alice owns the Web resource "confidential.html", which is protected with an ACL. This ACL allows Alice and Bob read and write access.

Bob and Carol are working together. Bob's work involves accessing the Web resource, confidential.html. When busy, he wishes to temporarily delegate the task that requires reading confidential.html for one week to Carol. However, he cannot give the read permission because he is not the owner of the Web resource. He needs to ask Alice to give the read permission to Carol. Alice must change the ACL of confidential.html for the temporary delegation. When Carol finishes the task that requires reading confidential.html, Bob asks Alice to again change the ACL of confidential.html, so Alice must restore the ACL. Since the ACL contains sensitive information, changing the ACL increases the workload of the owner of the Web resource. In ACL-based access control, Bob cannot help Alice.

To accelerate the project completion, Bob hires a temporary member of staff, Dave. Bob wants to allow Dave to read confidential.html, just like Carol. To achieve this, Dave needs a user account on the Web server. Sometimes, the system management policy prohibits Dave from having an account. In Web server Apache httpd [8], for example, Web resources can be protected with the statement `require valid-user` in the file .htaccess. This means that any user who has a valid account can access the Web resources. If the system includes Web resources with this statement, Dave is often not allowed to have an account.

B. Solving the problem of ACL-based access control with capabilities

To solve the problems of ACL-based access control, we introduce capability-based access control to the World Wide Web. Conceptually, a capability consists of an identifier of an object and a set of access rights for that object. A capability represents a self-authenticating permission to access a specified object in specified ways. It is like a ticket or door key: possession of a capability is proof of the holder's rights to

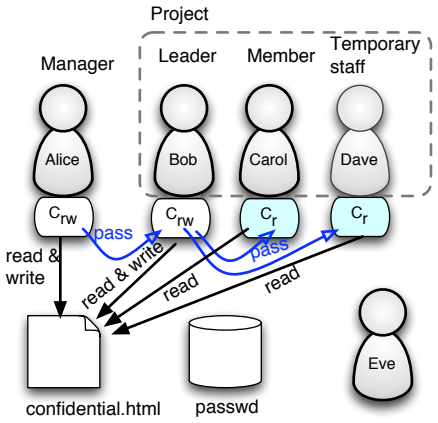

Fig. 2. Capability-based access control in the World Wide Web.

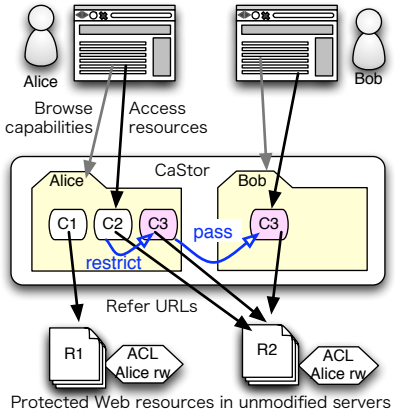

Fig. 3. The CaStor Web Application.

access an object. Without a capability for an object, a user cannot access it. We realize capability-based access control to protect Web resources.

Capability-based access control has an advantage over ACL-based one in terms of access rights delegation. Users can pass their holding capabilities to other users and delegate their tasks. At this time, the owners of objects have to do nothing. When passing capabilities, capability restriction is useful. *A restricted capability* is a capability that refers to the same object as a base capability but has fewer permitted operations than the base capability. For example, if Alice has a read and write capability for a Web resource, she can create a read-only capability for the Web resource.

The problems in Section II-A are solved with capabilities as follows (Figure 2).

1) Access rights delegation problem: Alice creates a read-write capability for confidential.html and passes it to Bob. When he becomes busy, he wishes to temporarily delegate the task of reading it to Carol for one week with read access to confidential.html. To achieve this, Bob has to do two things. First, based on the read-write capability, he creates a read-only capability for confidential.html that is valid for one week. Second, he passes the restricted capability to Carol. Carol then performs the task using the restricted capability for confidential.html for one week. Alice does not have to do anything in this task delegation.

2) User registration problem: To accelerate the project completion, Bob hires a temporary member of staff, Dave. Bob wishes to allow Dave to read confidential.html, just like Carol. To achieve this, Bob has to do same things as in Carol's case: he creates a restricted capability and passes it to Dave. At this time, if he passes the restricted capability with a file in a USB drive or by using any method that does not require user registration, Dave does not have to have a user account. Alice, the owner of the Web resource, and Eve, the system administrator, do not have to do anything in this task delegation.

3) Time-related access control problem: When Bob creates a restricted capability for Carol, he restricts the valid period of the capability to one week. Alice does not have to do anything in this task delegation.

III. CaStor

As described in Section II, introducing capability-based access control solves the problem of ACL-based access control on the World Wide Web. However, most current Web servers adopt ACL-based access control. To clarify the benefits of capability-based access control on the current Web, we have implemented a Web application named CaStor. CaStor stands for capability store. In CaStor, a user can create a capability for a protected Web resource without modifying any existing Web servers and browsers. A user can also distribute capabilities to other users. Capability Basket is another tool for capability distribution running on a personal computer.

A. The design of CaStor

We have designed CaStor to achieve the following goals.

- Realize capability-based access control in the Web without modifying existing servers and clients.
- Realize restricted capabilities.
- Enable users to manage capabilities safely.

CaStor provides the following operations of capabilities for users.

1) Creating a base capability.
2) Creating a restricted capability.
3) Managing Capabilities.

A user creates a capability for accessing his/her own Web resource in an unmodified Web server. We call this type of a capability *a base capability*. To create a base capability, a user sends CaStor the URL, the username and password. While a base capability holds a username and password, it does not expose them to capability holders. A base capability can include a script to log into the Web server. In Figure 3, Alice creates two capabilities, C1 and C2, for accessing her own Web resources, R1 and R2, in unmodified Web servers.

A user can create a restricted capability from a base capability. Restrictions include the maximum number of

uses, expiration dates, and allowed URL patterns in regular expressions. For example, a user can give a regular expression `^/dir1/(file2|file3)\.html$` to CaStor. In this case, CaStor relays the GET request `/dir1/file2.html`, while CaStor does not relay the GET request `/dir1/file4.html`. Restricted capabilities can be created from other restricted capabilities. For example, if a user has a restricted capability that is valid from July 1 to December 31, 2010, the user can create a restricted capability that is valid from October 1 to October 31, 2010. In Figure 3, Alice creates a read-only capability, C3, from the read-write capability, C2.

A user can organize holding capabilities with hieratical directories and send these holding capabilities to other users. In Figure 3, Alice has a directory that includes three capabilities and sends the capability C3 to Bob's directory.

The CaStor Web application consists of two components: the CaStor site and the CaStor proxy. The CaStor site manages directories, and the CaStor proxy enforces restrictions. Before a user uses a capability, the user logs into the CaStor site, browses directories, and finds the target capability. In the CaStor site, the target capability is represented as a URL to the CaStor proxy. A URL to the CaStor proxy includes a random number to protect the capability.

When the user clicks the hyperlink with a URL to the CaStor proxy in a Web browser, the request is sent to the CaStor proxy. The CaStor proxy extracts the random number in the URL and retrieves pieces of information to access the original Web resource. These peaces of information include the URL of the original Web resource, a username, a password, and a login script. Using these peaces of information, the CaStor proxy first logs into the original Web resource with the username, password, and login script, and obtains session cookies. Next, the CaStor proxy obtains the target contents usually in HTML by sending the URL and these session cookies. Finally, the CaStor proxy returns the contents to the Web browser.

The CaStor site is written in the Ruby language based on the framework Ruby on Rails. The CaStor proxy is written in the Java language.

B. Problems of CaStor in Collaboration

In CaStor, we have realized capability-based access control on the World Wide Web. We can create capabilities to access protected Web resources without modifying existing servers and clients. We can also create restricted capabilities and pass them to other users with whom we are collaborating. However, the following problems remain.

1) CaStor cannot handle the POST method. In an HTML, both the GET and POST methods are used to send data from a Web browser to a Web server. The POST method is common in Web resources with dynamic page generation. If a Web site uses a form of the POST method, the CaStor cannot create a capability to access the Web resource.

2) CaStor is limited by the expressive power of regular expressions. It is hard for casual users to describe desired restrictions in a regular expression. If there are two or more `<input>` parameters, a user has to specify them in several orders. For example, if a form has two parameters `<input name="a">` and `<input name="b">`, in a regular expression, we have to write both `a=...&b=....` and `b=...&a=....`.

3) CaStor provides a poor user interface as an authoring tool. When a user creates a base capability from an unmodified protected Web resource, the user has to give necessary pieces of information in a Web page in CaStor, but the user cannot see the unmodified protected Web resource. It is not simple to specify allowed hyperlinks in a Web resource.

C. Capability Basket

In CaStor, a capability is exported as a URL to a Web page in the CaStor proxy with a random number for protection. A URL can be either temporary or long lasting. A temporary URL expires when a user logged out from the CaStor site. A long lasting URL can be passed to other users with various methods, including encrypted e-mail.

Capability Basket is a capability distribution tool running on a personal computer. It is designed to distribute capabilities for e-mail inboxes to solve the false positive problem of spam filters [21]. It can handle capabilities for Web resources as well as those of e-mail inboxes. In Capability Basket, users can send capabilities by using the overlay network with the instant messenger Skype.

IV. CapaEdit and CapaGate

In Sections II and III, we introduced capability-based access control on the World Wide Web and described its implementation in the CaStor Web application. Although capability-based access control is useful, its implementation of CaStor has several problems: no support of the POST method, less expressive power for regular expression, and being a poor authoring tool. To solve these problems, we have implemented Web applications CapaEdit and CapaGate, which mainly replaces the function of the CaStor proxy. CapaEdit is an authoring tool that enables users to add filters for forms and hyperlinks to existing personal protected Web resources. CapaGate is a proxy that enforces the filtering rules specified with CapaEdit.

A. Filters to hyperlinks and form parameters

In CapaEdit and CapaGate, we deal with the following protected Web resources in unmodified servers, as shown in the bottom of Figure 4.

- A capability allows users to access one or a group of personal Web resources that are protected with a username and password. A capability can have a login script as an attribute.
- Web resources can be either static HTML files, images, or dynamically generated ones in servers. An HTML page can include simple JavaScript programs. See Section IV-F for details.
- An HTML page can include hyperlinks to other internal protected Web resources. *An internal protected Web resource* means a Web resource that can be accessed

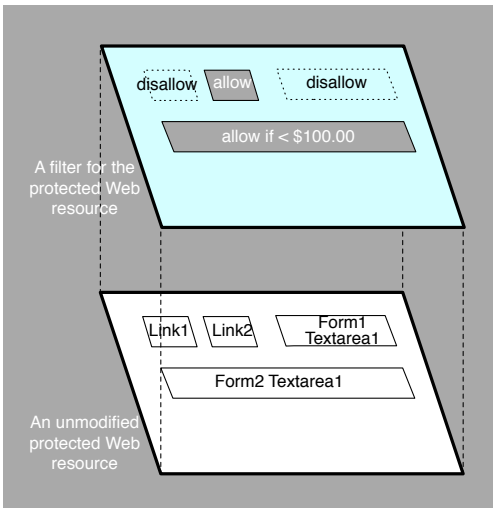

Fig. 4. Filters to hyperlinks and form parameters in CapaEdit and CapaGate.

Fig. 5. Adding a filter to a text box in CapaEdit.

with the same session cookies and needs no extra user authentication.

- An HTML page can include multiple forms. Each form can contain several input fields, such as text boxes, text areas, radio buttons, submit buttons, and hidden fields. Both the GET and POST method are allowed.

In an HTML page, texts and in-line images are passed, and all hyperlinks and submit buttons to internal protected Web resources are disabled by default. A user can add the following filters to hyperlinks and form parameters, as shown in the upper layer of Figure 4.

- A user can enable a hyperlink to an internal protected Web resource.
- A user can enable a submit button to an internal protected Web resource.
- In a form, a user can add a filter to each input parameter. For example, to a text box, a user can add a filter that allows the form submission only if the value of the text box is less than or equal to 100 as an integer.
- In addition to parameter range checking, a user also overwrites a input parameter with arbitrary values. For example, to a text box, a user can replace the value of a text box with the string “hello”.

B. CapaEdit

CapaEdit is an interactive authoring tool that enables a user to create a capability to access personal protected Web resource (Figure 5). As described in Section III-B, the previous Web application CaStor has three problems. CapaEdit along with CapaGate solves these problems.

At first, a user of CapaEdit gives a URL of protected Web resources to CapaEdit. CapaEdit shows the same content of the protected Web resource in the same style with small exceptions. An exception is the URL that appears in the Web browser, which is a URL in CapaEdit. Another exception is added context menus for hyperlinks, form submit buttons, and form input parameters. In Figure 5, for example, a user is adding a filter to a text box on the auction site, eBay.com. As shown in this figure, the page layout is the same as the original one on eBay.com. By showing a context menu of the text box, the user can add a filter to the text box. When a user moves the mouse pointer over a hyperlink, the user notices this fact because the shape of the pointer cursor changes. At this time, the user can enable the hyperlink with a context menu. Finally, the user obtains *an external representation* of a capability for the protected Web resource as a URL to CapaGate. We will show the format of external representation of capabilities in Section IV-E.

Since CapaEdit keeps the layout of unmodified protected Web resources, a casual user can easily use CapaEdit. This means that the user interface problem of CaStor has been solved. CapaEdit also can add filters to the forms that submit requests with the POST method. At runtime, CapaGate enforces these filters. This means that the POST method problem of CaStor has been solved. In CapaEdit, a user can add filters for input parameters and enable hyperlinks without regular expressions. This means that the regular expression problem of CaStor has been solved.

When a user finishes adding filters, CapaEdit saves the following pieces of information that are associated with a capability.

- The URL of the start page and the following allowed-to-be-seen internal pages.
- The list of allowed forms.
- Filters for form input parameters.
- A username, a password, and a login script.
- The maximum number of uses
- Expiration dates

These pieces of information are also associated with the external representation of the capability.

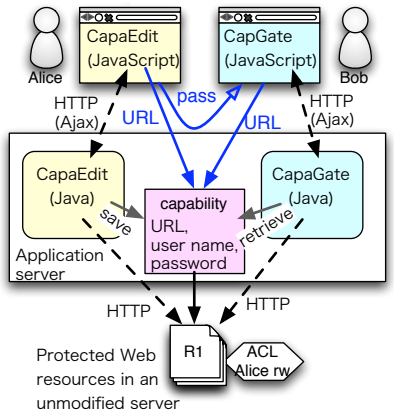

Fig. 6. Authoring and accessing protected Web resource through CapaEdit and CapaGate.

C. CapaGate

CapaGate is a Web application that runs in conjunction with CapaEdit (Figure 6). When CapaGate receives an external representation of a capability as a URL from a browser, CapaGate performs as follows.

1) CapaGate extracts a random number for protection.
2) By using this random number, CapaGate retrieves the saved information and validates the capability.
3) CapaGate reads the necessary information to access the unmodified protected Web resources.
4) CapaGate logs into the unmodified protected Web server with the saved username, password, and login script.
5) CapaGate obtains the start page and returns its contents to the client.

The user will see the start page in the unmodified protected Web resources.

After that, CapaGate receives all the click events in hyperlinks and submit buttons. If they are not enabled by the capability, CapaGate does nothing. If they are enabled, CapaGate performs appropriate actions. For a hyperlink, CapaGate goes to the next page. For a submit button, CapaGate checks input parameters to see if they are allowed by the filter. If they are allowed, CapaGate submits the form to the unmodified Web server. Otherwise, CapaGate shows an error page.

D. A usage example

Consider that a parent has protected Web resources on the auction site eBay.com and wants a child to bid on a PC on his or her behalf. It is a bad idea for the parent to pass his or her username and password to a child. If the parent does this, the child can access not only the target Web resource in question but also other Web resources. This is potentially very dangerous and usually unacceptable. When the parent delegates the bidding task to the child, the parent may also wish to limit the maximum price.

To achieve this, the parent first connects to CapaEdit. In CapaEdit, the parent browses the auction site. When the parent finds the target item, the parent enables a form as follows.

```
<form method="post"
  action="http://offer.ebay.com/eBayISAPI?
  MakeBid&item=280525128165">
<input type="text" name="maxbid">
<input value="Place bid" type="submit">
</form>
```

In CapaEdit, the parent enables the submit button `<input type="submit">`. Next, the parent adds a filter to the text box `<input name="maxbid">`, as shown in Figure 5. The parent sets the maximum value of the parameter `maxbid` to 100 as an integer. Finally, the parent obtains the external representation of the capability and passes it to the child.

E. Security Issues

The host of Web applications CapaEdit and CapaGate can be as safe as regular Web sites that hold protected Web recourses. Even if a user holds a capability for a protected Web resource, the user cannot obtain secret information about the capability. For example, the user cannot obtain the password or login script of the capability.

In CapaEdit, a user can export a capability as an external representation. This is a URL to CapaGate that includes a random number for protection. The format of this URL follows.

`http://host/capa/gate/`*random*

In this format, *random* is a random number to protect a capability. Since it is hard for a non-holder of a capability to estimate the URL, the Web resource described by the URL is protected from attackers. This protection method of capabilities is called *a password-capability model* [2] [4]. In this model, the integrity of a capability is ensured through the use of sparse random numbers (called passwords) in an astronomically large space. This model is also used in other capability-based systems [19] [9].

Since the safety of CapaEdit and CapaGate depends on the safety of exported URLs with random numbers, the users of CapaEdit and CapaGate must be able to handle capabilities. First, the users should keep exported capabilities secret and use secure channels to distribute capabilities to other users. If e-mail is not considered secure, users should not use it to send exported capabilities. We can protect HTTP communications between a browser and the host of CapaEdit and CapaGate with Secure Socket Layer (SSL) [1]. In CaStor, users can exchange capabilities within a site, and temporary exported capabilities expire automatically when users logout. In Castor and CapaEdit, exported capabilities can be distributed with Capability Basket in a safe way because communications in Capability Basket are automatically encrypted by the instant messenger Skype.

Second, the users of CapaEdit and CapaGate should pass their own capabilities to carefully selected people. This is

[1] In many Web sites including eBay, Facebook, and Twitter, SSL is used only at login to protect usernames and passwords. When the user authentication succeeds, a server typically sends back session cookies. Clients send these session cookies over regular HTTP without SSL.

similar to giving a telephone number to others. People carefully give their telephone numbers to friends. Similarly, people should carefully pass their capabilities to friends.

CapaGate enables accessing personal protected Web resources with limited URLs and prohibits accesses to other resources that can be accessed through predictable URLs. Consider that a base Web resource in HTML includes predictable hyperlinks. For example, a base HTML file `index.html` includes two hyperlinks: `<A href="file1.html">` and `<A href="file2.html">`. In CapaEdit, consider the case when a capability sender allows the former hyperlink and disallows the latter hyperlink. The receiver of the capability can easily guess there is a hyperlink `<A href="file2.html">`.

In this case, CapaEdit saves a list of allowed URLs with the capability. This list includes the URLs `index.html` and `file1.html` but not `file2.html`. Therefore, CapaGate allows the access to the Web resources `index.html` and `file1.html` and blocks the access to the Web resource `file2.html`.

Although CapaGate allows accessing personal protected Web resources with limited URLs and form parameters, privacy information can unintendedly leak through the allowed Web resource. For example, if a base Web resource includes advertisements that are generated from its owner's preferences, the capability receivers who access the page through CapaGate can guess those preferences from the advertisements. We plan to extend CapaEdit and CapaGate that remove such subcomponents from an HTML resource.

It is hard for capability-based systems to realize accountability [18], meaning determining who delegated access to a particular user, at least as part of an auditing (forensics) process. Restricted capabilities realize a degree of accountability in CapaEdit and CapaGate. For example, consider the case when Alice wishes to pass her capability to two colleagues, Bob and Carol. First, Alice creates two restricted capabilities based on the original capability. These restricted capabilities identify the same Web resource as the original one and have the same access rights in this case. Second, Alice passes one restricted capability to Bob and the other to Carol instead of passing the original capability, which Alice continues to use it. At this time, all three users (Alice, Bob, and Carol) use individual capabilities. When an incident occurs, the problematic capability is identified. If the capability is daily used by Bob, Bob must deal with the incident.

F. Implementation of CapaEdit and CapaGate

Both CapaEdit and CapaGate are Ajax (Asynchronous JavaScript and XML) Web applications based on the framework Google Web Toolkit (GWT). In GWT, developers write code in Java. A part of the written Java code runs as Java Servlets in a server. Another is translated into JavaScript code and HTML, and executed in a browser.

In the design of CapaEdit and CapaGate, we try to keep the Web resource design unchanged. Specifically, we do not add buttons and frames to a base Web resource, and we use style sheets the same as those in the base Web resource. Most functions of CapaEdit are available through added context menus and pop-up windows.

Since CapaEdit and CapaGate are Ajax applications, there is a limit to using JavaScript in the base protected Web resources. If the JavaScript code of CapaEdit and CapaGate conflict with that of the base protected Web resource, they do not work. For example, Ajax programs of Google including Gmail and Google Docs do not work in CapaEdit and CapaGate. Since CapaEdit adds items to context menus, these items can interfere with the base ones. In addition, the current implementation prohibits the use of frames.

CapaEdit and CapaGate support the JavaScript code that replaces a part of a HTML document or a tree structure in Document Object Model (DOM). For example, CapaEdit is compatible with the JavaScript code that changes the style sheet to show a login window.

CapaEdit runs on Web sites including ebay.com, facebook.com, and twitter.com.

V. EXPERIMENTS

If the performance of an access control mechanism is not good, no one will use it. To show the practicality of CapaEdit and CapaGate, we performed experiments.

Since both CapaEdit and CapaGate are Ajax applications, they perform well while they run without communication to backend servers. Therefore, in our experiments, we focus on the cases when these applications perform communications to backend servers.

A. Experimental setup

We measured the performance of CapaEdit and CapaGate by using the environment shown in Figure 7 and Table I . This environment consists of the local site and the remote site. The local site has two machines: the client PC and the server host. The client PC runs the Web browser Firefox along with the Web testing tool Watir. The server host runs the Web application server GlassFish. We use www.ebay.com as the common remote Web site that provides unmodified protected Web resources.

CapaEdit and CapaGate do not enforce accesses to external resources such as style sheets, images, analytics, or

TABLE I
HARDWARE SPECIFICATIONS AND SOFTWARE VERSIONS OF THE EXPERIMENTAL ENVIRONMENT.

Item	Specification or version
Client PC	Apple MacBook Pro 17inch with CPU Intel Core 2 Duo 2.93 GHz, Memory 8GB, Solid State Drive (SSD), OS MacOSX 10.5.8
Firefox	3.6.8
Watir	firewatir 1.6.5 and jssh 0.9
HttpFox	0.8.7
Server Host	A PC with Intel Core i7, Memory 6GB, OS Linux Kernel 2.6.27 SMP, 64bit, Ubuntu 8.10
GlassFish	3.0 running in NetBeans 6.9 and Java 1.6.0
LAN	Gigabit Ethernet
Remote Base Site	www.ebay.com

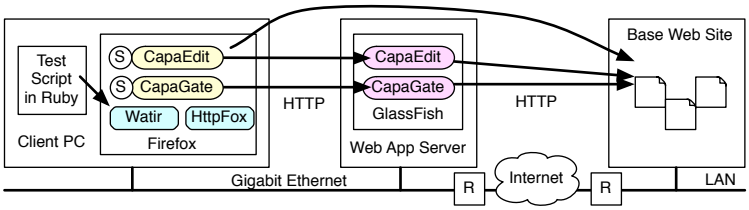

Fig. 7. The network configuration of the experimental environment.

JavaScript code. This is an intentional behavior for performance. For example, www.ebay.com has external servers {include,p,q,rtm}.ebaystatic.com to hold such resources. If an HTML resource includes such external resources, they were transferred from original servers to the browser without being checked by CapaGate. Such external resources are static and publicly accessible. In CapaEdit and CapaGate, transfer times of external resources are the same as those in the direct access case.

Using the Web testing tool Watir, we can repeat the same Web browser operations. For example, we can repeat operations: click the hyperlink, fill the text box with a parameter, click the submit button, etc.

CapaEdit and CapaGate consist of the JavaScript code and Java code. The JavaScript code runs in the Web browser, and the Java code runs on the Web application server. Since the JavaScript code and the Java code communicate with each other in an Ajax style, it is not simple to measure the execution times. Although the Web testing tool Watir provides a simple synchronization mechanism, this is not enough to measure the execution times because of the asynchronous behavior of the JavaScript code. To overcome this problem, we used the extension HttpFox of Firefox. With HttpFox, we can record the communication logs that include the start times, elapsed times, and URLs. A typical Web resource consists of not only the main HTML resource but also in-line images, style sheets, and JavaScript programs. In a sequence of HTTP requests, we measured browser load times on the basis of HTTP communication times:

- The start time of loading the main HTML resource.
- The finish time of loading the last non-HTML resource.

We repeated the same experiments more than ten times by stopping and restarting the Web browser to clear caches and obtained the minimum browser load times. Using the minimum times can remove the fluctuation of communication delays between the local site and the remote base site.

B. CapaEdit

Figure 8 shows the browser load times with and without the Web application CapaEdit. As a remote base Web resource, we used an auction item page in www.ebay.com.

In a direct access case, loading of the HTML file, style sheets, and images was finished in 1.14 seconds. With CapaEdit, the same page was loaded in 1.73 seconds. CapaEdit added 0.59 seconds in a Gigabit Ethernet environment. This

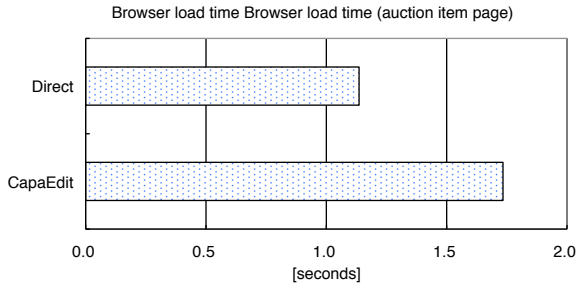

Fig. 8. The browser load times of a CapaEdit case and a direct access for an auction item page.

delay is acceptable for daily Web browsing on the Internet. We must emphasize that the JavaScript part of CapaEdit runs on the Web browser. This part responds promptly in the interactive use without communication.

C. CapaGate

Figure 9 shows the browser load times with and without the Web application CapaGate. We used the same auction page as in Section V-B with authorization. In CapaGate, we accessed the page with the exported URL by CapaEdit as described in Section IV. In the direct access case, we performed normal operations: viewing the auction item page, clicking the sign-in link, typing the username and password, and clicking the sign-in button. After signing-in, the first auction item page was shown again.

As shown in Figure 9, the browser load times of CapaGate and direct access cases were almost the same. In CapaGate, the signing-in process is done by the Java code in the server, and the browser showed only the single page after authorization with the capability. On the other hand, in the direct access case, the browser showed three Web pages with images. The first and third pages shared many images and style sheets. As a result, the browser load times of both cases were the same.

Next, we show the execution times of bidding. If the bidding is not allowed, this is quickly blocked by CapaGate within a LAN environment, as described in Section IV. Therefore we measured the bidding time in an allowed case.

Figure 10 shows the browser load times with and without the Web application CapaGate for an auction bidding page. In CapaGate, the maximum price was set to $0. In both cases, we bid $0 on an item, and this caused an error. In a direct

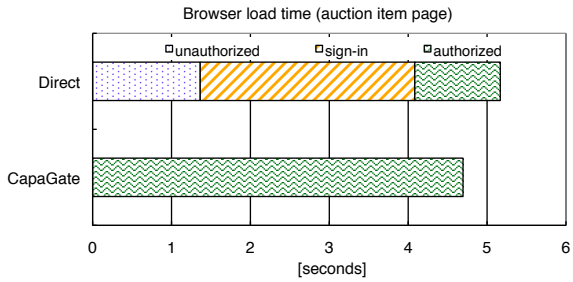

Fig. 9. The browser load times of a CapaGate case and a direct access case for an auction item page.

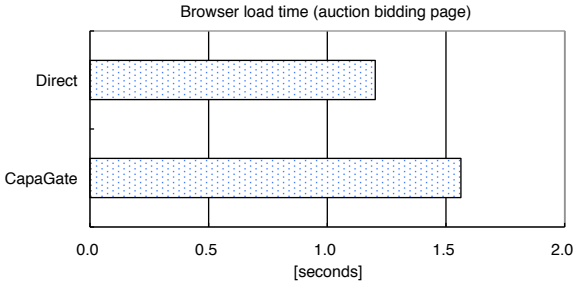

Fig. 10. The browser load times of a CapaGate case and a direct access case for an auction bidding page.

access case, loading of the HTML file, style sheets, and images was finished in 1.20 seconds. With CapaGate, the same page was loaded in 1.56 seconds. This delay is acceptable when compared with daily Web browsing on the Internet.

We compared the CapaEdit result in Figure 8 and the CapaGate result in Figure 10. Although these two graphs show the same tendency, CapaGate was faster than CapaEdit. This is because the auction bidding page included a smaller number of hyperlinks and forms than the auction item page.

VI. Related Work

Capability-based access control was used in early multiprocessors and distributed operating systems. Hydra [26] uses capabilities as references for objects, Mach [1] uses capabilities to control access to communication ports, and Amoeba [19] enables user processes to pass capabilities using general interprocess communication. Symbian OS [7], a recently developed operating system for mobile phones, uses capabilities to protect resources in a single-user system. Capability-based access control is used not only in distributed operating systems but also global database and file systems. HomeViews [9] uses capabilities to access views of databases that are in remote PCs, and CapaFS [20] uses capabilities to access files that are in remote PCs. Our prior reports have described the implementation of capability-based access control in egress network access [22], in components of XML Web services [16], in e-mail spam filters [21], in wireless LAN access [17], and in networked devices [15]. In this paper, we describe the implementation of capability-based access control in regular Web resources.

In some distributed file systems, authorization certificates are used for passing access rights, and they resemble capabilities [18]. In WebFS, for example, if a user has access rights for files, the user can create a certificate for another user to pass on a subset of the access rights. An access right transfer certificate is usually digitally signed by the public key in the sender's certificate. Some systems enable transitive delegation by chaining certificates. To realize authorization certificates on the World Wide Web, we have to modify existing servers. Our implementations of capability-based access control require no modifications to existing servers.

Several access control models have been proposed in addition to ACL-based and capability-based models [23]. In role-based access control (RBAC), a user has multiple roles, each with its own access rights. In task-based access control (TBAC), each task step is bound to a task and access right. When a user starts working on a step in a task, he or she can use the access right bound to the task step. In a context-based model, access rights of a user are determined on the basis of context information, such as access times, places, and network conditions. In RBAC, TBAC, or a context-based model, a user cannot pass his or her role, task step, or context to other users. Furthermore, a user must be authenticated by the system before the system performs access control. In our implementation of capability-based access control on the Web, users can create new rights and pass them to other users.

Several protocols provide single sign-on (SSO) facilities to the Web [12] [6] . In SSO, if a user is authenticated by the first Web server, the user can log in to the next federated Web server without duplicated user authentication. In these systems, access tokens are passed among Web servers, and they resemble capabilities. However, they differ in that an access token belongs to a single person while a capability belongs to no one. A capability can be passed from one user to another.

Several APIs and protocols provide authentication services to external XML Web services components. Flickr API [11], Google AuthSub [10], and OAuth Protocol [5] allow an external Web services component to access protected resources by using tokens. In these APIs and protocols, when a user tries to access a protected resource in a base server through an external server, the external server prompts a user to visit a Web page in a base server. When a base server authenticates the user with his or her ID and password, the base server sends a token to the external server. The external Web services component accesses the user's protected resource in the base component with the token. That base server permits the external server to access the protected resources on the basis of the token. While Flickr API, Google API, and OAuth Protocol allow a user to pass his or her access rights for a protected resource to an external server, they do not allow users to pass access rights to other users. In our implementation of capability-based access control on the Web, a user who does not own a protected resource can create a new access right and distribute his or

her access rights to other users.

In 2010, Google Docs introduced a new access control setting: "Anyone with the link" [3]. In this setting, users can pass access rights for their Web resources in Google Docs to others with URLs. These URLs act like our capabilities. However, our implementation of capability-based access control in the Web has several advantages over Google Docs. First, our implementation enables users to export protected Web resources on any Web servers. Second, our implementation enables restricted capabilities to be created from base capabilities. Third, our implementation provides better time-related access control.

With CapaEdit, users can publish a new Web resource on the basis of existing Web resources. This activity resembles a mashup by end-users [25] [24]. Unlike these end-user mashups, CapaEdit deals with personal protected Web resources.

VII. Conclusion

In this study, we introduced capability-based access control to the World Wide Web. This enables users to pass access rights to other users along with delegating tasks, and makes collaborations easier than in the current Web, which uses access control based on access control lists (ACLs). Furthermore, restricted capabilities are useful in passing access rights.

In this paper, we described the implementation of capability-based access control on the Web in two Web applications: CaStor and CapaEdit plus CapaGate. In CaStor, users can create a capability from a personal protected Web resource and distribute it on a server. However, CaStor was not good enough because it did not support POST method, its expressive power using regular expressions was weak, and it provided a poor user interface. CapaEdit and CapaGate solved these problems. CapaEdit is an authoring tool running as an Ajax (Asynchronous JavaScript and XML) application and provides a better user interface. CapaEdit enables a user to write restrictions about hyperlinks and form parameters interactively. CapaGate is also an Ajax application and enforces the restrictions set by CapaEdit. These programs realized capability-based access control on the World Wide Web without modifying existing servers and clients. Experimental results show that the Web applications CapaEdit and CapaGate perform well enough for interactive use if a browser and these Web applications runs in a LAN environment.

Since CapaEdit and CapaGate are Ajax applications, they cannot deal with other Ajax applications. In a future, we would like to support Ajax applications in CapaEdit and CapaGate. We also plan to add filtering facilities to CapaEdit and CapaGate that remove unintendedly released information.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, 1986.

[2] M. Anderson, R.D. Pose, and C.S. Wallace. A Password-Capability System. *The Computer Journal*, 29(1):1–8, 1986.

[3] Official Google Enterprise Blog. New Sharing Settings in Google Docs. http://googleenterprise.blogspot.com/2010/06/new-sharing-settings-in-google-docs.html, June 2010.

[4] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, 1994.

[5] E. Hammer-Lahav (ed.). The OAuth 1.0 Protocol, 2010.

[6] S. Cantor (ed.). Shibboleth Architecture Protocols and Profiles. http://shibboleth.internet2.edu/, 2005.

[7] L. Edwards and R. Barker. *A guide for Symbian OS C++ developers*. Pearson Higher Education, 2004.

[8] The Apache Software Foundation. Apache HTTP Server Version 2.2 Documentation. http://httpd.apache.org/docs/2.2/, 2009.

[9] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 235–246. ACM, 2007.

[10] Google. Authentication and Authorization for Google APIs. http://code.google.com/apis/accounts/, 2010.

[11] C. Henderson, A. S. Cope, E. Costello, S. Mourachov, and S. Butterfield. Flickr Authentication API. http://www.flickr.com/services/api/auth.spec.html, 2008.

[12] S. Landau and J. Hodges. A Brief Introduction to Liberty. *Sun Microsystems, Inc. TR-2002-113*, 2002.

[13] H.M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[14] M. Mabuchi, S. Ikejima, S. Kawasaki, J. Yoshino, K. Matsui, Y. Shinjo, A. Sato, D. Kamikawa, and K. Kato. CaStor: A Web Server for Management and Distribution of Capabilities to Access Web Resources. *IPSJ Journal, the Information Processing Society of Japan*, 50(8):1856–1869, 2009.

[15] M. Mabuchi, Y. Shinjo, K. Hasebe, A. Sato, and K. Kato. CapaCon: Access Control Mechanism for Inter-Device Communications through TCP Connections. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 706–712, 2010.

[16] M. Mabuchi, Y. Shinjo, A. Sato, and K. Kato. An Access Control Model for Web-Services That Supports Delegation and Creation of Authority. In *Proceedings of the Seventh International Conference on Networking (ICN 2008)*, pages 213–222. IEEE Computer Society, 2008.

[17] M. Mabuchi, S. Takada, T. Ozawa, H. Toyooka, K. Matsui, A. Sato, Y. Shinjo, and K. Kato. Implementation of a Network Control Mechanism that Enables Passing Access Rights among Users. *IPSJ Journal, the Information Processing Society of Japan*, 51(3):974–988, 2010.

[18] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis. Decentralized Access Control in Distributed File Systems. *ACM Computing Surveys*, 40(3):1–30, 2008.

[19] S. J. Mullender, G. Van Rossum, A. S. Tananbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[20] J.T. Regan and C.D. Jensen. Capability File Names: Separating Authorisation from User Management in an Internet File System. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, page 17. USENIX Association, 2001.

[21] Y. Shinjo, K. Matsui, T. Sugimoto, and A. Sato. An Anti-Spam Scheme Using Capability-Based Access Control. In *Proceedings of IEEE 34th Conference on Local Computer Networks, 5th IEEE LCN Workshop on Security in Communication Networks (SICK)*, pages 907–914, 2009.

[22] S. Suzuki, Y. Shinjo, T. Hirotsu, K. Itano, and K. Kato. Capability-Based Egress Network Access Control for Transferring Access Rights. In *Third International Conference on Information Technology and Applications (ICITA)*, pages 488–495. IEEE Computer Society, 2005.

[23] W. Tolone, G.J. Ahn, T. Pai, and S.P. Hong. Access Control in Collaborative Systems. *ACM Computing Surveys*, 37(1):41, 2005.

[24] G. Wang, S. Yang, and Y. Han. Mashroom: End-User Mashup Programming Using Nested Tables. In *Proceedings of the 18th international conference on World Wide Web*, pages 861–870, 2009.

[25] J. Wong and J. I. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the ACM SIGCHI conference on Human factors in computing systems*, pages 1435–1444, 2007.

[26] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the Kernel of a Multiprocessor Operating System. *Commun. ACM*, 17(6):337–345, 1974.